Department of Informatics, University of Zürich

BSc Thesis

QR decomposition integration into DBMS

Dzmitry Katsiuba
Matrikelnummer: 14-705-354
Email: dzmitry.katsiuba@uzh.ch

July 2, 2017

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

University of Zurich[UZH]

Department of Informatics

Acknowledgements

Most of all, I would like to express my sincere gratitude to my supervisor Oksana Dolmatova for her patience, helpfulness and support. I am very thankful for the feedback and guidance she provided. Also, I would like to thank Prof. Dr. Michael Böhlen for giving me the opportunity to write my bachelor thesis at the Database Technology Group of the University of Zurich.

Abstract

The demand to analyse data stored in DBMSs has increased significantly during the last few years. Since the analysis of scientific data is mostly based on statistical and linear algebra operations (e.g. vector multiplication, operations on matrices), the computation of the latter plays a big role in data processing. However, the current approach to deal with statistics is to export data from a DBMS to a math program, like R or MATLAB. This implies additional time and memory costs. At the same time the column-store approach has become popular and a number of hybrid or pure column-store systems, such as MonetDB, Apache Cassandra etc. are available. I investigate the benefits of incorporating a linear operation into a column-oriented DBMS. For this purpose, I integrate the QR decomposition in MonetDB, analyse the complexity of the implementation and empirically compare the performance with the existing R-solution (exporting the data with UDFs). The results of experimental evaluation show, that the embedded R solution works faster than the QR integration in MonetDB, when a virtualized environment used. On an unvirtualized host MonetDB has a significantly better performance, exceeding the results of R. The implemented Q_QR function allows calculation on relations directly in the DBMS. It uses the SQL syntax, which makes the usage of the function easy and intuitive. The user doesn't require any additional skills, while writing of UDF functions for different sets of parameters means a certain programmer effort.

Zusammenfassung

Die Nachfrage nach Datenanalyse der in Datenbanken gespeicherter Information ist in den letzten Jahren deutlich gewachsen. Die Auswertung der wissenschaftlichen Daten basiert meist auf statistischen und linearen Algebraoperationen (z.B. Vektorrechnungen, Matrizen-Operationen usw.). Dabei erhält die Möglichkeit, direkt in Datenbanken diese Operationen auszuführen, ein immer grösseres Gewicht. Die verbreitete Vorgehensweise besteht darin, Daten in ein mathematisches oder statistisches Programm, wie beispielsweise R oder MATLAB, zu exportieren. Das Problem dabei: Es verursacht zusätzlichen Zeit- und Speicheraufwand.

Gleichzeitig werden spaltenorientierte Datenbanken immer populärer. Auch sind auf dem Markt reine oder hybride spaltenorientierte Datenbanken (wie MonetDB, Apache Cassandra) verfügbar.

Meine Arbeit untersucht die Vorteile der Integration von linear algebraischen Operationen in einer spaltenorientierten Datenbank. Dazu integriere ich die QR-Zerlegung in MonetDB, analysiere die Komplexität der Implementierung und vergleiche empirisch ihre Performanz mit der existierenden R-Lösung (unter Anwendung von UDFs für den Datenexport).

Die Ergebnisse meiner experimentellen Analyse zeigen, dass in einer virtuellen Umgebung die eingebettete R-Lösung eine grössere Schnelligkeit aufweist, als die QR-Integration in MonetDB. Anders verhält es sich in einer nicht virtuellen Umgebung: Da weist MonetDB eine signifikant bessere Performanz auf und überholt gar die R-Lösung.

Die implementierte Q_QR-Funktion ermöglicht die Berechnungen auf Relationen direkt in der Datenbank. Die dabei genutzte, übliche SQL-Syntax macht den Gebrauch dieser Funktion einfach und intuitiv. Dafür benötigt der Benutzer keine zusätzlichen Kenntnisse, während das Schreiben von UDF-Funktionen für verschiedene Parameterkombinationen einen zusätzlichen Programmieraufwand bedeutet.

Table of Contents

1

Introduction

1.1 Motivation

The amount of data produced by scientists has increased significantly during the last decades. The demand for structured and practical approach to processing and storing these data is thus even higher. Database management systems (DBMSs) represent an efficient storage solution with integrated set of mathematical and statistical operations (e.g. *minimum, maximum, average, count etc.*). However, scientific data often require a more complicated evaluation and computations, such as linear algebra operations, which cannot always be processed directly in DBMSs. Math and statistics programs, like R or MATLAB, allow eluding the lack of statistical functionality in popular DBMSs.

At the same time column-oriented DBMSs have attracted a lot of attention. These databases differ in the way information is stored, namely the attribute values belonging to the same column are stored separately, contiguously and densely packed, while the traditional row-oriented database systems store the entire records (rows) one by one [1]. The column-store approach has become popular and a number of hybrid or pure column-store systems, such as MonetDB, Kdb, VectorWise, Infobright, Exasol are now available. Column-oriented systems have some important advantages compared to row-stores, e.g. compression of the repeating values in columns, working with linear sets of values, especially when those sets are much larger than available RAM. Specifically due to these advantages column-oriented databases offer a great flexibility for analytical workloads.

The goal of my thesis is to expand MonetDB functionality by integrating one of the most well-known linear algebra operations, the QR decomposition (QRD), and to investigate the benefits of incorporating a linear operation into a column-oriented DBMS.

1.2 Problem definition

Many mathematical and statistical problems arising during scientific research are solved with the help of linear algebra operations, which require matrix and array operations. QRD is one of the popular operations. Calculations of eigenvalue algorithm, the QR algorithm, the linear least squares problem and many other problems are often solved

using the QRD. Unfortunately, it doesn't belong to the standard set of database functions and additional steps for its execution are required. Moreover, the QRD, as any linear operation, is defined over matrices, but not relations. A relation should be "adjusted" in order to be suitable for the QRD. The current approach to perform this and other statistical calculation is to export data from a DBMS, perform the operation using a math tool and after that import the results back into the DBMS. This process implies additional time and memory costs. Moreover, it is a potential source of errors. The integration of the QRD in a column-oriented DBMS can be done by implementing a new function, which returns the matrix Q from the QRD (Q_QR function). This approach allows doing the decomposition directly in the database, using existing relations, without additional export and import operations. It also saves resources and excludes possible non-calculation errors.

2

Related Work

In this section I give an overview about QRD algorithms, describe DBMSs that allow working with linear algebra operations. I also mention the possibility of processing these operations on database relations.

2.1 QR Decomposition

QR Decomposition (QRD) of a matrix, also known as the QR factorization, is a decomposition of matrix A into a product $A = QR$, where Q is an orthogonal matrix, and R is an upper triangular matrix. I assume, that the matrix A is of size mxn, where m is a number of rows, n is a number of columns and $m \geqslant n$. QRD is used for solving several mathematical problems, such as: computation of matrix inversion, determination of the numerical rank of a given matrix, singular value decomposition, least squares problem, etc.

Algorithm 1 Classical Gram-Schmidt algorithm [2]

1: **for** k:=1 **to** n **do**
2: **for** i:=1 **to** k-1 **do**
3: $s := 0$
4: **for** j:=1 **to** m **do**
5: $s := s + a_{j,i} * a_{j,k}$
6: $r_{i,k} := s$
7: **for** i:=1 **to** k-1 **do**
8: **for** j:=1 **to** m **do**
9: $a_{j,k} := a_{j,k} - a_{j,i} * r_{i,k}$
10: $s := 0$
11: **for** j:=1 **to** m **do**
12: $s := s + {a_{j,k}}^2$
13: $r_{k,k} := sqrt(s)$
14: **for** j:=1 **to** m **do**
15: $a_{j,k} := a_{j,k}/r_{k,k}$

There are two common algorithms for executing QR-decomposition:

- Householder transformations
- Gram-Schmidt algorithm (see Algorithm 1)

These algorithms differ in terms of performance and possibility of parallelizing their execution. Transforming parts of the classical Gram-Schmidt algorithm into *euclidean norm* and *dot product* functions produces the vector-based version of the algorithm (see Algorithm 2). The algorithm normalizes the vectors in sequence, doing orthogonalization for the rest of the vectors after every normalization (reorthogonalizing them by already normalized vectors).

Algorithm 2 Vector-based Gram-Schmidt algorithm

```
1: for k:=1 to n number of vectors do
2:     r_{k,k} := euclidean norm(a_k)                ▷ normalization
3:     q_k := a_k / r_{k,k}
4:     for j:=k+1 to n do
5:         r := dot product(q_k, a_j)                ▷ orthogonalization
6:         a_j := a_j − r * q_k
```

The reorthogonalization doubles the computational effort and for this reason the method of Householder is usually preferred. The advantage of Gram-Schmidt algorithm is that it is possible to perform the orthogonalization of one vector to as many other vectors as it is needed [2]. If, as according to the task (see Chapter 3), only the matrix Q is needed, then the modified Gram-Schmidt algorithm is much more efficient [9].

2.2 Matrix operations in a DBMS

Data-intensive research fields rely on the ability to efficiently process massive amounts of experimental data using database technologies. A DBMS works mostly with relations and not with arrays or matrices, as it is often necessary for scientific purposes. Outsourcing those computations to other programs represents a solution for this computation gap.

The most DBMSs have only a limited set of algebra operations and functions, such as $count, minimum, maximum, average$ etc. These are functions that are not dependent on the order of the values in the attributes. More complex operations that respect the order of attributes are not built-in and can only be performed using user-defined functions (UDF), embedded packages or by exporting the data into external solutions. One of the biggest relational database Oracle has an R Enterprise solution, which integrates R with Oracle Database [10]. MonetDB has an embedded R package, which also allows using the full functionality of R inside of MonetDB. However, this approach needs programming effort and involves writing of single UDF function for each set of input parameters (e.g by changing the number or data type of input columns, etc.). The

implemented Q_QR function allows processing the linear algebra operation directly in DBMS, without exporting and re-importing of the data.

To bridge the gap between the needs of the scientific world and the current DBMS technologies, the first SQL-based query language (SciQL) for scientific applications was introduced. SciQL's key innovation is the introduction of an *array* type, which works on the same level as a *table*. It uses both tables and arrays as first-class citizens and provides relational algebra-like operations on them [15]. The difference between these two types is their structure: tables are represented by a set of tuples, while an array is identified by attributes (dimensions) and their constraints. Combination of indeces (values of these attributes) denotes a cell, where a value of one non-dimensional attribute is stored. Thus, one tuple from a table is represented in arrays by a set of cell indices and the non-dimensional value in this cell. SciQL uses MonetDB as the target platform. Its column-store architecture corresponds well to the array representation, which significantly reduces the impedance mismatch between query language and array manipulation and, as a result, the development effort [16]. To address arrays in queries, the user applies the table syntax. That makes the transition from SQL to SciQL easier [6]. The implemented Q_QR function allows working directly on relations, without paying attention to the array representation or learning a new query language.

SciDB is an open-source analytical database that provides not only data management, but also complex analytics. In contrast to SciQL, SciDB was completely new developed. SciDB is oriented towards the data management needs of scientists, fulfilling their requirements regarding complex analytics and working with the large data sets [14]. As such it mixes statistical and linear algebra operations with data management operations (e.g. create, delete, update values, managing of constrains, etc.). SciDB takes natural nested multi-dimensional array data model as basis that eliminates the conversion between tables and arrays. For this goal a new functional (AFL) and a SQL-like query (AQL) languages were invented. SciDB provides an extensibility framework (for user-defined data types, functions, aggregates but also array operators). That makes it possible to apply highly specific algorithms in conjunction with some pre-processing handled by SciDB's built-in or user-defined array operators [13]. In contrast to SciDB for employing the new defined function in MonetDB, the user doesn't need to learn a new query language or deal with a new data model. The Q_QR function runs directly on existing relations.

3

Task description

In this chapter I describe the function that has to be implemented, including its structure, input parameters and constraints. I give a running example for the input relation, on which I show the defined parameters. Finally, using the running example, I define the expected results of the function.

The goal of this thesis is to integrate the QR decomposition (QRD) into column-oriented DBMS by extending it with the new Q_QR function. So that afterwards the following SQL query can be processed:

$$\begin{aligned} SELECT\ *\ FROM\ Q\_QR\ (relation\_name\ ON\ on\_attributes \\ ORDER\ BY\ order\_by\_attributes); \end{aligned} \tag{3.1}$$

This query returns a relation with the same schema as the input relation and represents the Q matrix from the QRD. Since I want only the Q matrix as the result, QRD can be implemented according to the Gram-Schmidt Algorithm (see Chapter 2.1). Figure 3.1 illustrates the structure of the input relation.

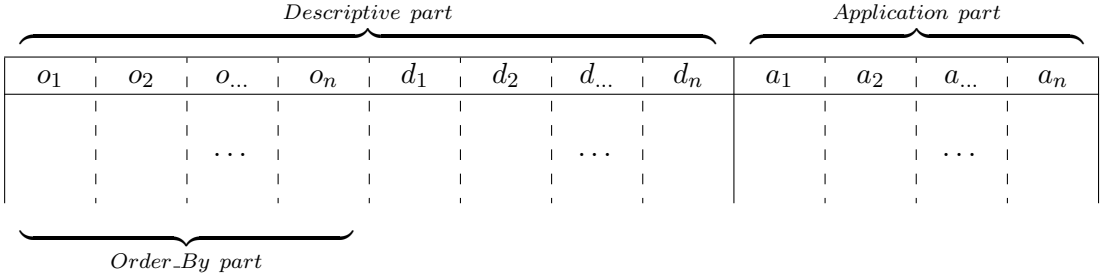

Figure 3.1: Structure of the input relation

The application part (in (3.1) *on_attributes*) includes the set of only numeric-domain attributes, which represents the matrix A for the QR decomposition (see Chapter 2.1). The relationship between the size of the application part and the whole number of attributes in the relation forms the ratio of the application part. Attributes, which are not in the application part, form the descriptive part of the input relation. The Order_By part is a subset of descriptive part (in (3.1) *order_by_attributes*). It includes

the attributes, the tuples in the output relation and the corresponding matrix A for the QRD have to be sorted by. The attributes in Order_By part can also be of non-numeric data types. The ratio of the Order_By part is a relationship between its size and the number of attributes in the descriptive part.

For the implementation I use MonetDB-11.23.13 (Jun2016-SP2). The column-oriented architecture of MonetDB works with the attributes as arrays. It allows me to use the vector-based version of the algorithm (see Algorithm 2), which handles the values of one attribute together as one vector.

The DBMS has to recognize ambiguous or not existing names of attributes and the input relation in the passed query, and, if applicable, it returns an appropriate error message. The input relation can be an existing relation or a result of an arbitrary select subquery.

As a running example I take a relation r, which is displayed in Figure 3.2.

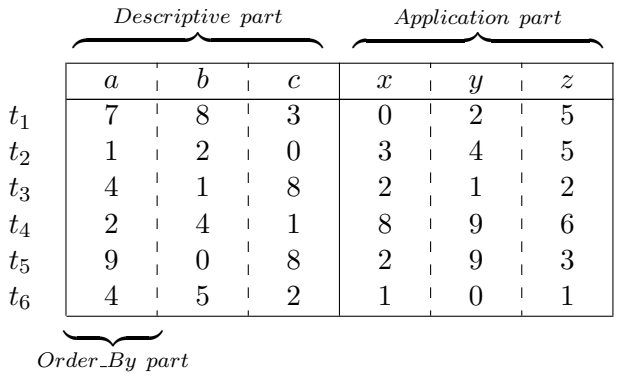

Figure 3.2: Relation r

An example SQL query for this relation is shown in Query (3.2)

$$SELECT\ *\ FROM\ Q\_QR\ (r\ ON\ x, y, z\ ORDER\ BY\ a); \tag{3.2}$$

QRD is performed over 3 attributes (x, y, z) from the application part of r. After ordering the attributes by a, the relation r and the matrix A look as shown in Figure 3.3.

	a	b	c	x	y	z
t_2	1	2	0	3	4	5
t_4	2	4	1	8	9	6
t_3	4	1	8	2	1	2
t_6	4	5	2	1	0	1
t_1	7	8	3	0	2	5
t_5	9	0	8	2	9	3

$$A = \begin{bmatrix} 3 & 4 & 5 \\ 8 & 9 & 6 \\ 2 & 1 & 2 \\ 1 & 0 & 1 \\ 0 & 2 & 5 \\ 2 & 9 & 3 \end{bmatrix}$$

Figure 3.3: Relation r ordered by attribute a and the corresponding matrix A

a	b	c	x	y	z
1	2	0	0.33129	0.02729	0.43491
2	4	1	0.88345	-0.16036	-0.15479
4	1	8	0.22086	-0.21495	0.15105
4	5	2	0.11043	-0.17742	0.11223
7	8	3	0	0.27978	0.83288
9	0	8	0.22086	0.90419	-0.24037

Figure 3.4: Results of the query (3.2)

The Order_By part in (3.2) is presented only by attribute a, the ratio of Order_By part is thus 1/3, while the ratio of the application part is 50%.

Figure 3.4 demonstrates what the results of the Q_QR function applied on the relation r has to look like.

4

Approach

In this chapter I give a description of the architecture and features of the used DBMS (MonetDB). Afterwards I describe the necessary implementation steps.

4.1 MonetDB

MonetDB is a column-oriented database management system developed at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. It is actively used in businesses such as health care, telecommunications as well as in sciences such as astronomy. Innovations at all layers of a DBMS such as storage model based on vertical fragmentation, a modern architecture of CPU-tuned query execution, different ways of optimizations, adaptive indexing and a modular software architecture allow MonetDB to achieve significant speed up compared to traditional designs [5]. It supports SQL:2003 standard on client side.

Architecture

MonetDB has a three-level software stack:

SQL front-end: The top layer of the architecture provides the access point for the user. The task of front-end is to map the user's data to the MonetDB's internal structure (BAT) and to translate user queries to MonetDB Assembly Language, known as MAL. The user query is transformed from a SQL query to an internal relational algebra representation, which is then optimized using domain-specific rules and heuristics (e.g. reducing the size of intermediate results, by pushing selections down in the tree). This represents the strategic optimization of the parsed query [5].

At this level I need to extend the parser with the new syntax, and ensure that all internal transformations and the strategic optimizations are done.

Tactical-optimizers: In the middle layer the resulting MAL-plan is going through a row of tactical optimizations. It is inspired more by the programming language than by classical database query optimization. Each optimizer, sprinkled with resource management or flow of control directives, takes a MAL plan and transforms it into a more efficient one. It provides facilities ranging from symbolic processing up to just-in-time

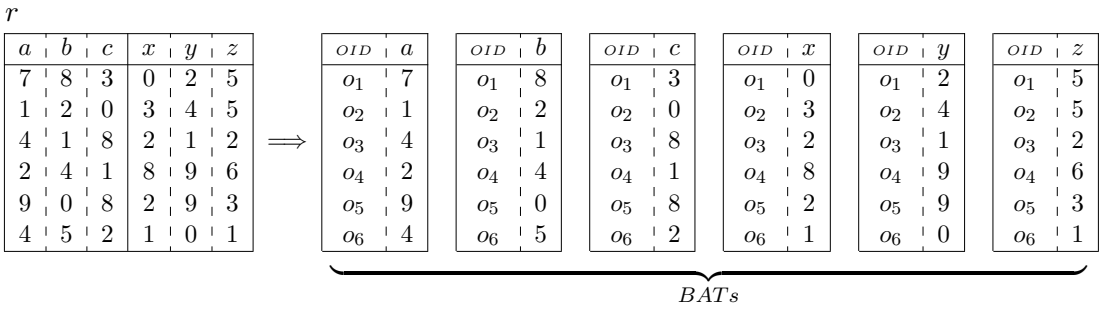

Figure 4.1: Internal (BAT) representation of relation r

data distribution [5].

Columnar abstract-machine kernel: The bottom layer of MonetDB stores each attribute of a relation as columns (in fact as arrays) known as BAT. It includes the library of highly optimized implementations of the binary relational algebra operators, stored in corresponding BAT modules. Those operators have access to the whole metadata of BATs, it allows them to perform operational optimization. The interface is described in the MAL module sections.

For the implementation of the task I can reuse already existing BAT algebra modules. Nothing additional needs to be done at this level for the purpose of optimization.

BAT

MonetDB stores each attribute of a relation in a separate table, called a BAT (Binary Association Table). Each BAT consist of two columns: the head column, where the object-identifiers (OID) are stored, and the tail column, where the actual attribute values are stored. Due to this approach every relation is internally represented as a collection of BATs. Physically BATs are represented as consecutive C arrays. They can be up to hundreds of megabytes. The operating system swaps them into memory and compresses on the disk upon need [3].

The example relation r consists of 6 attributes, for these attributes 6 BATs exist. Each BAT stores the respective attribute as (OID, value) pairs. The internal representation of r is illustrated in Figure 4.1. OIDs (o_1, o_2, ...,o_6) are generated from the system and identify the tuple the value belongs to. The values from the same tuple are assigned the same OID (e.g. in 6 BATs that represent relation r values 7, 8, 3, 0, 2, 5 at the very top of each BAT have the same OID o_1. That means, that all of them belong to the same tuple t_1).

The head OID column is not materialised, but rather implicitly given by the position of the value, so all values of one tuple will be on the same position in their respective column representation. The position, in turn, is determined by the inserting order of tuples [5].

Relational algebra plans are translated into simple BAT algebra operations and com-

piled to MAL programs. BAT algebra operators perform simple operations on an entire column of values ("bulk processing").

MAL

MonetDB Assembly Language (MAL) is the language used for the abstract machine of the MonetDB kernel. It is the target language for front-end query compilers. Every SQL query generates an execution plan, consisting of simple MAL instructions. These instructions represent all actions, which are necessary in order to provide and deliver the results (e.g. actions to ensure binding data and transaction control, the instructions to produce the results, and the administration steps for preparing and delivering the resulting relation to the front-end). BAT algebra operators correspond to simple MAL instructions, thus building its core. MAL instructions have only BATs as input, on which corresponding simple BAT operations are performed. Complex operators can be broken down into a sequence of BAT algebra operators, which in their turn are mapped to simple operations on arrays. This approach allows avoiding additional expression interpretations.

An example of such implementation of a select operator in MonetDB is shown in Figure 4.2 [5]. The select value (V) is compared to the values, stored in the tail part of the input BAT (B), which in its turn is stored as a C-Array. If the value is found, its OID is stored in the tail part of the resulting BAT (R).

```
                                                  for ( i=j=0 ; i<n ; i++ )
select (B:bat:[:oid:int], V:int)    ⟶              if ( B.tail[i] == V)
                                                      R.tail[j++] = i;
```

Figure 4.2: Implementation of select operation

The for-loops containing no external function calls provide high instruction locality. That enables better compiler optimization and leads to reduction in the number of instruction cache misses.

MonetDB provides an internal function to show the MAL-Plan of a passed query. The operations in it are executed one at a time. That means, that subsequent operations are invoked only after the operation completed the calculation over its entire input data. The idea of BAT operations is always to have a materialized result after its execution.

Query Processing

Symbol tree: A query passed into the client is parsed in order to detect the specified keywords (tokens). The tokens help the Yacc parser generator to generate a shift-reduce parser. The input to Yacc is a grammar (sql_parser.y) with snippets of C++ code ("actions") attached to rules of the grammar. As soon as the rule is recognised, the parser

calls the code snippets associated with this rule. During the execution of these snippets nodes of different types are built and connected together in a symbol tree. Symbol tree represents only the structure of parsed query.

Relation tree: The resulting symbol tree is used to build a relation tree. The relation tree consists of nodes, representing a single operation on input tables, containing necessary information for this operation. The nodes of the symbol tree are parsed, analysed, grouped together and converted into corresponding relation tree node, depending on tokens in them. Each node in the relation tree may have up to two children. Afterwards the strategic optimization is carried out.

Statement tree: A statement tree is a special feature of MonetDB. It is an intermediate step between a relation tree and a low-level execution plan (MAL-Plan). Comparing to the resulting relation tree, which nodes represent operations on or between relations, each statement in the statement tree represents the sequence of operations on attributes and returns one BAT as a result. Each operation is represented by a MAL-expression that in its turn corresponds to simple BAT algebra operations.

Putting it all together, MonetDB provides a good extensibility framework. That allows to extend its functionality, SQL syntax and to make changes at all levels of architecture.

The BAT structure using the OID guarantees, that the values from the same tuple belong together, despite the order of the tuples in BATs. This mechanism allows algebra and statistical operations, for which the order and the relation between the values are essential.

Breaking down the operations to the simple operations on arrays and the set of existing BAT algebra operations provide a good basis for implementation of sophisticated and complex calculations.

4.2 Implementation

In this subchapter I describe the changes in MonetDB, which has to be done in order to implement the task query. I also show the examples of trees built by MonetDB during the processing of Q_QR function, using the example SQL query (3.2).

4.2.1 Symbol tree

To parse the task query I expand the existing parser grammar by a new token Q_QR. For processing the new token I define additional rules and snippets of code for these rules. The extended part of the grammar can be found in Appendix A.1. Figure 4.3 illustrates the symbol tree, which is produced after parsing and executing the corresponding code.

SELECT * FROM Q_QR (r ON x, y, z ORDER BY a);

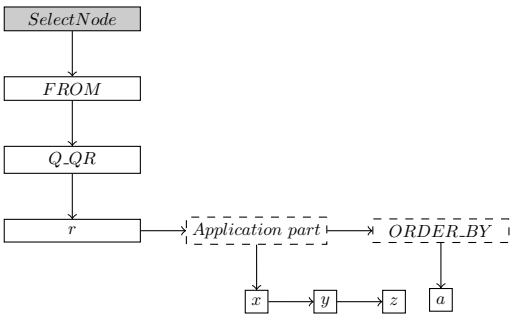

Figure 4.3: Symbol tree for the query (3.2)

The symbols are build according to the tokens recognized in the passed query. *SelectNode* is a route symbol-node for all SELECT-queries. It is connected to all symbols containing the information, which is necessary to produce the results from the input relation and to deliver the needed result relation (e.g. symbols like $WHERE$, $GROUP\_BY$, $ORDER\_BY$, $HAVING$ etc.).

The query (3.2) produces the *SelectNode* connected only to one symbol namely one with the information about the source relation ($FROM$). The new Q_QR token also produces a symbol, connected to the symbols with input parameters: relation r, *Application part* and $ORDER\_BY$. The last two parameters are lists, containing the $COLUMN$-symbols. The *Application part* contains the columns (x, y, z) for the matrix used during the QRD and $ORDER\_BY$ contains the column (a) for determining the order of values in the resulting relation. The information in $COLUMN$-symbols must correspond with the attributes in the input relation r. The result of the Q_QR-symbol represents the only source relation for the $FROM$-symbol.

4.2.2 Relation tree

To implement the task I add a new function that deals with the new Q_QR-symbol in the symbol tree. This function uses the Q_QR-symbol as well as the related symbols with corresponding information to build a proper relation tree node for the Q_QR function. The resulting relation tree has tree nodes, as shown in Figure 4.4.

SELECT * FROM Q_QR (r ON x, y, z ORDER BY a);

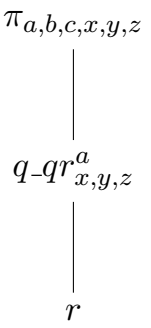

Figure 4.4: Relation tree for the query (3.2)

After conversion of symbol tree nodes into relation tree nodes the information about input relation r is stored in a separate node. For storing the lists of columns for QRD (x, y, z) and ordering (a) I define a new type of a relation tree node (q_qr relation).

The resulting relation tree corresponds to the relation plan, produced by the internal MonetDB function (see Appendix B).

4.2.3 Statement tree

To fulfil the translation of the new q_qr relation into MAL-Plan and make the calculation of the QRD possible, I need to extend the existing translation function. It takes the information from the q_qr relation and, following the steps of the vector-based Gram-Schmidt Algorithm, translates it in a sequence of statements. The pseudocode of the translation is shown in Algorithm 3. The extended part of the function can be found in Appendix A.2.

The resulting sequence of statements can be also presented in a tree form. Figure 4.5 displays such a statement tree for the q_qr relation of the query (3.2). It shows both the sequence and the origin of the data required for each particular statement. Each statement tree node is a single BAT, representing the results of the done statement calculations.

At the beginning I use the information from the q_qr relation in order to create two lists of BATs for the descriptive (a, b, c) and the application parts (x, y, z) of the input relation. Using the Order_By part (a), I determine the order of tuples in the output relation. For this purpose I reuse the operations of the existing statements for ordering and reordering of data in BATs. These statements are translated without changes into

Algorithm 3 Q_QR

```
1: procedure Q_QR(relation, on_attrs, order_by_attrs)
2:     attrs ← attributes of relation
3:     descriptive_part ← attrs − on_attrs
4:     if order_by_attrs ∃ then
5:         order_by ← order of order_by_attrs
6:         for descriptive_attr in descriptive_part do
7:             sort descriptive_attr by order_by
8:             add descriptive_attr to list_ output
9:         for on_attr in on_attrs do
10:            sort on_attr by order_by
11:            add on_attr to on_attrs_sorted
12:    rest_attrs ← on_attrs_sorted
13:    for on_attr in on_attrs_sorted do
14:        normalize on_attr
15:        add on_attr to list_output
16:        rest_attrs ← rest_attrs − on_attr
17:        for rest_attr in rest_attrs do
18:            orthogonalize rest_attr to on_attr
       return list_output
```

low-level MAL-plan and represented through existing BAT algebra operations. The ordering statement takes the BAT, by which it must be sorted, as input and returns a BAT containing the determined order of tuples ($O_{list\_order\_attributes}$ in Figure 4.5 O_a).

After that the attributes in the descriptive and the application parts are ordered by this BAT. As a result a new ordered BAT for each of the attributes is produced ($O_{order\_by\_attrs}(on\_attr)$ in Figure 4.5 $O_a(a)$, $O_a(b)$, ..., $O_a(z)$). The ordered results of the descriptive part are directly appended to the output list. The ordered application part is stored as an intermediate list that is used for the Q_QR calculation afterwards.

Q_QR calculation is defined by a sequence of normalization and orthogonalization operations on attributes. The operations of the corresponding statements and their results are not implemented in MonetDB and have to be defined. Detailed description of how these two operations work and how the resulting BATs ($Norm(attribute)$ and $Orth_{attribute_2}(attribute_1)$, e.g. in Figure 4.5 $Norm(x)$ and $Orth_x(y)$ or $Orth_y(z)$ respectively) are produced is given in the Chapter 4.2.4.

I append the BAT, representing the normalized attribute, to the output list. After all operations have been done and all attributes have been appended, the output list is passed as an argument to the next node of the relation tree. The projection of the results to the user is processed by existing statements, and doesn't require changes at this step.

The MAL-Plan constructed for the query (3.2) is shown in the Appendix C.1.

SELECT * FROM Q_QR (r ON x, y, z ORDER BY a);

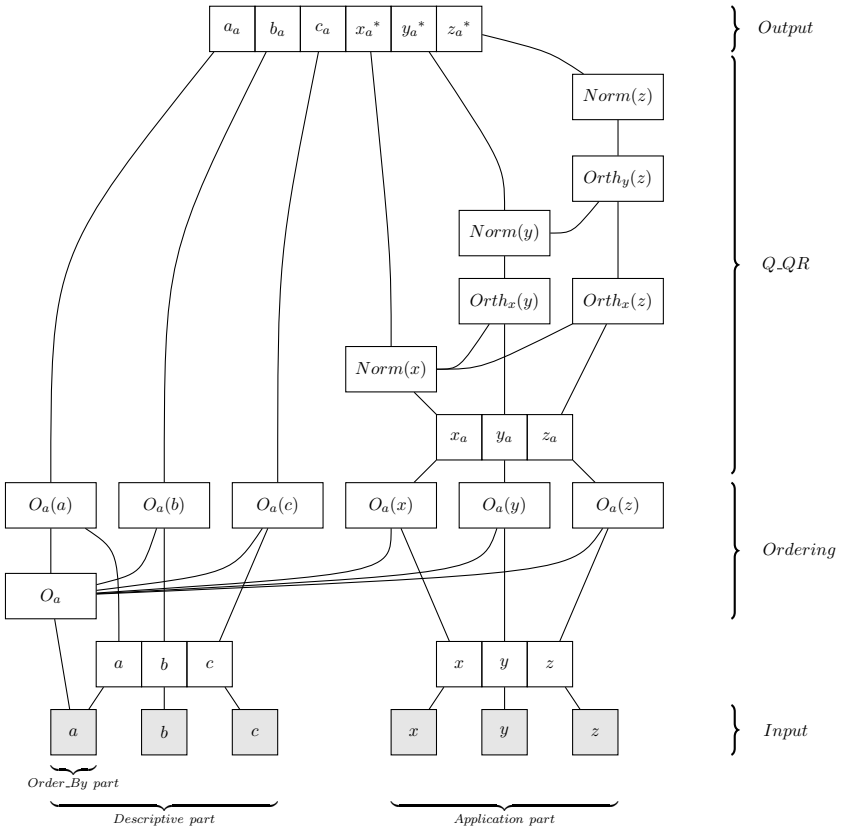

Figure 4.5: Statement tree for the query (3.2)

4.2.4 Translation to MAL-Plan

Two additional statements, required for the implementation of Gram-Schmidt algorithm and used by me for the translation of the q_qr relation into low-level MAL instructions, have to be implemented. These statements execute the operations of normalization and orthogonalization on attributes (see Algorithm 2).

To get the results of these two statements, I use BAT operations from existing BAT modules, such as:

- algebra calculations (multiplication, exponentiation, division, subtraction)
- aggregate functions (sum)
- mathematical functions (square root).

Both statements run over the ordered application part of the relation that is stored after the ordering operation in an intermediate list.

Normalization: Normalization statement has one BAT as input, which has to be normalized. First the euclidean norm for the attribute is calculated. The calculation is performed by multiplying the corresponding BAT of the attribute by itself. The values in the resulting BAT are summed and the square root of the sum is taken.

Each of the aggregate ($SUM()$) and mathematical ($SQRT()$) functions returns a single number, stored as a separate temporal BAT. This BAT is used in subsequent calculations later.

Division of each attribute value by its euclidean norm delivers the normalized form of the attribute. It is stored as a new BAT ($Norm(x)$) that is also appended to the output list of attributes.

Figure 4.6 illustrates the processes of normalization statement, on the example of the attribute x from the query (3.2).

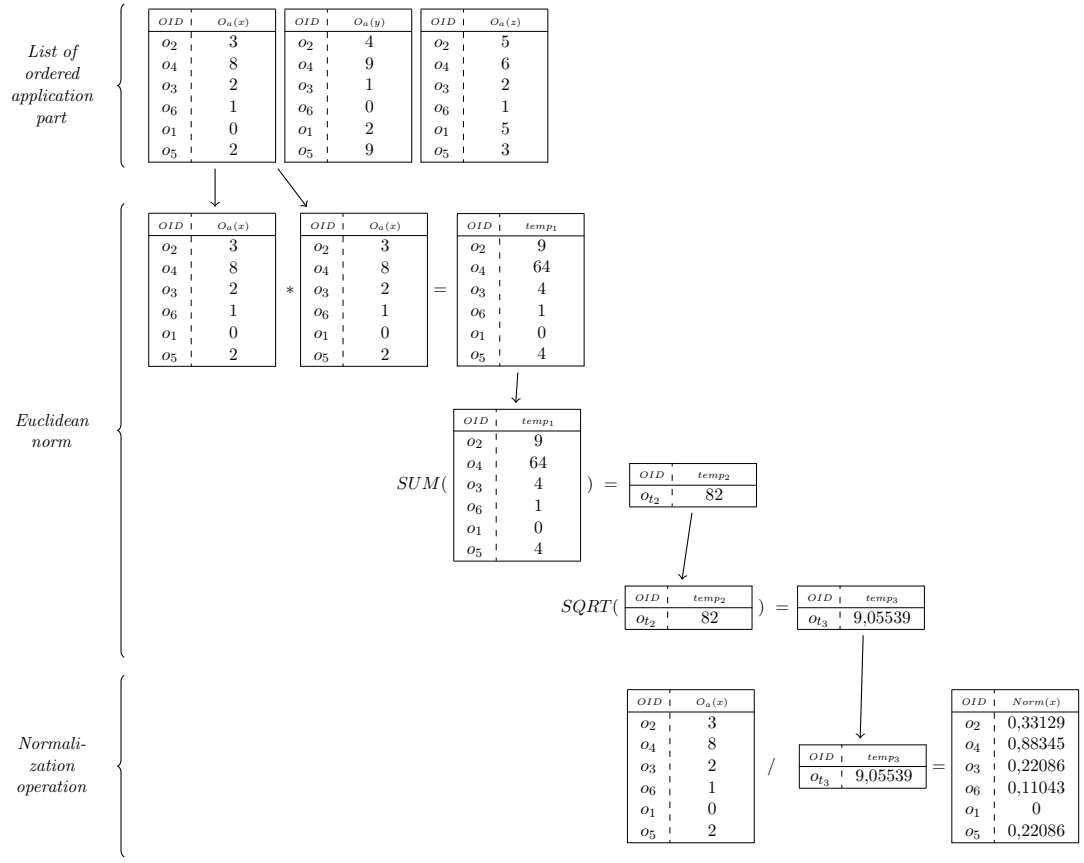

Figure 4.6: Processes of normalization statement, on the example of the attribute x from the query (3.2)

Orthogonalization: After each operation of normalization the rest of non-normalized attributes in the application part of the relation has to be orthogonolized to the last normalized vector. For this purpose the statement $Orth_{attribute_2}(attribute_1)$ takes two

attributes as an input: the attribute that has to be orthogonalized ($attribute_1$) and the attribute, which it has to be orthogonalized to ($attribute_2$).

An example of orthogonalization procedure for the attribute y (orthogonalized to attribute x) is displayed in Figure 4.7.

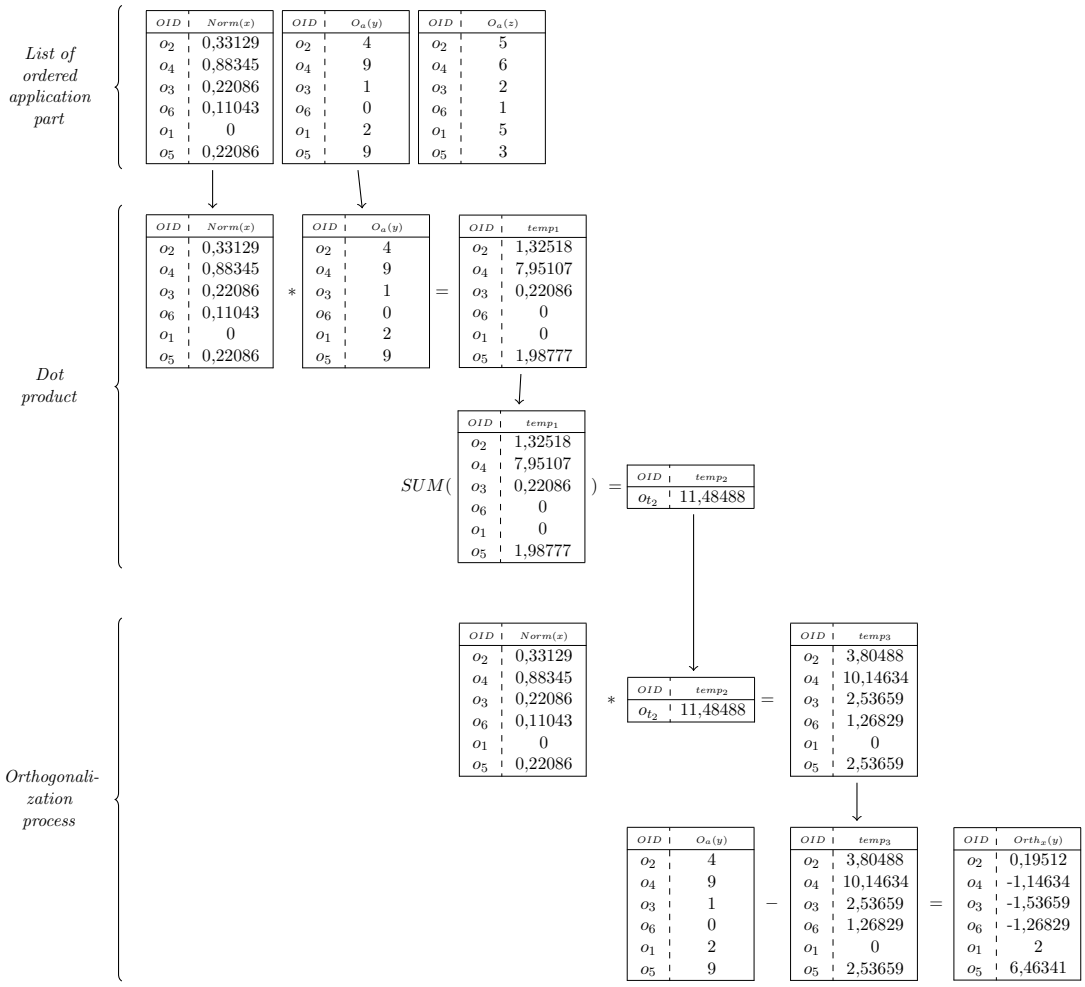

Figure 4.7: Processes of orthogonalization statement, on the example of the attribute y from the query (3.2) (orthogonalized to attribute x)

After the orthogonalization of the non-normalized attributes is completed, as shown in Figure 4.8, the next attribute can be normalized and the orthogonalization procedure iterated until there is only one attribute left.

After the normalization of the last attribute, the latter is appended to the output list, where it forms the output relation, together with the attributes from the descriptive part, as shown in Figure 3.4.

One of the advantages of the application of Gram-Schmidt algorithm for the matrix

OID	$Norm(x)$
o_2	0,33129
o_4	0,88345
o_3	0,22086
o_6	0,11043
o_1	0
o_5	0,22086

OID	$Orth_x(y)$
o_2	0,19512
o_4	-1,14634
o_3	-1,53659
o_6	-1,26829
o_1	2
o_5	6,46341

OID	$Orth_x(z)$
o_2	2,29268
o_4	-1,21951
o_3	0,19512
o_6	0,09756
o_1	5
o_5	1,19512

Figure 4.8: Stand of BATs after all attributes from the query (3.2) are orthogonalized to attribute x

Q calculation is the possibility of parallelizing its execution. Since there are no dependencies between the attributes and no overwriting operations of them during the orthogonalization, the latter can be done concurrently.

The described approach represents the processes and the order of their executions according to the used algorithm. However, during the low-level MAL-optimization of MonetDB, these processes are re-organized, so that the entire set of operations (the required number of orthogonalizations and the subsequent normalization) are performed consequently for each attribute. This is reflected in the produced MAL-Plan for Q_QR function (see Appendix C).

Such optimization helps to reduce the number of cache misses, since more information needed for all optimizations and the normalization is stored in the cache. That reduces the number of requests to the slow main memory and improves the runtime of the entire Q_QR function.

The code for the extended statements is shown in Appendix A.3.

5

Complexity Analysis

In this chapter the complexity of the original Gram-Schmidt algorithm and the implementation are estimated and compared.

5.1 Complexity of the Gram-Schmidt algorithm

The vector-based Gram-Schmidt algorithm, used for the implementation, comprises operations of normalization and orthogonalization. Using a matrix of size mxn, where m is the number of tuples and n is the number of attributes, the numbers of executions of the algorithm operations and their suboperations can be analyzed. Table 5.1 lists the estimated numbers of these operations. The normalization is executed only once for each vector, while orthogonalization is to be done for the rest (all non-normalized) vectors after every normalization. As a result, the execution number of orthogonalization equals to $(n-1)n/2$.

	Operation	Sub -operation	Number of executions
Normalization	Euclidian norm(e_n)		n
		^{2}	n(m)
		$+$	n(m-1)
		$\sqrt{}$	n(1)
	Devision by e_n		n
		/	n(m)
Orthogonalization	Dot product		(n-1)n/2
		$*$	((n-1)n/2)(m)
		$+$	((n-1)n/2)(m-1)
	Subtraction		((n-1)n/2)
		/	((n-1)n/2)(m)
		$-$	((n-1)n/2)(m)

Table 5.1: Operations in vector-based Gram-Schmidt algorithm and number of their executions

The total number of calculated QRD operations is summarized in Equation 5.1.

$$\begin{aligned} &nm + n(m-1) + 1n + nm + \\ &+\frac{(n-1)n}{2}m + \frac{(n-1)n}{2}(m-1) + \frac{(n-1)n}{2}m + \frac{(n-1)n}{2}m \end{aligned} \tag{5.1}$$

$$mn + 2mn^2 - \frac{1}{2}n^2 + \frac{1}{2}n \tag{5.2}$$

According to the simplified version of Equation (5.2), the algorithm has the total complexity of $\mathcal{O}(mn^2)$.

5.2 Complexity of the implementation

According to the task, the output relation of the implemented Q_QR function is ordered by the defined set of attributes. It means that the complexity of the Gram-Schmidt algorithm has to be extended by the complexity of additional operations of lists building, determination of order and ordering itself.

In the first step I build the lists of attributes for the application and descriptive parts of the relation, which I need to define the matrix for the QRD. For this purpose I iterate through all attributes of the input relation, what has a complexity of $\mathcal{O}(n^2)$.

The order of the attributes is to be determined according to the *order_by_attributes* passed in the query. If only one attribute is passed, the complexity is at least $\mathcal{O}(m \log m)$. If there are more than one attribute in the *order_by_attributes*, the following attributes are only used in case, if the first order_by attribute has at least two tuples with exactly same values. For ordering those tuples additional information is needed. That means, that the complexity of such additional ordering equals to $\mathcal{O}(k \log k)$, where k is the number of tuples with same values and $k \ll m$. The worst case of ordering is when all tuples of an order_by attribute have the same value ($k = m$). The complexity of ordering in this case is at least $\mathcal{O}(sm \log m)$, where s is the number of attributes with the same values.

After the order is determined, all attributes in the relation have to be sorted according to it. This results in the ordering complexity of $\mathcal{O}(m \log m \cdot n)$.

The total complexity of the Q_QR function is summarized in Equation(5.3).

$$\mathcal{O}(n^2) + \mathcal{O}(m \log m) + \mathcal{O}(m \log m \cdot n) + \mathcal{O}(mn^2) \tag{5.3}$$

For the met assumption, that $m \geqslant n$, the total complexity of the Q_QR function equals to $\mathcal{O}(mn^2)$. If it is assumed, that $m \ggg n$, the effort for ordering overtakes the cost of the Q_QR calculations, what leads to the resulting complexity of $\mathcal{O}(m \log m \cdot n)$.

6

Experimental evaluation

In this chapter I describe the experimental evaluation. It includes the explanation of the setup, test cases, data set, the results of the experiments and their discussion. The goal of the experimental evaluation is to analyse the complete runtime of the solution and to compare the performance of MonetDB with the existing R-solution empirically. Additionally to the complete runtime I analyse time spent on building the execution plan and for Q_QR execution itself.

6.1 Setup

The experiments are carried out on an Ubuntu (16.04 LTS) virtual machine having 5,5 GB RAM. The virtual machine (5.1.14) was running on MacOS Sierra (10.12.4) having 2.70 GHz Intel Core i5-5257U CPU and 8 GB 1867 MHz DDR3 memory.

To check the environmental dependency of MonetDB I use a server with 2 x Intel(R) Xeon(R), CPU E5-2440 0 @ 2.40GH, 64GB main memory, hard disks 4x320GB, SATA, 15000rpm and OS: CentOS 6.4.

6.2 Test Case Selection and Metrics

The implemented Q_QR function has only one input relation and three additional parameters, which influence the execution performance and can be manipulated during the evaluation (see Figure 3.1):

1. number of tuples in the input relation (m)
2. number of attributes in the application part (n_a)
3. number of attributes in the descriptive part (n_d)
4. ratio of the Order_By part in the descriptive part ($ratio$)

The distribution of values is irrelevant. To get a different size of the Order_By part of the relation I combine different Order_By ratios with the size of the descriptive part.

The input relation itself is an existing relation or a result of a passed sub-query. Since the processing of sub-query is not influenced during the implementation and it is not directly connected with the performance of the Q_QR function, I refrain from manipulating this parameter.

Experimental evaluation consists of two separate experiments:

- *MonetDB vs. R*: It is a comparison between the implementation in MonetDB and the existing R-solution. For this experiment, the embedded R package for MonetDB is used and the data is exported to R via UDF (see Appendix D).
- *Internal*: It provides the detailed information about the time consumption of the implementation function in MonetDB

Detailed information on chosen values of input parameters for these experiments is presented in Table 6.1.

Parameter	Values
m	100'000, 500'000, 1'000'000
n_a	20, 40, 60, 80, 100
n_d	1[1], 10, 50 [2]
$ratio$	100%, 30%, 60%, 90% [3]

Table 6.1: Values of the input parameters for experimental evaluation

A combination of all four parameters represents a test case to be performed. For each of them a complete runtime is measured using the internal MonetDB timing function. For the *Internal* evaluation I record the time spent on creation of execution plan (preparation time), additionally to the complete runtime. The combination of complete runtime and preparation time provides the information about execution time.

In order to evaluate time spent on ordering the relation, I run every defined test case in two modes (*with* and *without ordering*). For the *without ordering* mode I optimize the Q_QR function so that the order is not determined and no ordering of attributes is performed. For each test case the difference between runtimes in two modes provides the information about time spent on ordering all attributes in the relation.

6.3 Data Set

Every single test case has 3 runs that are performed one by one. The mean value is calculated based on the measurement results of each test case run.

For each test case a new relation with m tuples and n attributes ($n = n_a + n_d$) is generated. MonetDB and R solution use the same relation by each run.

[1] Descriptive part with 1 attribute has always *ratio* of 100%

[2] In the *without ordering* mode the descriptive part consists of only 1 attribute

[3] In the *without ordering* mode the ratio is for all test cases 100%

By executing the same query more than one time, the execution plan is created only once and stored in the cache. To avoid its reusing for measuring the preparation time 3 runs are done on different relations.

Test cases, where the Order_By *ratio* parameter is manipulated, are performed on the same relation.

Queries for creating and filling tables are generated using a python script program that takes random integers between 0 and 10000.

6.4 Results

In this section I present the test results for both experiments. At first I present the results of the tests in *with ordering* mode, which are followed by the results of the *without ordering* mode. The measurements are evaluated and comparison and evaluation of the results is given. Detailed test results are listed in Appendix E.

6.4.1 *With ordering* mode

In this part of the experimental evaluation I test the implemented solution in *with ordering* mode. The order of data in a relation can be essential for some statistical and linear algebra calculations, so the resulting relation is sorted by its Order_By part.

MonetDB vs. R

The first set of test cases compares the performance of MonetDB with existing R-solution. That is done using UDF and embedded R package in MonetDB. The results of test cases with ordering by one attribute are illustrated in Figure 6.1. It is recognisable, that for R the number of tuples doesn't play a significant role, while for MonetDB it is a decisive parameter.

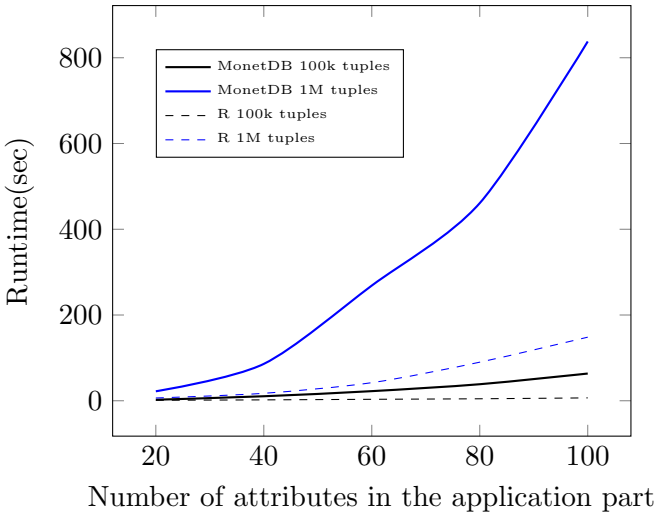

Figure 6.1: Complete runtime in the *MonetDB vs. R* experiment (ordered by 1 attribute)

The performance of R is over 4 times better than the performance of MonetDB for all combinations of the number of tuples and the size of the application part. And the more attributes in the application part are taken, the bigger is the absolute performance difference. By increasing the number of attributes in the application part and using the high number of tuples in the relation, the complete runtime in MonetDB changes

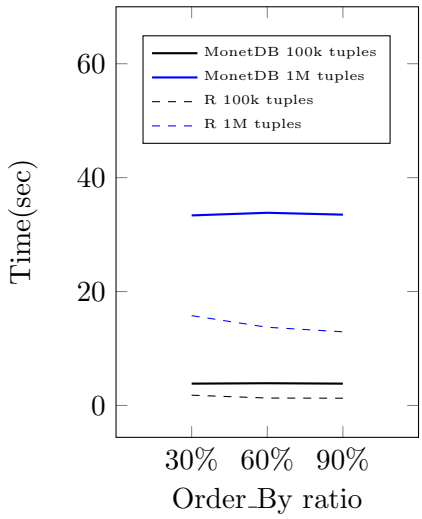

Figure 6.2: Complete runtime in the *MonetDB vs. R* experiment with 20 attributes in the application part, 10 in the descriptive part and different Order_By ratios

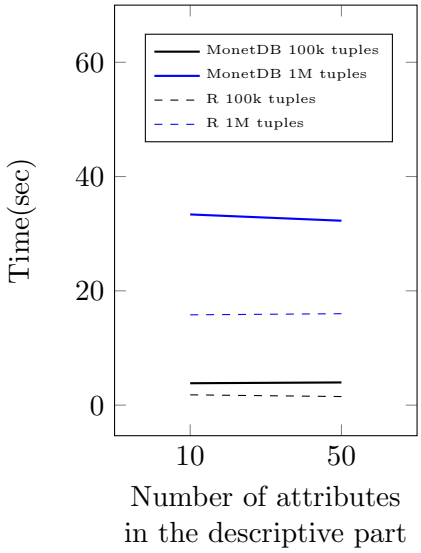

Figure 6.3: Complete runtime in the *MonetDB vs. R* experiment on relations with 20 attributes in the application part, 30% Order_By ratio and two different descriptive part sizes

quadratically. This result corresponds to the estimated complexity of the implementation for $m > n$ ($\mathcal{O}(mn^2)$).

At this point it is still unclear if the reason for this might be a difference in preparation, sorting or calculation processes. That could be better investigated during the *Internal* evaluation or during further tests in *without ordering* mode (see Chapter 6.4.2).

To evaluate how the Order_By ratio affects the performance of both solutions, I additionally change it for the fixed size of the descriptive and the application parts of the relation. Figure 6.2 demonstrates the results of tests for three different Order_By ratios (30%, 60%, 90%). We can see, that the ratio doesn't have much influence on the complete performance, since it is relevant only during the processes of order determination. The runtime in all the three test sets is almost identical.

Since the values are generated randomly, the number of tuples with the same value depends on the correlation between the number of tuples in the relation and the range for generating values. If the number of equal values is low, only a small amount of additional Order_By attributes deliver the necessary information, needed for determining the final order.

The same principle explains the results of test cases, where I manipulate the size of the descriptive part, by the fixed size of the application part and Order_By ratio. The results of these tests are shown in Figure 6.3. In the first case the resulting number of

Order_By attributes is equal to 3, and in the second case it equals to 16. For a small number of attributes it has practically no influence on the runtime. For a higher number of tuples the number of identical values increases, which leads to determination of order using additional attributes and to longer runtime. And the lager is the list of Order_By attributes, the longer the determination of order can last.

More information on the structure of time consumption in MonetDB is provided in the *Internal* evaluation.

Internal

During the *Internal* evaluation I record the time spent on different stages of the query execution: preparation, ordering and Q_QR calculation itself. This provides an overview of the change in the time consumption structure in MonetDB as a result of change of the input parameter's values.

First, I analyse the time spent on preparation. This stage includes the parsing of the passed query, the construction of symbol, relation and statement trees, the translation of statement tree into the MAL-Plan. I fix the number of attributes in the application and the descriptive parts and change the number of tuples in the relation. As expected, this has no influence on the preparation time.

For the next test I fix the number of tuples in the relation and change the number of attributes in the application part. Figure 6.4 shows that the preparation time changes linearly to the number of attributes in the relation, but still remains very small compared to the complete runtime (the longest preparation for 100 attributes in the relation takes only ≈900 msec. vs. 63 sec of complete runtime).

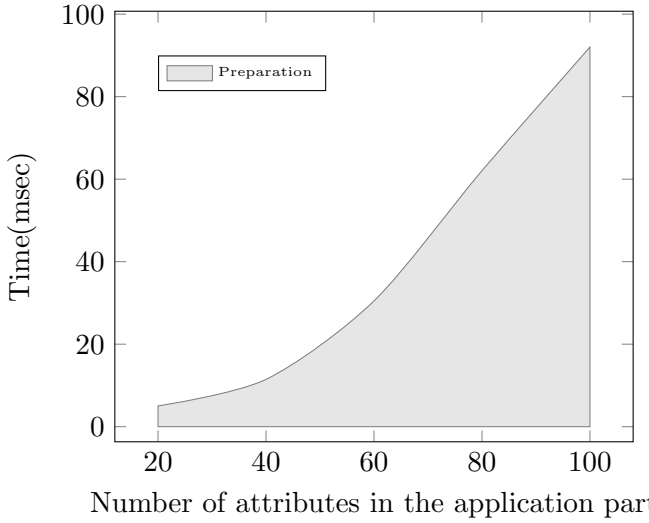

Figure 6.4: Preparation time in the *Internal* experiment on relations with 100'000 tuples (ordered by 1 attribute)

To see how the structure of time consumption changes at different sizes of the Order_By part, I change the ratio of the Order_By part and the size of the descriptive part. The rest of input parameters stay fixed (same as in the *MonetDB vs. R* experiment). Figures 6.5 and 6.6 demonstrate the results of these two manipulations respectively.

As assumed, the resulting difference in the complete runtime can be explained by lightly increased costs of ordering: namely at the stage of order determination and not at the stage of reordering the attributes (this step is always performed for all attributes regardless of the size of the Order_By part).

Further I want to analyse the influence of the application part size. For this purpose I

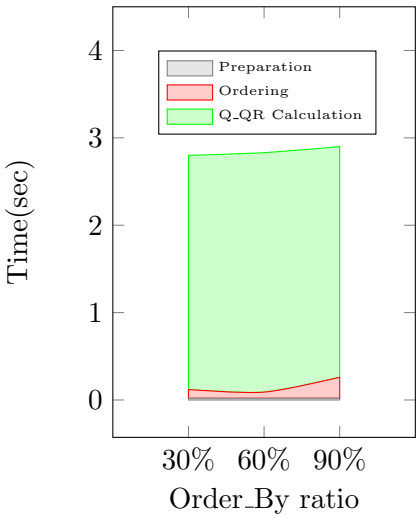

Figure 6.5: Time consumption in the *Internal* experiment on relations with 100'000 tuples, 20 attributes in the application part, 10 in the descriptive part and different Order_By ratios

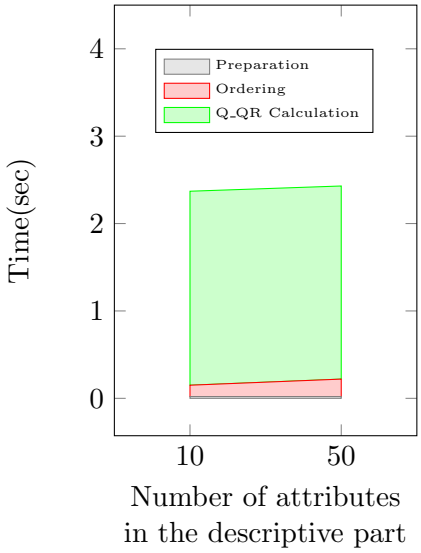

Figure 6.6: Time consumption in the *Internal* experiment on relations with 100'000 tuples, 20 attributes in the application part, 30% Order_By ratio and two different descriptive part sizes

fix the number of Order_By attributes and the number of tuples, and manipulate the size of the application part. The results of this test are presented in Figure 6.7. For the chosen combination of m and n, we can see, that the ordering time changes logarithmically, as assumed during the complexity analysis ($\mathcal{O}(m \log m \cdot n)$). The difference in complete runtime is caused primarily by the increased cost of calculations, which seems to change quadratically to the size of the application part, that corresponds to the calculated complexity of the Gram-Schmidt algorithm ($\mathcal{O}(mn^2)$).

The complexity of the used QRD algorithm correlates linearly to the number of tuples m. To prove this assumption I fix the number of the Order_By attributes and the size of the application part of the relation and change only the amount of tuples. Figure 6.8 illustrates the results of this test and proves that the time spent on calculations indeed changes strictly linearly.

Since the complete runtime grows mostly due to the increasing costs of calculation, I'm interested in investigating what these costs contain. For this purpose I can use the internal function of MonetDB, which allows seeing how much time the execution of each particular MAL-instruction (*trace* function). According to Table 5.1, the calculation of Q_QR function includes 6 different operations takes. They are executed using 5 MAL-instructions (sorted by the total number of executions):

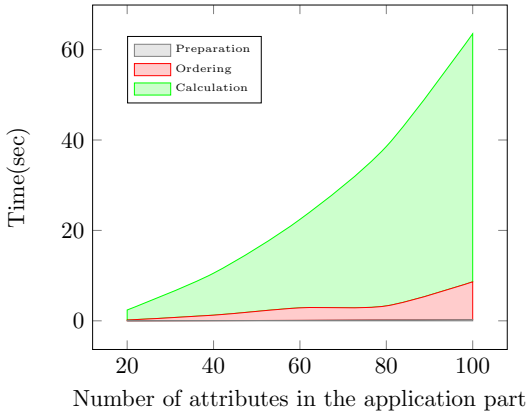

Figure 6.7: Time consumption in the *Internal* experiment on relations with 100'000 tuples and different application part sizes (ordered by 1 attribute)

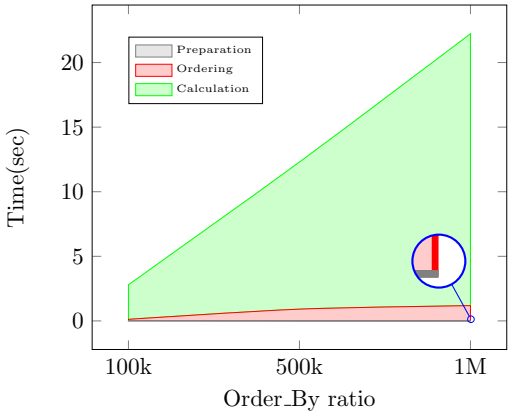

Figure 6.8: Time consumption in the *Internal* experiment on relations with 20 attributes in the application part and different number of tuples (ordered by 1 attribute)

- $*$ (for multiplication and exponentiation, total number of executions : $(n + (n-1)n/2) * m)$
- $/$ (for division, total number of executions : $(n + (n-1)n/2) * m)$
- *sum* (for sum, total number of executions : $(n + (n-1)n/2) * (m-1))$
- $-$ (for subtraction, total number of executions : $((n-1)n/2) * m)$
- *sqrt* (for square root, total number of executions : $n)$

I fix the size of the application and the Order_By parts, as well as the number of tuples in the relation, and do the tests concurrently calling the *trace* function. Summarizing the obtained data per executed MAL-instruction provides an overview, which instructions take the most time during the execution of the Q_QR function. The results of this analysis are shown in Figure 6.9. Comparison between the analysis of the execution number and the results of the measurements shows, that only two MAL-instructions are executed over 90% of the calculation time (these are $*$, $-$ operation and not $*$, $/$ as could be expected based on the total number of executions). The explanation for this can be, that $*$ and $-$ are operations over two BATs. That means that the operations between BATs need more read-write operations. The operations on one BAT or one BAT and one value (such as *sum*, *sqrt*, $/$) are executed relatively fast.

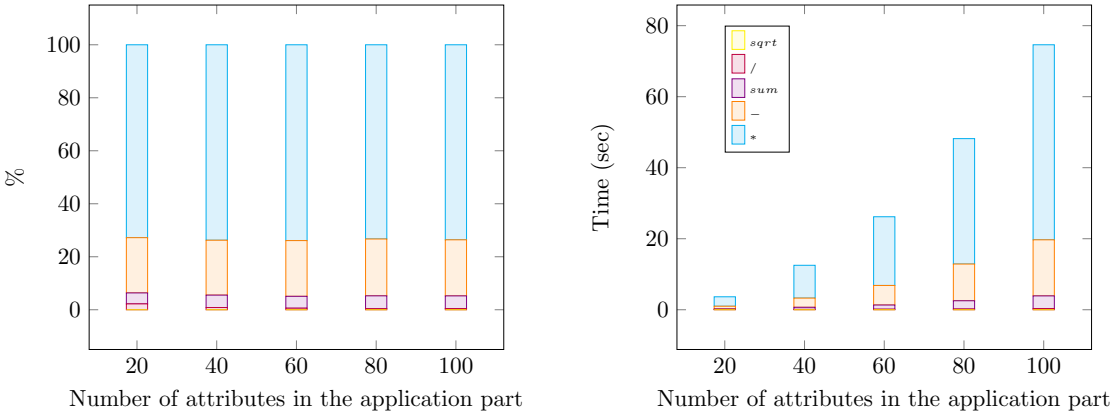

Figure 6.9: Time consumption by the used MAL-instructions, during the Q_QR calculation on 100'000 tuples

R solution

R solution in MonetDB is an embedded but still external package. That means, that in order to use its functionality the information from MonetDB has to be transferred to and from this package, using UDF. To estimate and analyse the time consumption structure when executing the UDF, I manipulate the UDFs to measure the time spent only for transferring the data and Q_QR calculations (see Appendix D). Figure 6.10 shows that the time for data transfer is growing depending not only on the number of tuples in the relation (i.e. the size of input arrays, which need to be passed to R) but also on the size of the application part. The latter increases the transfer time, due to casting of computational results from integer (data type of the input values) to numeric values, which takes double more space on the disk. That means that not only the values in resulting attributes change, but also the amount of data changes after the execution of the QRD. As a result, the transfer costs grow and can take up to over 1/3 of the complete runtime for chosen parameters.

During the experimental evaluation, I noticed a remarkable difference in the usage of cache between MonetDB and the R solution. MonetDB uses the cache and releases the memory as soon as it no longer needs the data in it. R solution, on the other hand, holds the used objects in virtual memory and it seems to apparently retain the data after the completed calculations. That results in the gradual growth of the occupied memory. After a certain number of processed operations no new vectors can be loaded, which leads to a server crash, with the corresponding error message. That represents a memory limit for the whole computation process in R.

There are also limits on individual objects. The storage space cannot exceed the address limit, thus R can only support vectors up to 2^{31}, about $2 * 10^9$ elements, which is also the limit on each dimension of an array [11]. That may lead to problems, that the data is too large to be loaded in memory or the data is loaded, but with remaining

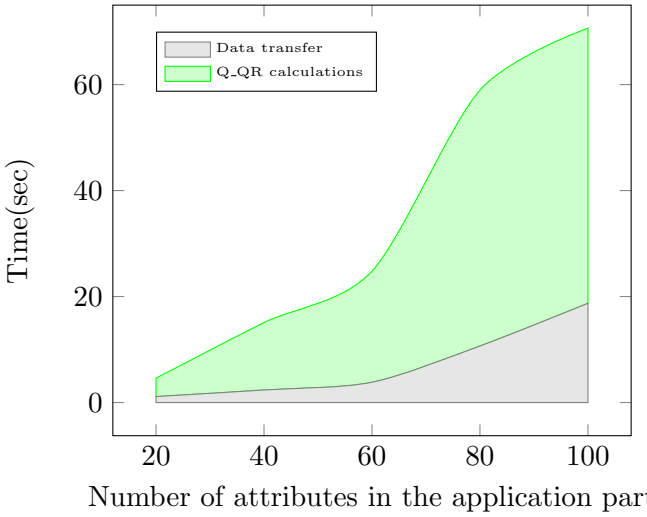

Figure 6.10: Time consumption in R on relations with 1'000'000 tuples (ordered by 1 attribute)

memory no computations can be performed (requires more memory).

Different approaches dealing with these limits are available. When the computational task implies a huge number of data, the server needs to be periodically restarted, or changes in data types and experiment structure can be made. Nevertheless, the unclear mechanism of cleanup in R and the memory limits might be a certain restriction for using this solution.

6.4.2 *Without ordering* mode

The used version of Gram-Schmidt algorithm produces the results by using the operations on vectors. That means, that during the calculation of the dot product or subtraction, the values need to be at the same positions in vectors. The mapping of values in MonetDB, using *OID* in BATs, guarantees, that the values in attributes are multiplied or subtracted only with the values from the same tuple, sharing the same *OID*. That means, that the order in the relation has no influence on the correctness of the Q_QR function. It allows me in the second part of the experimental evaluation to test the implemented solution in *without ordering* mode. The tuples in the output relation have exactly the same values as in the *with ordering* mode, only the order of tuples stay the same as in the input relation.

MonetDB vs. R

The results of test cases for MonetDB and R are displayed in Figure 6.11. The only difference in comparison with the results of the *with ordering* mode is the lack of costs for ordering the values. The complete runtime for performing the function is reduced, but the MonetDB-R performance ratio remains the same. Now it is more obvious, that the complexity of Q_QR computations in R is almost linear, while in MonetDB, with the increasing number of attributes, it tends to quadratic.

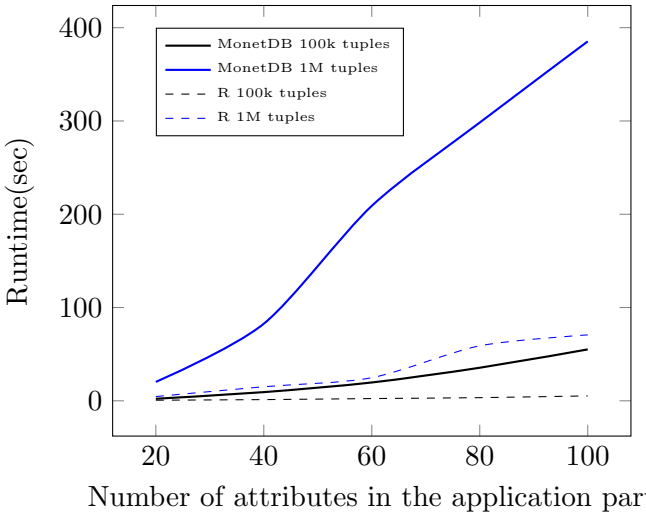

Figure 6.11: Complete runtime in the *MonetDB vs. R* experiment (in *without ordering* mode)

Internal

Also the *Internal* experiment is likely to prove the findings, that Q_QR function spends the most time on calculation of the results. Figure 6.12 illustrates the results of the tests for 100'000 tuples.

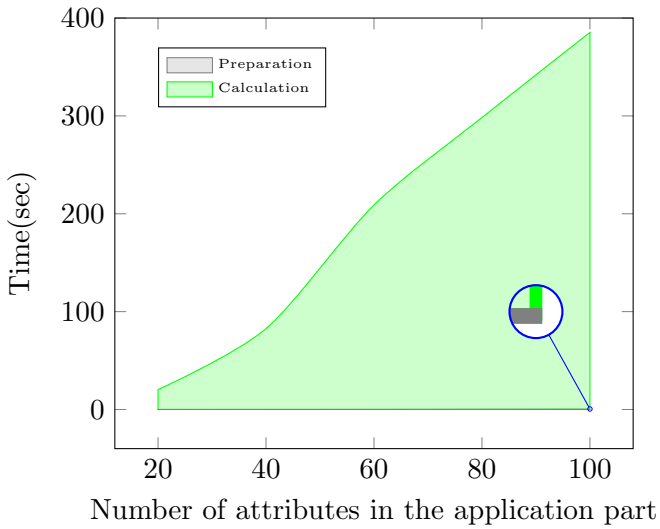

Figure 6.12: Time consumption in the *Internal* experiment on relations with 1'000'000 tuples (in *without ordering* mode)

As Figure 6.13 indicates, the time spent on calculation of the Q_QR function with fixed size of the application part is linearly dependent on the number of tuples in the relation, which means a relatively good performance of chosen algorithm and the column-oriented DBMS, in respect to this parameter.

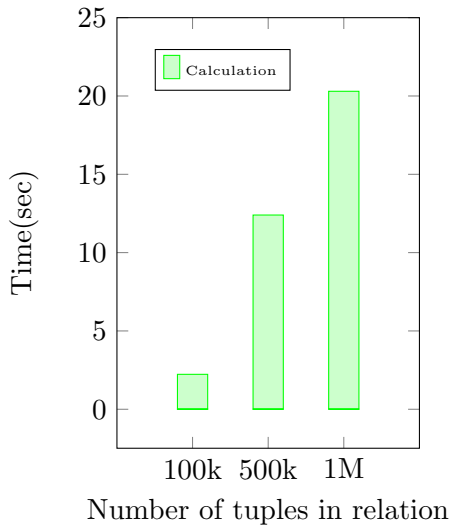

Figure 6.13: Calculation time in the *Internal* experiment on relations with 20 attributes in the application part and different number of tuples in relation (in *without ordering* mode)

6.4.3 Operating environment

For testing the implemented solution I use the virtualization environment VirtualBox. This environment adds an additional level of abstraction between the hardware and the database. This leads to an inevitable performance overhead, since DBMSs are often optimized for the hardware, using the advanced CPU instructions for optimal performance [8].

The number of researches, which handle the topic of performance differences of databases in native and virtualized environments is limited. Nevertheless, they confirm small performance loss (around 10%) in virtualized environments for different states of cache (with and without data) and workloads [7, 4]. There is also one publication available, which found an astonishing performance reduction of MonetDB running in VirtualBox [8].

To prove these findings and if it is the reason for the modest results of the new Q_QR function in MonetDB during the experimental evaluation, I additionally repeat the *MonetDB vs. R* experiment for 100'000 and 1'000'000 tuples, ordered by 1 attribute, with different sizes of the application part under the host operating system on unvirtualized hardware.

Figure 6.14 indicates, that the execution of queries in VirtualBox has a significant overhead, compared to performance under the host. Simple join operations require over twice as much time, while the execution of the Q_QR function shows a tremendous slowdown.

Figure 6.15 demonstrate, that the resulting performance of MonetDB is significantly better than its performance in VirtualBox, while R shows the same results. The per-

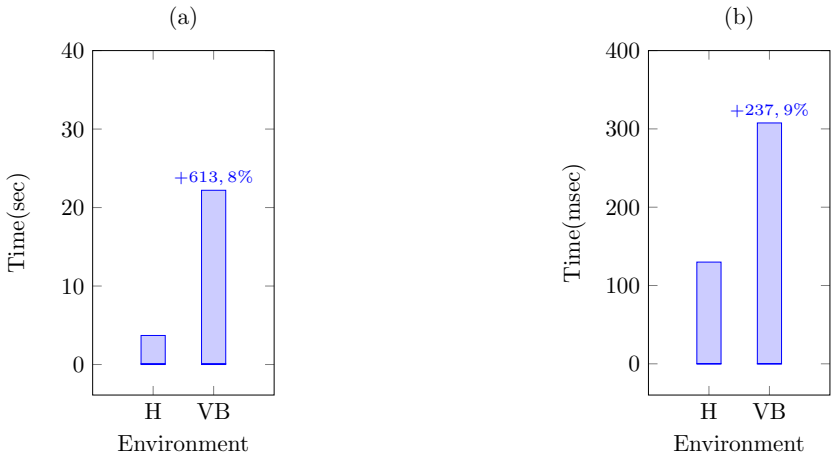

Figure 6.14: Complete runtime of two functions, performed in VirtualBox(VB) and under host operating system (H): (a) Q_QR for 1'000'000 tuples, with 20 attributes in the application part (ordered by 1 attribute); (b) JOIN for two relations with 1000 tuples

formance of the integrated QRD is comparable with the performance of the R solution. For a high number of tuples MonetDB consistently needs almost 3 seconds less than the embedded R.

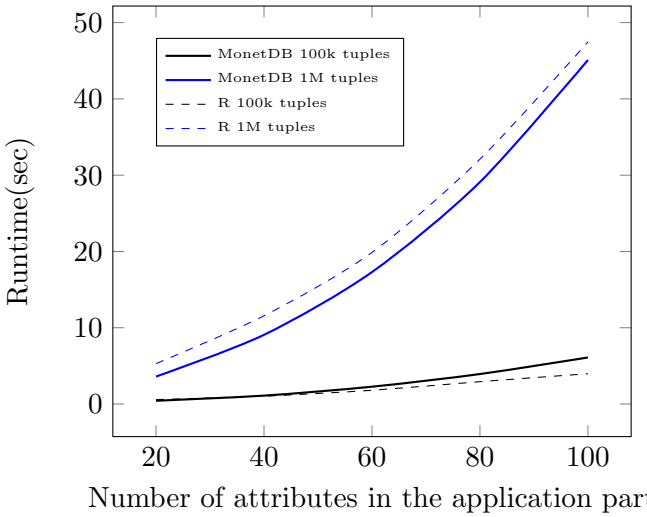

Figure 6.15: Complete runtime in the *MonetDB vs. R* experiment, run under host operating system (ordered by 1 attribute)

Time consumption by executed MAL-instructions has the same structure as in VirtualBox. This means that all operations are performed equally faster under the host operating system, and the execution of latter cannot explain the performance overhead.

A possible reason for this significant performance degradation of MonetDB under virtualized environments can be a huge write load on the memory bus or a high number of system calls during query execution [8].

Another explanation of the performance difference can be execution parallelization inside the DBMS. The number of threads the MonetDB server uses to perform main processing equals the number of available CPU cores in the system [3]. The analysis of the data delivered by the *trace* function points out that MonetDB uses only one thread in VirtualBox. That means no parallelism during query execution in virtualized environment. The used host system allows the server to work with 25 threads, which can be also proved using the results of the *trace* function.

The results of the performed tests show, that MonetDB is environment sensitive and hypervisors (especially VirtualBox) significantly impact the entire performance of MonetDB.

6.4.4 Change of complexity

In the Chapter 5 is assumed, that by some m to n ratios ($m \ggg n$) the costs of ordering prevail over the Q_QR calculation costs, leading to the total complexity of $\mathcal{O}(m \log m \cdot n)$. To prove this assumption I run additional tests on the server, by which I fix the number of tuples on a high level (10'000'000) and manipulate the size of the application part, the results have to be ordered by 1 attribute. Figure 6.16 illustrates the structure of the resulting time consumption. I do not picture the preparation time, since it is very small (< 0.4 *sec*) compared to the total time consumption.

Logarithmic growth of the ordering costs and quadratic growth of the Q_QR calculation costs correspond to the assumed complexities. Using the low number of attributes the ordering of tuples indeed takes more than 50% of the spent time. By increasing the size of the application part the ordering costs still grow logarithmically, while the costs of Q_QR calculations grow quadratically. And from a certain m to n ratio the latter dominate in the time consumption structure. This results in the final complexity of $\mathcal{O}(mn^2)$.

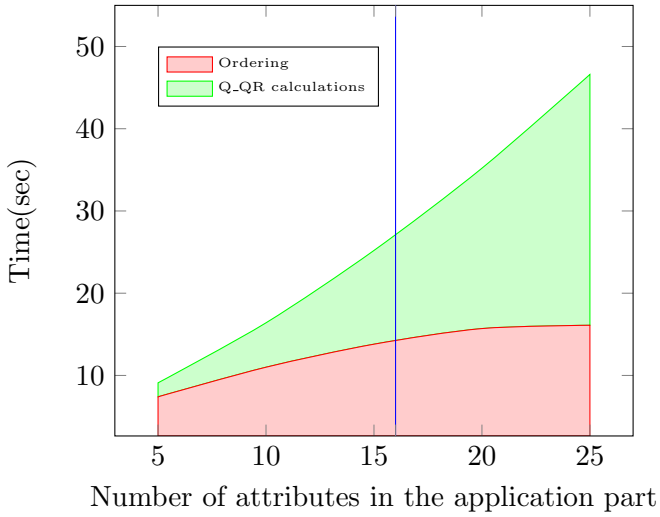

Figure 6.16: Complexity evaluation for 10'000'000 tuples, run under non-virtualized environment (ordered by 1 attribute)

6.5 Optimization

The extensibility framework of MonetDB, allows optimizations on each of the execution stages, which can string out the performance of the entire function:

Preparation: Preparation of the plan is one of core functionalities of MonetDB. As the experimental evaluations show, the operations of this stage are executed already very fast. Therefore, improvements at this step would not have significant effect on the complete runtime. The preparation costs in the R solution are presented by the transfer costs of the input and the output data, which are inevitable and high. By increasing the amount of data transfer costs grow continuously, while the operations in MonetDB run inside the DBMS and no transfer costs arise.

Ordering: The *MonetDB vs. R* and *Internal* experiments show, that high differences in performance of both solutions can be partly explained with the time consumption for ordering the tuples. R uses "radix" sort method, which relies on simple hashing to scale time linearly with the input size. The asymptotic time complexity is $\mathcal{O}(m)$. For small inputs ($m < 200$), the implementation uses an insertion sort ($\mathcal{O}(m^2)$). For integer vectors of range less than 100,000, it switches to a simpler and faster linear time counting sort. In all cases, the sort is stable. The "radix" method generally outperforms other methods, especially for character vectors and small integers [12]. Empirically obtained results show, that the complexity of the existing ordering algorithm in MonetDB is logarithmic, which seems to be the best attainable. But the absolute time consumption for ordering is highly influenced by the environment, where MonetDB is running.

Q_QR calculations: The most time is consumed by the execution of the QRD. The advantage of Gram-Schmid algorithm is, that it can be executed in parallel: for example at the orthogonalization step. That represents an option for further optimization of the Q_QR function. But it is only possible, if the system allows multiple threads during the execution, which seems to be a problem in virtualized environments.

Multithreading in virtualized environments can significantly speed up not only the execution of the Q_QR function but also the performance of the entire DBMS. To achieve it, it is necessary to make the parallelizing of execution possible, that in practice means to resolve the dependency on the amount of CPU cores available.

To avoid the existing performance degradation in virtualized environments the user can move to unvirtualized hardware. If it is not possible, there is an another approach to speed up the performance. Working in a virtualized environment, MonetDB maps the database files on the VirtualBox disk image. Moving these files to an in-memory file system (/dev/shm) helps to massively improve the performance [8].

With these optimisations and taking into account the transfer costs and memory restrictions in R, MonetDB is an attractive solution for further statistical and linear algebra calculations.

7

Summary and future work

The goal of this thesis was to integrate the QR decomposition into column-oriented DBMS. First, I researched related works, showing which QRD algorithm is most suitable for the task and chose one of them. In the next step, I analysed how linear algebra and statistical operations are used in existing DBMS, and which approach is common for carrying out statistical and linear operations.

Then the implementation approach was introduced, including detailed description of the task, introducing the architecture and the main components of MonetDB. Implementation is described using a running example. Experimental evaluation provides information for judging the complexity of the implementation and comparing it with an existing R solution.

As a result, MonetDB provides a good basis for customising DBMS's functionality. Existing libraries and primitive BAT operations allow to implement practically every advanced statistical or linear algebra operation. The existing optimization logic at all levels of MonetDB architecture makes it fairly easy to extend the rules for the newly integrated operations.

I also analysed the performance of MonetDB, depending on the environment used. The virtualized environment (especially VirtualBox) significantly impacts the performance not only of the implemented Q_QR function but of the entire performance of MonetDB.

Big input and resulting relations require a lot of main memory, which might be a problem on machines with a small main memory. Also computing a lot of concurrent queries calling the function on the same server leads to memory problems. However, as values of one attribute in MonetDB are stored so "densely" (together in the same C-array), they do not require much place. Together with optimized CPU instructions and different compression techniques at many levels it provides an efficient mechanism to use the advantage of fast cache memory in full.

Running the implemented Q_QR function in a non-virtualized environment shows a performance comparable with the performance of math and analytic software. The possibility of calculations directly on relations also saves time and resources for exporting and reimporting the data. All that makes MonetDB attractive for statistical or linear algebra operations.

Further linear algebra operations can be also integrated into MonetDB: e.g. implementation of a function which returns R matrix from QRD etc.

An interesting topic for further research can be investigation the exact reasons of the performance degradation in VirtualBox and solving this performance overhead. That would allow using MonetDB on different platforms, without the loss of performance.

References

[1] Abadi, D. J., Boncz, P. A., and Harizopoulos, S. Column-oriented database systems. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1664–1665.

[2] Gander, W. Algorithms for the qr decomposition. *Res. Rep 80*, 02 (1980), 1251–1268.

[3] Group, D. A. Monetdb. *http://www.monetdb.org/*, May 2017.

[4] Grund, M., Schaffner, J., Krueger, J., Brunnert, J., and Zeier, A. The effects of virtualization on main memory systems. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (2010), ACM, pp. 41–46.

[5] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, S., Kersten, M., et al. Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering 35*, 1 (2012), 40–45.

[6] Kersten, M., Zhang, Y., Ivanova, M., and Nes, N. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases* (2011), ACM, pp. 1–12.

[7] Minhas, U. F., Yadav, J., Aboulnaga, A., and Salem, K. Database systems on virtual machines: How much do you lose? In *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on* (2008), IEEE, pp. 35–41.

[8] Mühlbauer, T., Rödiger, W., Kipf, A., Kemper, A., and Neumann, T. High-performance main-memory database systems and modern virtualization: Friends or foes? In *Proceedings of the Fourth Workshop on Data analytics in the Cloud* (2015), ACM, p. 4.

[9] O'Leary, D. P., and Whitman, P. Parallel qr factorization by householder and modified gram-schmidt algorithms. *Parallel computing 16*, 1 (1990), 99–112.

[10] Oracle. Oracle r enterprise. *http://www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html*, June 2017.

[11] R. Memory limits in r. *https://stat.ethz.ch/R-manual/R-devel/library/base/html/Memory-limits.html*, June 2017.

[12] R. R: Sorting or ordering vectors. *https://stat.ethz.ch/R-manual/R-devel/library/base/html/sort.html*, June 2017.

[13] Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Balazinska, M., DeWitt, D., Heath, B., Maier, D., Madden, S., Patel, J., et al. Overview of scidb. In *2010 International Conference on Management of Data, SIGMOD'10* (2010).

[14] Stonebraker, M., Brown, P., Poliakov, A., and Raman, S. The architecture of scidb. In *Scientific and Statistical Database Management* (2011), Springer, pp. 1–16.

[15] Zhang, Y., Kersten, M., Ivanova, M., and Nes, N. Sciql: bridging the gap between science and relational dbms. In *Proceedings of the 15th Symposium on International Database Engineering & Applications* (2011), ACM, pp. 124–133.

[16] Zhang, Y., Kersten, M., and Manegold, S. Sciql: array data processing inside an rdbms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1049–1052.

A

Code

A.1 sql_parser.y

```
...
table_ref:
 simple_table
 subquery table_name     { $$ = $1;
                           if ($$->token == SQL_SELECT) {
                              SelectNode *sn = (SelectNode
                              *)$1;
                                   sn->name = $2;
                           } else {append_symbol($1->data.
                              lval, $2);}
}
 QQR '(' table_ref ON column_ref_commalist opt_order_by_clause
    ')'
        { dlist *l = L();
         append_symbol(l, $3);
         append_list(l, $5);
         append_symbol(l, $6);
         $$ = _symbol_create_list( SQL_QQR, l); }
 QQR '(' qqr_subquery ON column_ref_commalist
    opt_order_by_clause ')'
        { dlist *l = L();
         append_symbol(l, $3);
         append_list(l, $5);
         append_symbol(l, $6);
         $$ = _symbol_create_list( SQL_QQR, l); }
 .
qqr_subquery:
        subquery table_name     {       $$ = $1;
                                        if ($$->token ==
                                           SQL_SELECT) {
```

```
                                                SelectNode *sn = (
                                                SelectNode*)$1;
                                                    sn->name = $2;
                                            } else { append_symbol(
                                                $1->data.lval, $2);}
                                    }
| subquery      { $$ = NULL;
                 yyerror(m, "Subquery table reference needs
                    alias, use (SELECT . ) AS xxx and ");
                 YYABORT;}

...
```

A.2 rel_bin.c

...

```
static stmt *
rel2bin_qqr(mvc *sql, sql_rel *rel, list *refs)
{
        clock_t start = clock();

        list *lAppUnsorted;
        list *lDescr;
        list *lAppSorted;
        list *qqrFinal;
        list *pl;
        node *n;
        node *en;
        node *exp;
        node *k;
        node *l;
        node *j;
        stmt *left = NULL;
        stmt *onV = NULL;
        stmt *psub = NULL;
        stmt *orderby_ids = NULL;
        stmt *orderby_grp = NULL;

        lAppUnsorted = sa_list(sql->sa);
        lDescr = sa_list(sql->sa);
        lAppSorted = sa_list(sql->sa);
        qqrFinal = sa_list(sql->sa);

        //Take the table
        if (rel->l)
                left = subrel_bin(sql, rel->l, refs);
        if (!left)
                return NULL;
        printf("\n***Table %s***\n", table_name(sql->sa, left->
           op4.lval->h->data));

        //Make 2 lists for descriptive part and for application
            part
        if (rel->exps){

                for (n = left->op4.lval->h; n; n = n->next) {
```

```
                        stmt *sc = n->data;
                        char *cname = column_name(sql->sa, sc);
                        char *tname = table_name(sql->sa, sc);
                        int is_same = 0;

                        for (exp = rel->exps->h; exp ; exp =
                           exp->next){
                                sql_exp *EXPsc = exp->data;
                                char *EXPtname = EXPsc->l;
                                char *EXPcname = EXPsc->r;

                                if (strcmp(cname,"%TID%") == 0)
                                   {
                                        is_same = 0;
                                        break;}
                                if (strcmp(cname,EXPcname)== 0)
                                   {
                                        is_same = 1;
                                        break;}
                        }

                        stmt *nc;
                        nc = column(sql->sa, sc);
                        if (is_same == 1)
                                list_append(lAppUnsorted,
                                   stmt_alias(sql->sa, nc,
                                   tname, cname));
                        else
                                list_append(lDescr, stmt_alias(
                                   sql->sa, nc, tname, cname));
                }
        }

        //Find ORDER
        if (rel->r) {
                list *oexps = rel->r;

                for (en = oexps->h; en; en = en->next) {
                        stmt *orderby = NULL;
                        sql_exp *orderbycole = en->data;
                        stmt *orderbycolstmt = exp_bin(sql,
                           orderbycole, left, psub, NULL, NULL,
                            NULL, NULL);
```

```
                    if (!orderbycolstmt) {
                            assert(0);
                            return NULL;
                    }
                    /* single values don't need sorting */
                    if (orderbycolstmt->nrcols == 0) {
                            orderby_ids = NULL;
                            break;
                    }
                    if (orderby_ids)
                            orderby = stmt_reorder(sql->sa,
                                orderbycolstmt,
                                is_ascending(orderbycole),
                                orderby_ids, orderby_grp);
                    else
                            orderby = stmt_order(sql->sa,
                                orderbycolstmt, is_ascending
                                (orderbycole));

                    orderby_ids = stmt_result(sql->sa,
                        orderby, 1);
                    orderby_grp = stmt_result(sql->sa,
                        orderby, 2);
            }
    }

    //ORDER
    if (orderby_ids){

            for(n = lDescr->h; n; n = n->next){
                    stmt *sc = n->data;
                    char *cname = column_name(sql->sa, sc);
                    char *tname = table_name(sql->sa, sc);

                    sc = stmt_project(sql->sa, orderby_ids,
                        sc);
                    sc = stmt_alias(sql->sa, sc, tname,
                        cname );
                    list_append(qqrFinal, sc);
            }

            for(l = lAppUnsorted->h; l; l = l->next){
                    stmt *sc = l->data;
                    char *cname = column_name(sql->sa, sc);
```

```
                        char *tname = table_name(sql->sa, sc);

                        sc = stmt_project(sql->sa, orderby_ids,
                             sc);
                        sc = stmt_alias(sql->sa, sc, tname,
                           cname);
                        list_append(lAppSorted, sc);
                }
        }

        // Make QQR for the sorted Application part and append
           it to the result list qqrFinal
        for(k = lAppSorted->h; k; k = k->next){
                stmt *original;
                stmt *norm;
                original = k->data;
                norm = stmt_norm(sql->sa, original);
                list_append(qqrFinal, norm);
                k->data = norm;

                for(j = k->next; j; j = j->next){
                        stmt *original = j->data;
                        stmt *orth = stmt_orth(sql->sa,
                           original, norm);
                        j->data = orth;
                }
        }
        return stmt_list(sql->sa, qqrFinal);
}

...
```

A.3 sql_gencode.c

...

```
static int
_dumpstmt(backend *sql, MalBlkPtr mb, stmt *s)
{
        InstrPtr q = NULL;
        node *n;
        if (s) {
                if (s->nr > 0)
                        return s->nr;   /* stmt already handled
                            */

                switch (s->type) {
                case st_norm:{
                                InstrPtr sum, v, p;
                                int l, res;

                                l = _dumpstmt(sql, mb, s->op1);
                                assert(l >= 0 );

                                q = newStmt(mb, batcalcRef,
                                    "*");
                                q = pushArgument(mb, q, l);
                                q = pushArgument(mb, q, l);
                                res = getDestVar(q);

                                sum = newStmt(mb, aggrRef, "sum
                                    ");
                                sum = pushArgument(mb, sum, res
                                    );
                                res = getDestVar(sum);

                                v = newStmt(mb, mmathRef, "sqrt
                                    ");
                                v = pushArgument(mb, v, res);
                                res = getDestVar(v);

                                p = newStmt(mb, batcalcRef,
                                    "/");
                                p = pushArgument(mb, p, l);
                                p = pushArgument(mb, p, res);
                                res = getDestVar(p);
```

```
                        s->nr = getDestVar(p);
                return s->nr;
            } break;
            case st_orth:{
                        InstrPtr sum, p, m;
                        int l, r, res;

                        l = _dumpstmt(sql, mb, s->op1);
                        r = _dumpstmt(sql, mb, s->op2);
                        assert(l >= 0 && r >= 0);

                        q = newStmt(mb, batcalcRef,
                            "*");
                        q = pushArgument(mb, q, l);
                        q = pushArgument(mb, q, r);
                        res = getDestVar(q);

                        sum = newStmt(mb, aggrRef, "sum
                            ");
                        sum = pushArgument(mb, sum, res
                            );
                        res = getDestVar(sum);

                        p = newStmt(mb, batcalcRef,
                            "*");
                        p = pushArgument(mb, p, res);
                        p = pushArgument(mb, p, r);
                        res = getDestVar(p);

                        m = newStmt(mb, batcalcRef,
                            "-");
                        m = pushArgument(mb, m, l);
                        m = pushArgument(mb, m, res);

                        s->nr = getDestVar(m);

                        return s->nr;
            } break;
}
...
```

B

Relation-Plan

```
plan SELECT * FROM q_qr (table1 ON x,y,z ORDER BY a);
+=====================================================+
rel
+=====================================================+

project (
qqr (
 table(sys.table1) [ table1.x, table1.y, table1.z, table1.a,
    table1.b, table1.c, table1.%TID% NOT NULL ]
) [ table1.x, table1.y, table1.z ]
) [ table1.x, table1.y, table1.z, table1.a, table1.b, table1.c
   ]
+=====================================================+
```

C

MAL-Plan

C.1 MAL-Plan for *with ordering* mode

```
explain SELECT * FROM q_qr (table1 ON x,y,z ORDER BY a);
+=====================================================+
mal
+=====================================================+
function user.s5_1():void;
X_121:void := querylog.define("explain select * from q_qr (
   table1 on x,y,z order by a);","default_pipe",97);
X_80 := bat.new(nil:oid,nil:str);
X_88 := bat.append(X_80,"sys.table1");
X_96 := bat.append(X_88,"sys.table1");
X_101 := bat.append(X_96,"sys.table1");
X_106 := bat.append(X_101,"sys.table1");
X_111 := bat.append(X_106,"sys.table1");
X_116 := bat.append(X_111,"sys.table1");
X_83 := bat.new(nil:oid,nil:str);
X_90 := bat.append(X_83,"x");
X_97 := bat.append(X_90,"y");
X_102 := bat.append(X_97,"z");
X_107 := bat.append(X_102,"a");
X_112 := bat.append(X_107,"b");
X_117 := bat.append(X_112,"c");
X_84 := bat.new(nil:oid,nil:str);
X_91 := bat.append(X_84,"int");
X_98 := bat.append(X_91,"int");
X_103 := bat.append(X_98,"int");
X_108 := bat.append(X_103,"int");
X_113 := bat.append(X_108,"int");
X_118 := bat.append(X_113,"int");
X_85 := bat.new(nil:oid,nil:int);
X_93 := bat.append(X_85,32);
```

```
X_99 := bat.append(X_93,32);
X_104 := bat.append(X_99,32);
X_109 := bat.append(X_104,32);
X_114 := bat.append(X_109,32);
X_119 := bat.append(X_114,32);
X_87 := bat.new(nil:oid,nil:int);
X_95 := bat.append(X_87,0);
X_100 := bat.append(X_95,0);
X_105 := bat.append(X_100,0);
X_110 := bat.append(X_105,0);
X_115 := bat.append(X_110,0);
X_120 := bat.append(X_115,0);
X_1 := sql.mvc();
C_2:bat[:oid] := sql.tid(X_1,"sys","table1");
X_5:bat[:int] := sql.bind(X_1,"sys","table1","a",0);
(C_8,r1_8) := sql.bind(X_1,"sys","table1","a",2);
X_11:bat[:int] := sql.bind(X_1,"sys","table1","a",1);
X_13 := sql.delta(X_5,C_8,r1_8,X_11);
X_14 := algebra.projection(C_2,X_13);
(X_15,r1_15,r2_15) := algebra.subsort(X_14,false,false);
X_19:bat[:int] := sql.bind(X_1,"sys","table1","x",0);
(C_21,r1_22) := sql.bind(X_1,"sys","table1","x",2);
X_23:bat[:int] := sql.bind(X_1,"sys","table1","x",1);
X_24 := sql.delta(X_19,C_21,r1_22,X_23);
X_25:bat[:int] := algebra.projectionpath(r1_15,C_2,X_24);
X_26 := batcalc.*(X_25,X_25);
X_27 := aggr.sum(X_26);
X_28 := mmath.sqrt(X_27);
X_29 := batcalc./(X_25,X_28);
X_30:bat[:int] := sql.bind(X_1,"sys","table1","y",0);
(C_32,r1_34) := sql.bind(X_1,"sys","table1","y",2);
X_34:bat[:int] := sql.bind(X_1,"sys","table1","y",1);
X_35 := sql.delta(X_30,C_32,r1_34,X_34);
X_36:bat[:int] := algebra.projectionpath(r1_15,C_2,X_35);
X_37 := batcalc.*(X_36,X_29);
X_38 := aggr.sum(X_37);
X_39 := batcalc.*(X_38,X_29);
X_40 := batcalc.-(X_36,X_39);
X_41 := batcalc.*(X_40,X_40);
X_42 := aggr.sum(X_41);
X_43 := mmath.sqrt(X_42);
X_44 := batcalc./(X_40,X_43);
X_45:bat[:int] := sql.bind(X_1,"sys","table1","z",0);
(C_47,r1_50) := sql.bind(X_1,"sys","table1","z",2);
```

```
X_49:bat[:int] := sql.bind(X_1,"sys","table1","z",1);
X_50 := sql.delta(X_45,C_47,r1_50,X_49);
X_51:bat[:int] := algebra.projectionpath(r1_15,C_2,X_50);
X_52 := batcalc.*(X_51,X_29);
X_53 := aggr.sum(X_52);
X_54 := batcalc.*(X_53,X_29);
X_55 := batcalc.-(X_51,X_54);
X_56 := batcalc.*(X_55,X_44);
X_57 := aggr.sum(X_56);
X_58 := batcalc.*(X_57,X_44);
X_59 := batcalc.-(X_55,X_58);
X_60 := batcalc.*(X_59,X_59);
X_61 := aggr.sum(X_60);
X_62 := mmath.sqrt(X_61);
X_63 := batcalc./(X_59,X_62);
X_64 := algebra.projection(r1_15,X_14);
X_65:bat[:int] := sql.bind(X_1,"sys","table1","b",0);
(C_67,r1_71) := sql.bind(X_1,"sys","table1","b",2);
X_69:bat[:int] := sql.bind(X_1,"sys","table1","b",1);
X_70 := sql.delta(X_65,C_67,r1_71,X_69);
X_71:bat[:int] := algebra.projectionpath(r1_15,C_2,X_70);
X_72:bat[:int] := sql.bind(X_1,"sys","table1","c",0);
(C_74,r1_79) := sql.bind(X_1,"sys","table1","c",2);
X_76:bat[:int] := sql.bind(X_1,"sys","table1","c",1);
X_77 := sql.delta(X_72,C_74,r1_79,X_76);
X_78:bat[:int] := algebra.projectionpath(r1_15,C_2,X_77);
sql.resultSet(X_116,X_117,X_118,X_119,X_120,X_29,X_44,X_63,X_64
    ,X_71,X_7 : 8); end user.s5_1
```

C.2 MAL-Plan for *without ordering* mode

```
explain SELECT * FROM q_qr (table1 ON x,y,z ORDER BY a);
+=====================================================================+
mal
+=====================================================================+
function user.s2_1():void;
X_116:void := querylog.define("select * from q_qr (table1 on x,
   y,z order by a);","default_pipe",95);
X_75 := bat.new(nil:oid,nil:str);
X_83 := bat.append(X_75,"sys.table1");
X_91 := bat.append(X_83,"sys.table1");
X_96 := bat.append(X_91,"sys.table1");
X_101 := bat.append(X_96,"sys.table1");
X_106 := bat.append(X_101,"sys.table1");
X_111 := bat.append(X_106,"sys.table1");
X_78 := bat.new(nil:oid,nil:str);
X_85 := bat.append(X_78,"x");
X_92 := bat.append(X_85,"y");
X_97 := bat.append(X_92,"z");
X_102 := bat.append(X_97,"a");
X_107 := bat.append(X_102,"b");
X_112 := bat.append(X_107,"c");
X_79 := bat.new(nil:oid,nil:str);
X_86 := bat.append(X_79,"int");
X_93 := bat.append(X_86,"int");
X_98 := bat.append(X_93,"int");
X_103 := bat.append(X_98,"int");
X_108 := bat.append(X_103,"int");
X_113 := bat.append(X_108,"int");
X_80 := bat.new(nil:oid,nil:int);
X_88 := bat.append(X_80,32);
X_94 := bat.append(X_88,32);
X_99 := bat.append(X_94,32);
X_104 := bat.append(X_99,32);
X_109 := bat.append(X_104,32);
X_114 := bat.append(X_109,32);
X_82 := bat.new(nil:oid,nil:int);
X_90 := bat.append(X_82,0);
X_95 := bat.append(X_90,0);
X_100 := bat.append(X_95,0);
X_105 := bat.append(X_100,0);
X_110 := bat.append(X_105,0);
```

```
X_115 := bat.append(X_110,0);
X_1 := sql.mvc();
C_2:bat[:oid] := sql.tid(X_1,"sys","table1");
X_5:bat[:int] := sql.bind(X_1,"sys","table1","x",0);
(C_8,r1_8) := sql.bind(X_1,"sys","table1","x",2);
X_11:bat[:int] := sql.bind(X_1,"sys","table1","x",1);
X_13 := sql.delta(X_5,C_8,r1_8,X_11);
X_14 := algebra.projection(C_2,X_13);
X_15 := batcalc.*(X_14,X_14);
X_16 := aggr.sum(X_15);
X_17 := mmath.sqrt(X_16);
X_18 := batcalc./(X_14,X_17);
X_19:bat[:int] := sql.bind(X_1,"sys","table1","y",0);
(C_21,r1_21) := sql.bind(X_1,"sys","table1","y",2);
X_23:bat[:int] := sql.bind(X_1,"sys","table1","y",1);
X_24 := sql.delta(X_19,C_21,r1_21,X_23);
X_25 := algebra.projection(C_2,X_24);
X_26 := batcalc.*(X_25,X_18);
X_27 := aggr.sum(X_26);
X_28 := batcalc.*(X_27,X_18);
X_29 := batcalc.-(X_25,X_28);
X_30 := batcalc.*(X_29,X_29);
X_31 := aggr.sum(X_30);
X_32 := mmath.sqrt(X_31);
X_33 := batcalc./(X_29,X_32);
X_34:bat[:int] := sql.bind(X_1,"sys","table1","z",0);
(C_36,r1_36) := sql.bind(X_1,"sys","table1","z",2);
X_38:bat[:int] := sql.bind(X_1,"sys","table1","z",1);
X_39 := sql.delta(X_34,C_36,r1_36,X_38);
X_40 := algebra.projection(C_2,X_39);
X_41 := batcalc.*(X_40,X_18);
X_42 := aggr.sum(X_41);
X_43 := batcalc.*(X_42,X_18);
X_44 := batcalc.-(X_40,X_43);
X_45 := batcalc.*(X_44,X_33);
X_46 := aggr.sum(X_45);
X_47 := batcalc.*(X_46,X_33);
X_48 := batcalc.-(X_44,X_47);
X_49 := batcalc.*(X_48,X_48);
X_50 := aggr.sum(X_49);
X_51 := mmath.sqrt(X_50);
X_52 := batcalc./(X_48,X_51);
X_53:bat[:int] := sql.bind(X_1,"sys","table1","a",0);
(C_55,r1_55) := sql.bind(X_1,"sys","table1","a",2);
```

```
X_57:bat[:int] := sql.bind(X_1,"sys","table1","a",1);
X_58 := sql.delta(X_53,C_55,r1_55,X_57);
X_59 := algebra.projection(C_2,X_58);
X_60:bat[:int] := sql.bind(X_1,"sys","table1","b",0);
(C_62,r1_62) := sql.bind(X_1,"sys","table1","b",2);
X_64:bat[:int] := sql.bind(X_1,"sys","table1","b",1);
X_65 := sql.delta(X_60,C_62,r1_62,X_64);
X_66 := algebra.projection(C_2,X_65);
X_67:bat[:int] := sql.bind(X_1,"sys","table1","c",0);
(C_69,r1_69) := sql.bind(X_1,"sys","table1","c",2);
X_71:bat[:int] := sql.bind(X_1,"sys","table1","c",1);
X_72 := sql.delta(X_67,C_69,r1_69,X_71);
X_73 := algebra.projection(C_2,X_72);
sql.resultSet(X_111,X_112,X_113,X_114,X_115,X_18,X_33,X_52,X_59
    ,X_66,X_73);
end user.s2_1;
```

D

UDF

D.1 UDF for *with ordering* mode

```
CREATE FUNCTION function_R (a1 INTEGER, a2 INTEGER, a3 INTEGER, a4
    INTEGER, a5 INTEGER, a6 INTEGER)
RETURNS TABLE (x1 INTEGER, x2 DOUBLE, x3 DOUBLE, x4 DOUBLE, x5
    DOUBLE, x6 DOUBLE) LANGUAGE R {
frame <- data.frame(x1=a1, x2=a2, x3=a3, x4=a4, x5=a5, x6=a6);
frameSorted <- frame[with(frame, order(x1)),];
frameforQR<-frameSorted[,c(2,3,4,5,6)];
mat <- as.matrix(frameforQR);
Q <- qr.Q(qr(mat));
cbind(frameSorted[,c(1)], as.data.frame(Q));};
```

D.2 UDF for *without ordering* mode

```
CREATE FUNCTION function_R (a1 INTEGER, a2 INTEGER, a3 INTEGER, a4
    INTEGER, a5 INTEGER, a6 INTEGER)
RETURNS TABLE (x1 INTEGER, x2 INTEGER, x3 INTEGER, x4 DOUBLE, x5
    DOUBLE, x6 DOUBLE) LANGUAGE R {
frame <- data.frame(x1=a1, x2=a2, x3=a3, x4=a4, x5=a5, x6=a6);
frameforQR<-frame[,c(4,5,6)];
mat <- as.matrix(frameforQR);
Q <- qr.Q(qr(mat));
cbind(frame[,c(1,2,3)], as.data.frame(Q));};
```

D.3 UDF without Q_QR

```
CREATE FUNCTION function_R (a1 INTEGER, a2 INTEGER, a3 INTEGER, a4
    INTEGER, a5 INTEGER, a6 INTEGER)
RETURNS TABLE (x1 INTEGER, x2 DOUBLE, x3 DOUBLE, x4 DOUBLE, x5
    DOUBLE, x6 DOUBLE) LANGUAGE R {
frame <- data.frame(x1=a1, x2=a2, x3=a3, x4=a4, x5=a5, x6=a6);
frameforQR<-frame[,c(2,3,4,5,6)];
mat <- as.matrix(sapply(frameforQR, as.numeric));
cbind(frame[,c(1)], as.data.frame(mat));};
```

E

Detailed test results

Table E.1: Complete runtime in *with ordering* mode, ordered by one attribute (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	2,30	9,50	20,20	45,00	80,00
	2	2,40	10,90	25,00	39,30	54,70
	3	2,50	11,20	22,10	31,40	55,80
	Mean	2,40	10,53	22,43	38,57	63,50
1M	1	21,90	88,00	265,00	480,00	826,00
	2	22,10	85,00	270,00	444,00	846,00
	3	22,10	85,00	270,00	444,00	846,00
	Mean	22,30	85,00	271,00	459,00	842,00

R

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	1,80	2,30	4,10	5,10	8,90
	2	0,86	1,80	2,90	5,00	5,40
	3	0,81	2,10	3,20	3,80	5,90
	Mean	1,16	2,07	3,40	4,63	6,73
1M	1	7,30	18,40	44,10	84,00	133,00
	2	6,20	15,40	41,20	690,00	153,00
	3	6,00	18,40	41,30	96,00	159,00
	Mean	6,50	17,40	42,20	90,00	148,33

Table E.2: Complete runtime in *with ordering* mode on relations with 10 attributes in the descriptive and 20 attributes in the application part (in sec.)

MonetDB

Number of tuples	Run	Order_By ratio		
		30	60	90
10k	1	2,20	2,40	2,20
	2	2,50	2,30	2,20
	3	2,40	2,20	2,20
	Mean	2,37	2,30	2,20
1M	1	22,10	23,60	24,80
	2	22,20	24,50	25,90
	3	22,40	25,30	24,90
	Mean	22,23	24,47	25,20

R

Number of tuples	Run	Order_By ratio		
		30	60	90
100k	1	0,81	0,75	0,82
	2	0,80	0,71	0,77
	3	0,76	0,70	0,84
	Mean	0,79	0,72	0,81
1M	1	9,20	17,00	14,80
	2	8,50	13,90	20,10
	3	7,40	20,10	18,00
	Mean	8,37	17,00	17,63

Table E.3: Complete runtime in *with ordering* mode, on relations with 30% Order_By ratio and 20 attributes in the application part (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the descriptive part	
		10	50
100k	1	2,20	2,40
	2	2,50	2,30
	3	2,40	2,60
	Mean	2,37	2,43
1M	1	22,10	30,70
	2	22,20	29,20
	3	22,40	32,70
	Mean	22,23	30,87

R

Number of tuples	Run	Number of attributes in the descriptive part	
		10	50
100k	1	0,81	1,10
	2	0,80	0,90
	3	0,76 0,95	
	Mean	0,79	0,98
1M	1	9,20	20,40
	2	8,50	28,30
	3	7,40	21,80
	Mean	8,37	23,50

Table E.4: Preparation time in *with ordering* mode, ordered by one attribute (in msec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	13,68	51,00	78,50	213,00	288,28
	2	14,63	48,40	89,10	143,41	245,51
	3	16,20	40,50	129,60	240,37	231,84
	Mean	14,83	46,63	99,07	198,93	255,21

Table E.5: Complete runtime in *without ordering* mode (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	2,30	9,50	20,50	35,50	55,00
	2	2,20	9,20	18,80	35,50	53,40
	3	2,20	9,30	19,70	35,40	57,00
	Mean	2,23	9,33	19,67	35,47	55,13
1M	1	21,10	82,00	224,00	302,00	373,00
	2	19,70	82,00	202,00	298,00	371,00
	3	20,10	84,00	201,00	295,00	412,00
	Mean	20,30	82,67	209,00	298,33	385,33

R

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	1,00	1,30	2,50	3,50	5,70
	2	0,51	1,30	2,90	3,30	5,20
	3	0,53	1,30	1,90	3,40	5,00
	Mean	0,68	1,30	2,43	3,40	5,30
1M	1	4,70	15,30	27,40	58,80	67,00
	2	5,30	14,50	23,90	57,10	69,00
	3	3,80	15,50	23,10	61,00	76,00
	Mean	4,60	15,10	24,80	58,97	70,67

Table E.6: Preparation time in *without ordering* mode (in msec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	12,27	43,70	80,55	173,61	259,26
	2	11,77	46,73	77,43	174,03	264,87
	3	13,25	45,68	73,31	152,18	266,02
	Mean	12,43	45,37	77,10	166,60	263,38

Table E.7: Time for data transfer to and from R (in sec.)

R

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	1,40	2,40	3,90	9,40	26,20
	2	1,00	2,50	4,10	13,50	18,90
	3	1,00	2,20	3,50	9,10	11,10
	Mean	1,13	2,37	3,83	10,67	18,73

Table E.8: Time consumption by used MAL-instructions (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	$*$	2,6591	9,2369	19,3498	35,3222	54,9333
	$-$	0,7616	2,5941	5,5058	10,3352	15,7897
	sum	0,1511	0,5878	1,1764	2,3311	3,5729
	$/$	0,0807	0,1037	0,1524	0,1994	0,3300
	$srqt$	0,0001	0,0003	0,0005	0,0008	0,0012

Table E.9: Complete runtime in *with ordering* mode, ordered by one attribute and run in non-virtualized environment (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	0,4	1,1	2,2	3,9	6,2
	2	0,4	1,1	2,3	3,9	6,1
	3	0,4	1,1	2,3	4,0	6,0
	Mean	0,4	1,1	2,3	3,9	6,1
1M	1	3,6	9,0	17,3	29,1	44,7
	2	3,6	9,1	17,2	29,4	45,0
	3	3,6	9,1	17,4	28,8	45,6
	Mean	3,6	9,1	17,3	29,1	45,1

R

Number of tuples	Run	Number of attributes in the application part				
		20	40	60	80	100
100k	1	0,7	1,1	1,8	2,8	4,0
	2	0,5	1,0	1,8	3,2	4,0
	3	0,5	1,0	1,8	2,8	3,9
	Mean	0,6	1,0	1,8	2,9	4,0
1M	1	6,3	12,4	19,9	31,8	45,9
	2	4,6	11,3	20,1	31,2	48,4
	3	5,0	11,0	19,6	33,3	48,1
	Mean	5,3	11,6	19,9	32,1	47,5

Table E.10: Time consumption in *with ordering* mode, on relations with 1'000'000 tuples, odered by one attribute and run in non-virtualized environment (in sec.)

MonetDB

Number of tuples	Run	Number of attributes in the application part				
		5	10	15	20	25
Q_QR	1	1,7	5,4	11,4	19,5	30,9
	2	1,7	5,5	11,3	19,4	30,5
	3	1,7	5,4	11,4	19,5	30,2
	Mean	1,7	5,4	11,4	19,5	30,5
Q_QR	1	7,4	10,9	14,2	16,1	15,7
	2	7,4	11,0	13,8	15,1	15,8
	3	7,4	11,1	13,3	16,0	16,9
	Mean	7,4	11,0	13,8	15,7	16,1

List of Figures

List of Tables