Universidade do Minho
Escola de Engenharia
Departamento de Informática

Lucas Ribeiro Pereira

Analytical Querying with Typed Linear Algebra: Integration with MonetDB

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Lucas Ribeiro Pereira

Analytical Querying with Typed Linear Algebra: Integration with MonetDB

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Nuno Oliveira
Alberto José Proença

December 2021

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor José Nuno Oliveira and Professor Alberto Proença, my dissertation supervisors, for all the invaluable assistance and guidance, enthusiastic encouragement and useful critiques of this work. With their help, I was able to tackle many challenges that came my way.

While doing this work, I held a Research Grant AE2021-0049 awarded by INESCTEC, so I wish to thank INESCTEC and all the people involved in the project for the opportunity.

A special thank you to Patrícia Moreira, for the unconditional support given throughout this project, for always being there in the good and not so good moments, and for the encouragement to do the right thing.

I would like also, to thank my parents, my brother Ricardo and my sister Matilde. To my parents, thank you for investing on my education and for giving me the tools needed to accomplish my endeavours.

Finally, a special thank you to my grandfather José, for the love and affection I received, and for always being in my mind, giving me strength to achieve my dreams. Thank you, You are missed.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Current digital transformations in society heavily rely on safe, easy-to-use, high-performance data storage and analysis for smart decision taking. This triggered the need for efficient analytical querying solutions and the columnar database model is increasingly regarded as the most efficient model for data organization in large data banks. MonetDB is a pioneer in the column-wise database model and is currently at the forefront of high performance DBMS engine.

A Linear Algebra Querying (LAQ) engine, using a columnar database paradigm and strongly inspired on Typed Linear Algebra (TLA), was developed in a former MSc. dissertation, with a prototype Web interface. Performance benchmarking of this engine showed it outperformed conventional referenced DBMS but it failed to beat MonetDB's performance.

This dissertation aims to improve the performance of the LAQ engine by following a different path: instead of a standalone engine, the new approach implements the engine on top of MonetDB extended with RMA (Relational Matrix Algebra) and inspired by the TLA approach. This enables the use of LAQ scripting to replace the main stream relational algebra query language approach given by SQL.

Matrix operations commonly used in LAQ/TLA, such as matrix-matrix multiplication, Khatri-Rao product or Hadamard-Schur product, had to be implemented in RMA to shift from the relational algebra paradigm to TLA.

A thorough analysis of the MonetDB/RMA showed the need to implement key TLA operators that are not available at the frontend. Such operators were implemented and successfully tested and validated, paving the way to future benchmarking its performance with TPC-H/OLAP queries and consequent fine tuning of the engine.

KEYWORDS OLAP, Columnar DB, Typed Linear Algebra, Relational Matrix Algebra, MonetDB.

RESUMO

Atualmente, as transformações digitais na sociedade confiam fortemente no armazenamento e na análise de dados seguros, fáceis de usar e de alto desempenho para tomadas de decisão inteligentes. Este facto desencadeou a necessidade de soluções de consultas analíticas eficientes, em que o modelo de bases de dados colunar é cada vez mais considerado o modelo mais eficiente para organização de dados em grandes bancos de dados. MonetDB é um sistema pioneiro no modelo de bases de dados colunar e atualmente está na vanguarda de DBMS's de alto desempenho.

Um motor Linear Algebra Querying (LAQ), que usa o paradigma de bases de dados colunar e fortemente inspirado em Álgebra Linear Tipada (TLA), foi desenvolvido numa antiga dissertação de mestrado em Engenharia Informática. O benchmarking do desempenho deste motor mostrou que supera DBMS tradicionais, mas não conseguiu superar o desempenho do MonetDB.

Esta dissertação visa melhorar o desempenho do motor LAQ seguindo um caminho diferente: em vez de um motor autónomo, a nova abordagem implementa o motor sobre o motor do MonetDB estendido com RMA (Álgebra Relacional Matricial) e inspirado na abordagem de TLA. Isto permite o uso de scripts LAQ para substituir a abordagem da linguagem de consulta de álgebra relacional fornecida pelo SQL.

Operações de matrizes comumente usadas em LAQ / TLA, como multiplicação de matrizes, produto Khatri-Rao ou produto Hadamard-Schur, tiveram de ser implementadas em RMA para mudar do paradigma da álgebra relacional para TLA.

Uma análise completa do MonetDB / RMA mostrou a necessidade de implementar os principais operadores de TLA que não estão disponíveis no front-end. Esses operadores foram implementados, testados e validados com sucesso, abrindo caminho para um futuro benchmarking do seu desempenho com queries TPC-H / OLAP e consequente, ajuste do motor.

PALAVRAS-CHAVE OLAP, Base de dados colunar, Álgebra Linear Tipada, Álgebra Relacional Matricial, MonetDB.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

According to Salley and Codd (1998), "*[...] relational DBMS were never intended to provide the very powerful functions for data synthesis, analysis and consolidation that is being defined as multi-dimensional data analysis.*"

Database systems have been at the forefront of research in computing, fueled by the industry, due to the need not only for data storage but also for data analysis. In the age of information, more and more companies depend on services that allow them to make more informative and complete decisions based on their massive information data banks. The size of such data banks have been growing exponentially over the years and therefore, the need for reliable and fast services and technologies is a priority in today's world.

As research progresses and industry makes use of it, many companies opt to implement multidimensional analysis techniques in their data warehousing (DW) database systems — multiple categories into which the data are broken down — to maximize the value of the stored data, extracting as much useful information as possible. For example, a sales company might have several dimensions related to location(country, state), time(year, month, day), products(food, clothing, brand) and many more. Such a technology, referred to as Online Analytical Processing (OLAP) (Salley and Codd, 1998) (Chaudhuri and Dayal, 1997), provides a more efficient and robust way of organizing information leading to a more efficient way of querying such databases. As expected from these systems, in spite of the complexity and dimensions involved, their main focus is to minimize the querying response time.

Online analytical processing and its multidimensional analysis is based on the OLAP Hypercube (multidimensional cube) (Datta and Thomas, 1999). The Hypercube is an array-based multidimensional structure that extends the traditional two-dimensional table model, typically found in relational databases, with additional layers, more accurately named *dimensions*. In the example given previously, these would be location, time and products. *Dimensions* refer to the qualitative side of the Hypercube.

There is also the quantitative side, being the actual facts of the Hypercube which, referred to as *measures*, correspond to consolidated data like number of sales or prices of products. Along with these two distinct sets of attributes, come operations that can be performed with the Hypercube and its algebra (Datta and Thomas, 1999).

As studied by Macedo and Oliveira (2015) and Oliveira and Macedo (2017), there is a lack of a formal standard conceptual model for OLAP, able to unify the Hypercube algebra and semantics, both the qualitative side, the *dimensions*, and the quantitative side, the *measures*. As a result, the authors proposed Type Linear Algebra (TLA) to replace the mainstream Relational Algebra (RA) to encode and resolve OLAP queries. A full understanding of

RA and its higher abstraction language SQL can be consulted in Afonso (2018). As for TLA, the next chapter presents an overview of the research and results achieved so far by Macedo and Oliveira (2015) and Oliveira and Macedo (2017).

Taking such previous work into account, Afonso (2018) developed a novel database management system (DBMS) based on TLA and named Linear Algebra Querying (LAQ) engine. This engine takes as input the LAQ scripting language and produce a C++ equivalent version. It implements the building block of Linear Algebra, the matrix, and a minimal Linear Algebra (LA) kernel. The LAQ engine was tested and validated with queries from a standard benchmark suite, the TPC-H. When comparing its performance against competitive systems, namely MonetDB, the results were promising but did not top the latter. One main contribution was a paper submitted to VLDB (Afonso et al., 2018), whose reviews pointed to relevant clues to improve the work already done. With this knowledge in hand, MonetDB was taken as the target for a deeper study of its software stack and database kernel.

MonetDB is a state-of-the-art DBMS targeting mainly data warehousing systems. In constant development and improvement since 1993 (Idreos et al., 2012), MonetDB pioneered the column-wise database storage model that has reached industry-level usability in various fields, such as business and science. MonetDB is a full-stack software product which is the focus of Chapter three in this dissertation, which also exposes the "guts" of MonetDB and its internals, from the front-end layer to the database kernel.

Alongside of MonetDB, there is already database system in development at ETH Zurich that mixes Relational Algebra in Linear Algebra called Relational Matrix Algebra (RMA) (Dolmatova et al., 2020b). This novel system is an extension to MonetDB and already integrates some Linear Algebra operators. Through RMA, more operators will be defined that are needed to implement queries according to TLA theory.

1.1 CHALLENGES AND GOALS

The background of this dissertation starts with the work of Macedo and Oliveira (2015), who proposed a way to replace RA by TLA to encode and resolve OLAP queries. Afonso (2018) developed a novel DBMS based in TLA with a minimal LA kernel, testing and validating it against the standard TPC-H benchmark suite. The results were promising but did not top the fully fledged MonetDB DBMS.

This dissertation aims to make a change in the approach followed in such previous work: the goal is to take advantage of MonetDB and RMA software and address it from a TLA approach, using the web prototype built by Pereira and Baptista (2019), to bridge the user interface with a strong and fully fledged DBMS.

The key challenges and goals in this dissertation work include:

- to improve the web prototype and build a bridge that connects it to MonetDB;
- to use the key-value pair paradigm data-unit present in MonetDB and its extension RMA, to develop an approach to LA;
- to develop a minimal set of operations in MonetDB/RMA scripting language for TLA operators;
- to improve the MonetDB performance through TLA, namely in query plan optimization.

1.2 DISSERTATION OUTLINE

The current Chapter gave an overview of the work previously developed by the cited authors, helping in understanding the multitude of concepts that are embodied in such research work. Chapter 2 "lifts the lid" on what TLA is and how it works. Chapter 3 is an insightful look and explanation of the MonetDB software layers, the frontend, the backend and the database kernel. Chapter 4 addresses RMA system architecture, the newly defined matrix operators and how to integrate them in MonetDB. Chapter 5 goes through the new implemented operators that were missing, to be able to compute queries in compliance with TLA, including a case study. Chapter 6 concludes this document with a synthesis of the work carried out and presents suggestions for future work.

2

TYPED LINEAR ALGEBRA FOR OLAP

SQL is beyond doubt the mainstream query language for database engines underlying OLAP operations. However, as noted previously by Macedo and Oliveira (2015), there is a lack of a formal standard conceptual model for OLAP that unifies the Hypercube algebra and its semantics. Their proposal of Typed Linear Algebra (TLA) comes into play to improve RA (and consequently SQL) especially in the quantitative side of the operators, such as data aggregations or cross-tabulations, which are key opertors to deal with data analysis.

This Chapter provides an overview of the improved and more elegant way of approaching a query through TLA, with a minimal kernel of LA operators and its data representation. For the interested reader, Macedo and Oliveira (2015) and Oliveira and Macedo (2017) can be consulted to develop a deeper understanding of TLA.

2.1 FOUNDATION OF TLA QUERYING AND TYPE DIAGRAMS

As a starting point for describing the TLA approach, consider the following example with two relational tables (Table 1): a *job* table with a job code plus its description and monthly salary, and an *employees* table with an identification number, a reference to the job assigned to the employee, plus the employee's name, the branch of activity and the country.

j_code	j_desc	j_salary
PR	Programmer	1000
SA	System Analyst	1100
GL	Group Leader	1333

e_id	e_job	e_name	e_branch	e_country
1	PR	Mary	Mobile	UK
2	PR	John	Web	UK
3	GL	Charles	Mobile	UK
4	SA	Ana	Web	PT
5	PR	Manuel	Web	PT

Table 1: Jobs and employees tables

These two tables can be combined into a single diagram, more visual appealing to the eye, giving birth to the so called Type Diagram (TD), Figure 1. TDs abstract from the unnecessary knowledge of the actual facts that tables contain, allowing a graphical way of representing a full database schema, as also designing and planing every possible query.

Figure 1, shows two main entities: *#j* and *#e*. These entities represent the cardinality of jobs and employees table records, respectively, and can be seen as the "two centers" of the TD. Surrounding both entities and with the

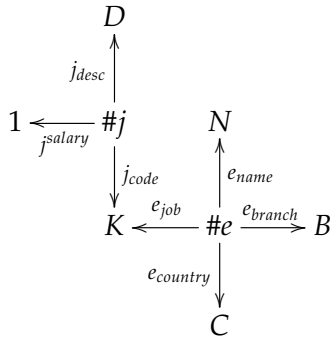

Figure 1: Jobs and employees type diagram

origin in them, there are multiple *arrows*, j_{code} or j_{desc} for example, referring to every attribute that each entity has, correlating to each table. At the end point of each *arrow* are the number of possible values that each attribute has. In sum, each *arrow* represents a *binary relation* between two *types*. The careful reader may be able to identify a difference between two notations in the TD, since one stands out for having the correspondent attribute in exponential (Figure 2), contrasting with all others that are at the index (Figure 3, for example).

$$1 \xleftarrow[j^{salary}]{} \#j$$

Figure 2: Relation j^{salary}

$$K \xleftarrow[j_{code}]{} \#j$$

Figure 3: Relation j_{code}

This is an important distinction in the notation made to identify the notions of *measure* and *dimension*, in Figures 2 and 3: the quantitative and the qualitative attributes, respectively.

Generally speaking, tables like Table 1 can be broken into sub-tables according to their attributes, giving rise to a *binary relation* for each sub-table created. Each *binary relation*, represented by an arrow, has finite types associated, given by their cardinalities, with the exception that if the *binary relation* refers to a *measure*, its particular type is the unitary type, *one*.

2.2 MATRICES AS ARROWS

In the previous section, one deconstructed the TD from Figure 1 and arrived to the construction of a single *arrow*. What if now, one thinks of arrows as a matrices ?

Effectively, this question is the core of TLA. From this point on, every arrow represents a matrix, the latter being the data representation used by linear algebra. More, arrow notation is usually associated with the representation of functions, creating the notion that a matrix can be thought of as a function, as shown in Figure 4.

$$b \xleftarrow[M]{} a$$

Figure 4: Matrix M from type a to type b

Understanding how information initially encapsulated in relations gets converted to matrices is straightforward: take the jobs table in Figure 1 and, for each attribute, create a sub-table, as shown in Table 2.

j_{code}	1	2	3
GL	0	0	1
PR	1	0	0
SA	0	1	0

j_{desc}	1	2	3
Group Leader	0	0	1
Programmer	1	0	0
System Analyst	0	1	0

Table 2: j_{code} and j_{desc} sub-tables

The j_{code} and j_{desc} are attributes that represent *dimensions*, and therefore result in matrices that are refereed as bitmaps. As stated previously, matrices can be thought as functions. In the case of the arrow representing j_{code} (Figure 3), it receives an argument of type *#j*, the cardinality of records in the jobs table, and produces a value of type *K*, the cardinality of the distinct values that j_{code} can assume. In sum, for a given row of the table *jobs*, its corresponding column in the matrix j_{code} identifies the attribute value (K), by checking the row which contains the value "1".

j^{salary}	1	2	3
1000	0	0	1
1100	1	0	0
1333	0	1	0

j^{salary}	1	2	3
1	1000	1100	1333

Table 3: j^{salary} sub-table

As what concerns j^{salary}, this attribute of the jobs table represents a *measure*, and instead of representing *measures* as bitmaps (Figure 3 left), these will collapse into a specific matrix structure with only one row (Figure 3 right), the *vector*.

Arranging all three matrices through their arrow notation, one can build the jobs table TD (Figure 5).

If the same process is applied to the employees table, this will get as a result the TD shown in Figure 1.

2.3 TLA ALGEBRAIC QUERYING OPERATORS

Queries in TLA are represented by LA expressions that emerge from TD such as Figure 1. Algebraic expressions are used to simplify a query, step by step. One can go to a TD of a database schema and "choose a path" that will represent a query. With this "path chosen" and using TLA algebraic operators, one can start simplifying the original TD until it reaches a single arrow (algebraic expression).

From the range of the LA operators, three key algebraic operators essential for TLA query resolution are (i) the dot product, (ii) the Khatri-Rao product and (iii) the Hadamard-Schur product. Complementing these operators are

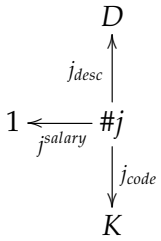

Figure 5: Jobs TD

three more derivations: filtering, folding and lifting. Each of these algebraic operators will be briefly explained, first through their respective TD and later through LA operators.

2.3.1 *Matrix-matrix multiplication*

$C = A \,.\, B$ describes the multiplication of matrices *A* and *B*, where each element of the resulting matrix *C* is the dot product of a row in *A* with a column in matrix *B*. The operator for this matrix multiplication in this document is referred to as a dot product. As a requirement for this operation, the number of columns in *A* needs to match the number of rows in *B*. Therefore, the dimension of matrix *A* must be $i \times k$ and matrix *B* must be $k \times j$, where the first element is the number of rows and the second element is the number of columns. The dimension of the result matrix *C* is $i \times j$.

An element in C — c_{xy}, where *x* represents the row number and *y* represents the column number — is given by the following definition of the dot product:

$$c_{xy} = a_{x1} \times b_{1y} + a_{x2} \times b_{2y} + ... + a_{xk} \times b_{ky} = \sum_{n=1}^{k} a_{xn} \times b_{ny} \tag{1}$$

A step by step example can be followed below.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} . \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1\times1+2\times4 & 1\times2+2\times5 & 1\times3+2\times6 \\ 3\times1+4\times4 & 3\times2+4\times5 & 3\times3+4\times6 \\ 5\times1+6\times4 & 5\times2+6\times5 & 5\times3+6\times6 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 15 \\ 11 & 26 & 33 \\ 29 & 40 & 51 \end{bmatrix}$$

Figure 6: Example of a matrix-matrix multiplication

Figure 7 is the TD representation of the dot product operation.

Thinking of matrices as functions, one can identify a similarity between functional composition and matrix multiplication. Both have to meet the similar requirement that matrices dimensions have to align just like types in function composition, allowing the representation of this operator by $C = A \,.\, B$.

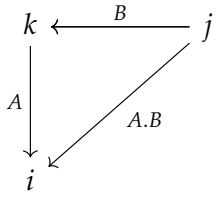

Figure 7: Dot product TD

2.3.2 *Khatri-Rao product*

$C = A \bigtriangledown B$ describes the Khatri-Rao product between matrices *A* and *B* with *C* as result. As a requirement for this operation, the number of columns in *A* needs to match the number of columns in *B*. Therefore, *A* dimension must be $i \times k$ and *B* dimension $j \times k$. The dimension of the result matrix *C* is $(i * j) \times k$.

Values in C are obtained by multiplying each row of *A* by the whole matrix *B*, row by row.

A step by step example can be followed below.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \times 1 & 2 \times 2 \\ 1 \times 3 & 2 \times 4 \\ 1 \times 5 & 2 \times 6 \\ 3 \times 1 & 4 \times 2 \\ 3 \times 3 & 4 \times 4 \\ 3 \times 5 & 4 \times 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 8 \\ 5 & 12 \\ 3 & 8 \\ 9 & 16 \\ 15 & 24 \end{bmatrix}$$

Figure 8: Khatri-Rao product example

Figure 9 is the TD representation of the Khatri-Rao product operation.

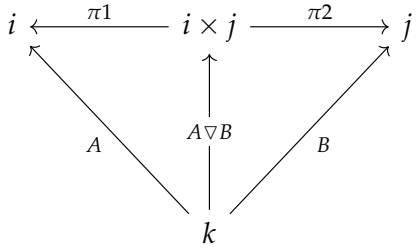

Figure 9: Khatri-Rao TD

Playing the same analogy game as in the dot product operation, the similarity this time is between the functional split, defined by $(f \bigtriangledown g)x =< f(x), g(x) >$. One special property of this operator is that there is no loss in information, since the original matrices can be retrieved.

2.3.3 *Hadamard-Schur product*

$C = A \times B$ describes the Hadamard-Schur product between matrices *A* and *B* with *C* as result. As a requirement for this operation, both matrices must have the same dimensions. Therefore, *A* dimension must be $i \times j$ and *B* dimension $i \times j$. The dimension of the result matrix *C* is $i \times j$ as well.

Values in C are obtained by multiplying each element of *A* by the correspondent element in matrix *B*, being a point to point matrix multiplication.

A step by step example can be followed below (Figure 10).

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 2 \times 6 \\ 3 \times 7 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

Figure 10: Hadamard-Schur product example

2.3.4 *Filter*

The filter operation is the equivalent to the relational selection. It filters the columns of a matrix based on the labels of its corresponding rows. This filter is obtained by combining a predicate and a matrix as input. Applying the predicate directly to the matrix labels, generates a Boolean vector stating which labels comply with the predicate. The next step is applying the dot product between the newly created vector and the initial matrix, generating another Boolean vector, identifying the columns that have a row satisfying the predicate. Figure 11 represents the filter operation TD.

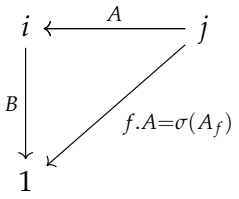

Figure 11: Filter TD

Below, an example of the filter operator is given.

2.3.5 *Fold*

"!" or *bang*, is a vector of arbitrary length, with all its elements being "1". It is mainly used to condense information, reducing a matrix to a single value. Combining *bang* with dot product, allows for the implementation of aggregation function such as *sum, count, avg, min or max*. The TD below (Figure 14) shows how one can combine the operators.

$$(\neq SA) = \begin{bmatrix} GL & PR & SA \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{array}{ccc|c} 1 & 2 & 3 & j_{code} \\ \hline 0 & 0 & 1 & \text{GL} \\ 1 & 0 & 0 & \text{PR} \\ 0 & 1 & 0 & \text{SA} \end{array}$$

Figure 12: Dot product between the predicate $(\neq SA)$ and the initial matrix

$$\sigma(j_{code} \neq SA) = \begin{array}{ccc} \multicolumn{3}{c}{\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}} \\ \hline \multicolumn{3}{c}{\begin{bmatrix} 1 & 0 & 1 \end{bmatrix}} \end{array}$$

Figure 13: Final result of the filter operation

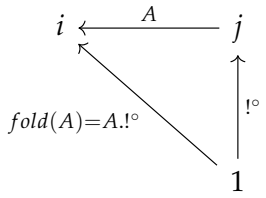

Figure 14: Fold TD

Figure 15 illustrates one use of the aggregation function *sum*.

$$sum(j^{salary}) = \begin{bmatrix} 1000 & 1100 & 1333 \end{bmatrix} . \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 3433$$

Figure 15: Fold example

2.3.6 *Lift*

The lift operator is the application of a mathematical expression to a vector, or a set of vectors, corresponding to the function arguments. This is done for each element in the provided vector(s), creating a new vector(s) as shown in Figure 16.

$$lift(2 \times j^{salary}) = 2 \times \begin{bmatrix} 1000 & 1100 & 1333 \end{bmatrix} = \begin{bmatrix} 2000 & 2200 & 2666 \end{bmatrix}$$

Figure 16: Lifting example

2.4 QUERY EXAMPLE

As an example to illustrate the TLA operators and data model working, the database shown in Figure 1 is going to be used to perform the following query: obtain the total monthly salary per country/branch, ordering the result by countries.

This query involves three main attributes from the TD shown in Figure 1: the salary of a job and all the countries and activity branches of all employees. From this point, one can start cleaning up and preparing the TD from the specific query to achieve, drawing the specific paths and removing entities that are not necessary, as show in Figure 17.

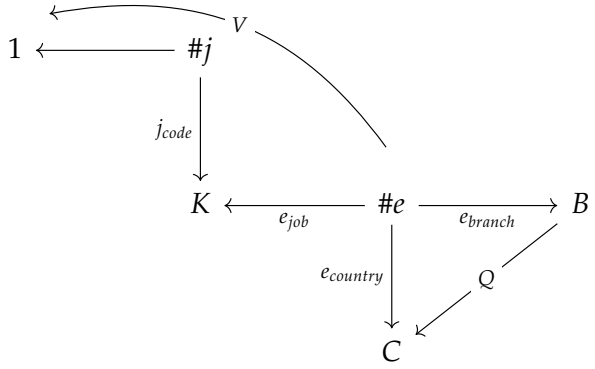

Figure 17: Query type diagram

One can identify two new arrows in the TD, *V* and *Q*. *Q* is the tabulation that is required between (and from) activity branches and (to) countries between employees. *V* is the result of getting every salary of each employee.

Both these results, *Q* and *V*, can be combined into a single TLA script:

$$Q = e_{country} \,.\, (V \,\triangledown\, ID) \,.\, e^{\circ}{}_{branch} \qquad where\ V = j^{salary} \,.\, j^{\circ}{}_{code} \,.\, e_{job} \tag{2}$$

As seen before in Section 2.2, every arrow (or relation) gives place to a specific matrix and what is show in the TLA script is indeed a chain of arrows, a path, and therefore a chain of matrices. This path is resolved by TLA operators, namely dot and Khatri-Rao products of matrices to achieve the result of the query.

$e_{country}$	mobile	web
PT	0	2100
UK	2333	1000

Table 4: Query result

2.5 SUMMARY

This chapter gave an overview of the TLA proposal for data representation and querying, with an example. TLA provides the encoding of relational tables into matrices through attribute-wise partitioning of relational data, thus allowing for an inherently column-wise database approach.

Matrices and their dimensions can be described through the arrow notation, with types being the matrices dimensions, enabling the use of TD for a graphical and more elegant way of representing database schema and its queries. Through this notation of matrices seen as function, TLA ensures type correctness by construction.

3

ARCHITECTURE OF MONETDB

MonetDB is a state-of-the-art Database Management System (DBMS) developed at the Centrum Wiskunde & Informatics (CWI, Netherlands) in 1993 and now owned by *MonetDB Solutions* company. This DBMS is a pioneer in the development of a column-wise approach to database storage, providing a modern and scalable solution for the current systems.

Besides the column storage main feature, MonetDB also aimed to innovate all layers of its software stack, with a CPU-tuned query execution architecture, indexing features, a run-time query optimizer and others, keeping in mind a good modular software architecture. At the front-end of the software layers, the main interaction language between a user and the DBMS is *SQL*, based on the *SQL 2003 standard*, offering full support for foreign keys, joins, views, triggers, and stored procedures. MonetDB complies with the ACID standard model and supports a rich spectrum of programming interfaces such as Java Database Connectivity, Python, C/C++ and others.

MonetDB features a three-level software stack. The first level contains a SQL interpreter and a translator, the second level is a combination of query plan tactical optimizers and the last level is a columnar abstract-machine kernel. Good modularity at all layers allows for external developers and researchers to build on top of MonetDB or even modify one or more layers.

MonetDB shines in applications where the database hot-set can largely fit in main-memory, exploiting cache-awareness algorithms to further improve its performance. Due to the columnar approach, MonetDB takes great advantage of its multi-core parallel execution.

3.1 THE FRONTEND

The first layer of MonetDB contains the *SQL* interpreter and a translator to a domain specific language. Through a client program, a user can interact with the server, using *SQL* as the input language. Upon receiving a *SQL* query, the client program translates it into a MonetDB Assembly Language (MAL) query. Appendix A.1 shows a script that loads the database schema used in chapter two into MonetDB.

3.1.1 *MonetDB Assembly Language and its algebra*

MAL is the Domain Specific Language (DSL) of the database kernel. This DSL reflects the virtual machine architecture of the kernel libraries, being designed for speed of parsing, ease of analysis and ease of target compilation by query compilers. A MAL program is considered of an intended computation and data flow behaviour, adopting a functional style definition of actions.

MAL is the target language for query compilers at the frontend layer. Even simple SQL queries, as the one showed in Listing 3.1, can generate a long sequence of MAL instructions, as shown in Listing 3.2.

```
SELECT e_name
FROM "empl";
```

Listing 3.1: Simple query example

```
1  function user.main():void;
2       querylog.define("explain\nselect e_name\nfrom \"empl\";":str, "default_pipe
            ":str, 19:int);
3    barrier X_80:bit := language.dataflow();
4       X_24:bat[:str] := bat.pack("sys.empl":str);
5       X_25:bat[:str] := bat.pack("e_name":str);
6       X_26:bat[:str] := bat.pack("char":str);
7       X_27:bat[:int] := bat.pack(15:int);
8       X_28:bat[:int] := bat.pack(0:int);
9       X_4:int := sql.mvc();
10      C_5:bat[:oid] := sql.tid(X_4:int, "sys":str, "empl":str);
11      X_17:bat[:str] := sql.bind(X_4:int, "sys":str, "empl":str, "e_name":str, 0:
            int);
12      X_22:bat[:str] := algebra.projection(C_5:bat[:oid], X_17:bat[:str]);
13   exit X_80:bit;
14      sql.resultSet(X_24:bat[:str], X_25:bat[:str], X_26:bat[:str], X_27:bat[:int
            ], X_28:bat[:int], X_22:bat[:str]);
15   end user.main;
```

Listing 3.2: MAL program

Listing 3.2 is effectively the MAL program of the *SQL* query in Listing 3.1. One can obtain such description by prepending the *EXPLAIN* statement to the query script. With this statement, one can have every translation from a *SQL* query to a MAL program, in a step by step way.

A MAL program can be broken essentially into three parts: (i) a group of setup operations, (ii) another group with real computations using MAL algebra, and (iii) a return statement with the query result. If one takes a look to Listing 3.2, it is possible to identify the groups within these statements. From line 2 to line 10, this group of statements corresponds to the setup statements.

The *bat.pack* operation materializes the received as argument into a Binary Association Table (BAT) which is the key data structure for the columnar representation of MonetDB. BATs will be explained in a section ahead. The *sql.mvc* operation is a loading mechanism of the *SQL* subsystem of MonetDB. As for the *sql.tid* operation, this gives the knowledge to the database of what tables are actually visible to operate on given the user that is using the client. Finally closing the setup group statements, the operation *sql.bind* loads the table received as an argument to a BAT, effectively materializing it.

Jumping to the MAL algebra group, line 12, one encounters the *algebra.projection*. In this example it is easy to compare the original *SQL* query to this operation since this is just a attribute selection, but for the purpose of a simple demonstration of what a MAL program is, choosing the most simple query possible is the optimal solution given that MAL programs can grow disproportionally while comparing to the *SQL* version.

The final group is the single statement in line 14, *sql.resultSet* operation prepares the result table that is going to be sent to the front-end client.

All these mentioned operators are just a small part of a much bigger set of operators that are implemented in MonetDB. A full compendium of the MAL algebra operators can be found in the appendix B.

3.1.2 *Query planning*

Alternatively to the *EXPLAIN* statement, one can use the *PLAN* statement. This results in a more simpler view of how the query will carry its execution.

Query plans are much more compact then *query traces* and therefore a more complex query (in Listing 3.3) can be used to demonstrate how the result query is shown to the user.

```
SELECT e_branch, e_country, sum(j_salary)
FROM empl, jobs
WHERE j_code = e_job
GROUP BY e_country, e_branch
ORDER BY e_country;
SELECT e_name
```

Listing 3.3: Query 2 example

```
1  project (
2  | group by (
3  | | join (
4  | | | table("sys"."empl")
5  | | | table("sys"."jobs")
6  | | ) [ "empl"."%empl_e_job_fkey" NOT NULL = "jobs"."%TID%" NOT NULL JOINIDX "
       sys"."empl"."empl_e_job_fkey" ]
7  | ) [ "empl"."e_country" NOT NULL, "empl"."e_branch" NOT NULL ] [ "empl"."
       e_country" NOT NULL, "empl"."e_branch" NOT NULL, "sys"."sum" no nil ("jobs".
       "j_salary" NOT NULL) NOT NULL as "%1"."%1" ]
```

```
8   ) [ "empl"."e_branch" NOT NULL, "empl"."e_country" NOT NULL, "%1"."%1" NOT NULL
        ] [ "empl"."e_country" ASC NOT NULL ]
```

Listing 3.4: Query 2 MAL Plan

With a careful observation of Listing 3.4, one can understand it straightforwardly and make a good comparison with the *SQL* query.

3.2 THE BACKEND

After processing the *SQL* query and translating it into a *MAL program*, the MonetDB server has a pipeline mechanism to improve that same *MAL program*. This pipeline of optimizers has operations like removing dead code or garbage collection of temporary variables.

3.3 THE KERNEL

3.3.1 *Goblin Database Kernel*

The innermost library of the MonetDB database system is formed by the library called the Goblin Database Kernel (GDK). Its development was originally rooted in the design of a pure active-object-oriented programming language, before development was shifted towards a re-usable database kernel engine.

GDK is a C library that provides ACID properties, using main-memory database algorithms built on virtual-memory OS primitives and multi-threaded parallelism. The GDK implements operators that work directly with MonetDB data unit, the BAT, and supports one built-in search accelerator for hash tables, to accelerate lookup and search in BATs, given that hash tables implementation is efficient for main-memory.

3.3.2 *The data model with Binary Association Tables*

A BAT is a self-descriptive main-memory structure that represents the binary relationship between two atomic types, being implement as an *array*.

BATs are built on the concept of heaps, but if the system grows out of memory, GDK supports operations that cluster BAT heaps, to improve the performance of its main-memory. All BATs are registered in the BAT buffer pool. This directory is used to guide swapping in and out of BATs. Table 5 and 6 show how a relational table can be represented in a BAT form.

j_code	j_desc	j_salary
PR	Programmer	1000
SA	System Analyst	1100
GL	Group Leader	1333

Table 5: Jobs relation table

OID	j_code
100	PR
101	SA
102	GL

OID	j_desc
100	Programmer
101	System Analyst
102	Group Leader

OID	j_salary
100	1000
101	1100
102	1333

Table 6: Jobs relation table with BAT representation

3.4 SUMMARY

This chapter was devoted to dissecting the MonetDB architecture and its internals. Binary Association Tables are a key finding for the development of this work, because a BAT is essentially a key-value pair data structure, effectively bringing MonetDB and TLA closely together.

4

RELATIONAL MATRIX ALGEBRA

The Relational Matrix Algebra system, RMA, is effectively an extension to MonetDB developed at ETH Zurich by Dolmatova et al. (2020a), Dolmatova et al. (2020b). The authors proposed an extension to the relational model with matrix operations, without modifying the MonetDB main data structure, the BATs, or its process pipeline, leading to not affecting the existing functionality and proven performance.

The first iteration of RMA implements a set of basic operations such as matrix addition or matrix cross-product. These operations were carefully chosen in order to develop a set of performance benchmarks to be compared with the original DBMS MonetDB. Its notation and theory will be briefly described, with the concern of trying to find parallel concepts to the TLA theory. Chapter 5 will specifically address which characteristics appear to have similarities and what was done to further explore this concept of using LA in the relational model.

The extension was done on all the three layers in MonetDB, from the frontend — extending the SQL interface to be able to recognize the new matrix operators — to the backend and, consequently, the Goblin Database Kernel, adding functionally at the level of the MAL.

4.1 NOTATION OF RELATIONS AND MATRICES

Consider a relation like the one in Table 7, with m tuples.

j_{code}	j_{desc}	j^{salary}
PR	Programmer	1000
SA	System Analyst	1100
GL	Group Leader	1333

m	1
1	PR
2	SA
3	GL

Table 7: Jobs relation table and matrix M

RMA defines some notations and operators that are necessary to better understand the new concepts:

- the schema of relation j_{code} is the set of attributes $\{j_{code}, j_{desc}, j^{salary}\}$;
- $|j_{code}|$ denotes the number of tuples and $\# j_{code}$ denotes the number of columns (attributes) in relation j_{code};

- column cast of attribute U, $\triangledown U$, is a operation that creates a set of ordered values of an attribute U that forms a key in a relation and is to generate a schema, for example, $\triangledown j_{code}$ = (PR, SA, GL);
- schema cast of set U, $\triangle U$, is a operation that creates a matrix with a single column from the attributes of a set, for example, considering the set U = (PR, SA, GL), $\triangle U$ creates matrix *M* in Table 7;
- matrix concatenation, $|A \square B|$, is the operation of concatenating matrix *A*, with |A| = m and #A = k, and matrix *B*, with |B| = n and #B = k;
- matrix operations are shape restricted, meaning, the number of result rows is equal to the number of rows of one of the input matrices (r), the number of columns of one of the input matrices (c), or one (1). The same holds for the number of result columns;
- this shape restriction defines the shape type of the matrix operations, writing r_1 if the dimension of the result is equal to the number of rows in the first matrix, r_2 if the dimension of the result is equal to the number of rows in the second matrix, and r_* if the result dimension is equal to the number of rows in the first and second matrix (meaning, $r_1 = r_2$); the same notation holds for the number of columns.

Table 8 presents the shape types of the newly added matrix operations in RMA.

Cardinality	Shape type	Operations
$\|a_1 \times b_1\|, \|a_1 \times b_1\| \longrightarrow \|a_1 \times b_1\|$	(r_*, c_*)	ADD
$\|a_1 \times b_1\| \longrightarrow \|a_1 \times b_1\|$	(r_1, c_1)	QQR
$\|a_1 \times b_1\|, \|a_1 \times 1\| \longrightarrow \|b_1 \times 1\|$	(r_*, c_*)	SOL
$\|a_1 \times b_1\|, \|a_1 \times b_2\| \longrightarrow \|b_1 \times b_2\|$	(c_1, c_2)	CPD

Table 8: Shape types of matrix operations

Additionally, a relation can always be divided in four parts: *order schema* - containing the attributes, one or more, that form the key of the relation; *application schema* - containing all the attributes that are not key in the relation; *order part* - contains the values (or tuples if the key has more then one attribute) of the order schema; *application part* - contains all the tuples formed by the application schema. Table 9 is the example of all the parts that relation j_{code} can be divided in.

j_{code}

j_{desc}	j^{salary}

PR
SA
GL

Programmer	1000
System Analyst	1100
Group Leader	1333

Table 9: Order schema, application schema, order part, application part of relation j_{code}, respectively

4.2 RELATIONS AND MATRICES CONSTRUCTORS

To define the relational matrix operations, two relevant definitions are needed to make the switch from relations to matrices and vice versa: the matrix constructor and the relation constructor.

A matrix constructor, μ, takes a set of values and constructs a matrix from them. For instance, looking at the order schema of the relation j_{code}, $\mu_{j_{code}} j_{code}$ returns the matrix containing the values of the set that is the order part of the relation (Table 9).

Relation constructor, γ takes a matrix and a schema and builds a relation from these two. For instance, Table 9 has a set in application schema and a matrix in application part. $\gamma(application\ schema, application\ part)$ creates the relation that can be found in Table 10.

j_{desc}	j^{salary}
Programmer	1000
System Analyst	1100
Group Leader	1333

Table 10: Relation constructed from $\gamma(application\ schema, application\ part)$

With these two key constructors definitions plus the the previous mentioned notations and definitions, the matrix relational operators are now ready to be defined.

4.3 FROM RELATIONS TO MATRICES AND BACK

Consider relations *r*, *s* and the following query: perform addition on relations *r* and *s*. The result relation is composed from the order schema, order part, application schema and application part, with the help of the constructors of relations and matrices.

U	X	Y
A	1	2
B	3	4

T	W	Z
C	4	1
D	1	2

Table 11: Relation *r* and *s*, respectively

Tables in 12 and 13 are the result of breaking down relations *r* and *s* with matrix constructors.

U	X	Y

	1
1	A
2	B

T	W	Z

	1
1	C
2	D

Table 12: Breakdown in parts of relations *r* and *s*

With both relations applications part, the addition operator can now be applied to both matrices, with the result shown in Table 14.

	1	2
1	1	2
2	3	4

	1	2
1	4	1
2	1	2

Table 13: Application part of relations *r* and *s*

	1	2
1	5	3
2	4	6

Table 14: Result of addition operation

Once the operation in the application parts is concluded, the contextual information necessary in order to keep the result coherent needs to fall in place. For this to happen, the concatenation operator, $\square$, is used to "glue" back together the order part from both relations *r* and *s* to the result of the addition. After the concatenation, the result is a matrix, Table 15, with the contextual information needed and the result of the addition. One more step is needed in order for the result to be a valid relation and not a matrix.

	1	2	3	4
1	A	C	5	3
2	B	D	4	6

Table 15: Concatenation of the contextual information with the base result

The last step is building a relation through the relation constructor, γ. This constructor for the addition operator takes as argument the matrix in Table 15 and schemas *U* and *T* and returns the final and valid result.

U	T	X	Y
A	C	5	3
B	D	4	6

Table 16: Result relation to the query

Figure 18 illustrates the big picture of how RMA breaks down a query in order to solve it and return a valid result.

4.4 OPERATIONS IN RMA

Table 17 shows all available relational matrix operations available in RMA. RMA currently has one unary operation, matrix qr decomposition (*qqr*), and 3 binary operations, matrix addition (*add*), solve equation system (*sol*) and the matrix cross-product (*cpd*). Every relational matrix operation is defined with the relation or relations that is going to work on and their specific schemas.

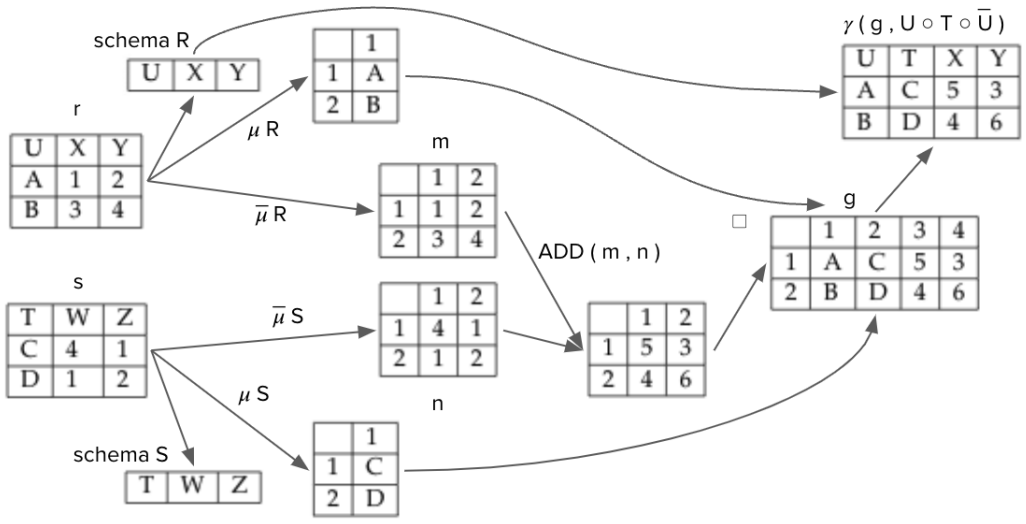

Figure 18: Addition over relations r and s

Shape type	**Operation**	**Definition**
(r_*,c_*)	add	$op_{U;T}(r,s) = \gamma(\mu_U(r) \square \mu_T(s) \square OP(\overline{\mu}_U(r), \overline{\mu}_T(s)), U \circ V \circ \overline{U})$
(r_1,c_1)	qqr	$op_U(r) = \gamma(\mu_U(r) \square\ OP(\overline{\mu}_U(r)), U \circ \overline{U})$
(c_1,c_2)	sol	$op_{U;T}(r,s) = \gamma(\Delta\overline{U} \square OP(\overline{\mu}_U(r), \overline{\mu}_T(s)), (C) \circ \overline{T})$
(c_1,c_2)	cpd	$op_{U;T}(r,s) = \gamma(\Delta\overline{U} \square OP(\overline{\mu}_U(r), \overline{\mu}_T(s)), (C) \circ \overline{T})$

Table 17: Defining available operations in RMA

Notice that RMA has no relational matrix operation for matrix multiplication nor matrix transpose.

4.5 RMA IMPLEMENTATION IN MONETDB

The implementation of relational matrix operations includes the processing of context information and the calculation of basic results. Context information is processed internally in MonetDB, and the calculation of basic results can be done in MonetDB or delegated to an external library (such as Math Kernel Library). The integration of each relational matrix operation required extensions in the entire system, but with the key point that no new data structures are created neither changes to the query processing pipeline.

Figure 19 illustrates the algorithm that computes the addition over two relations, on the desired attributes:

- splitting separates the contextual information from the application part of both relations;
- sorting orders application parts according to the sorted contextual information;
- morphing puts together the contextual information according to the relation matrix addition definition in table 17;

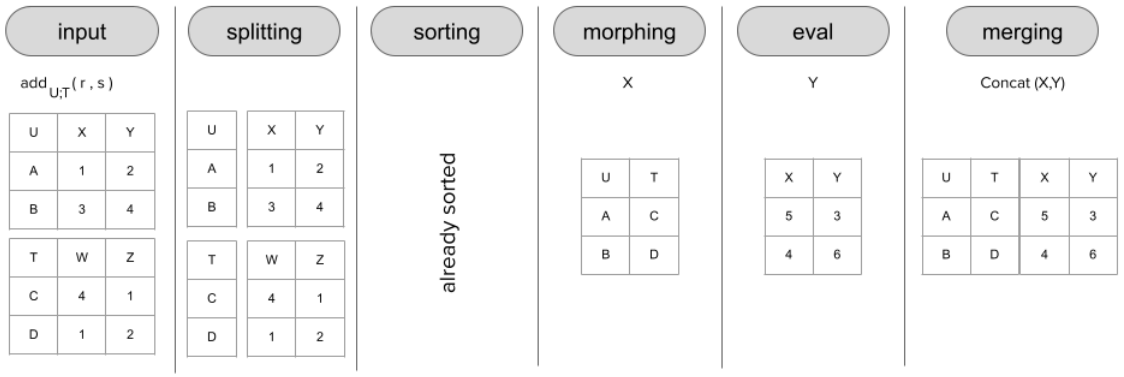

Figure 19: Splitting, sorting, morphing eval and merging in query $v = add_{U;T}(r, s)$

- eval computes the matrix addition; this operation can be computed inside MonetDB or can be delegated to an outside library;
- merging constructs a valid result through the relation constructor concept, as seen in section 4.2.

4.5.1 *SQL extension*

MonetDB SQL parser was extend to make the relational matrix operations available, using the **FROM** clause.

```
SELECT *
FROM (r on X,Y) add (s on W,Z);
```

Listing 4.1: SQL extension

Listing 4.1, illustrates how queries are accepted as input in the RMA extension to MonetDB system. In the **FROM** clause, the user needs to specify what attributes of both relations the RMA should perform the matrix addition. RMA supports nested **SELECT** clauses for more complex queries.

The **WITH .. AS** clause is also very useful for query "cleanness" and "readability", as the reader will notice in Chapter 5.

4.6 SUMMARY

This chapter was devoted to dissecting the RMA theory and how it is implemented in MonetDB. TLA and RMA appear to share similar aims although through different approaches. RMA still has the strong component of relational algebra that comes from MonetDB. This component in TLA is somewhat hidden behind composing matrices or other operations, and its strong type system. Some examples are shown of how a query is computed.

In particular, relational matrix operation cross-product, *cpd*, is going to be very relevant for the work described in chapter 5.

5

TLA EXTENSION IN RMA

This Chapter describes how TLA query plans and operators can be implemented in RMA, and consequently on MonetDB. The main goal is to see if there are any benefits to the alternative query plans that are proposed by TLA, since these queries represent a different approach in the sense that they both aim to maximize the use of linear algebra.

Since RMA does not have all matrix operations necessary for working with TLA, these operations had to be derived from existing relational matrix operations, namely the cross-product. Section 5.2 will give the reader the opportunity to follow along the use of the newly derived operations with an example, the same example that was introduced in Section 2.4.

5.1 THE MISSING OPERATORS

As seen in Chapter 2, two operations in TLA are essential to build queries and to "follow paths" in type diagrams: the *matrix transpose*, or matrix converse, and the *dot product* in matrices, which correlates to functional composition. Both are based on the definition of the relational matrix cross-product, present in RMA.

jcode	c1	c2	c3
GL	0	0	1
PR	1	0	0
SA	0	1	0

ejob	c1	c2	c3	c4	c5
GL	0	0	1	0	0
PR	1	1	0	0	1
SA	0	0	0	1	0

Table 18: Bitmaps of attributes *jcode* and *ejob* of the jobs and employees relations

Given two relations, X and Y, in which both application parts are numerical, the relational matrix cross-product between these relations is given by:

$$CPD(X,Y) = X^{\circ} \cdot Y \tag{3}$$

To compute the relational matrix cross-product between relations *jcode* and *ejob* (Table 18), the following query is executed in the RMA system:

```
SELECT *
FROM (jcode on c1,c2,c3) cpd (ejob on c1,c2,c3,c4,c5);
```

Listing 5.1: Compute the CPD between *jcode* and *ejob*

The produced result is shown in Table 19.

$jcode^{\circ} \cdot ejob$	c1	c2	c3	c4	c5
c1	1	1	0	0	1
c2	0	0	0	1	0
c3	0	0	1	0	0

Table 19: Relational matrix cross-product betweent *jcode* and *ejob*

5.1.1 *Identity matrix*

Although RMA has no definition to transpose a matrix or to multiply two matrices, these two operations are being internally computed in the algorithm of relational matrix cross-product. This key finding is going to be leveraged to produce and define new relational matrix operations.

A key concept to derive these new operations is the use of the *Identity matrix*, **ID**. The *Identity matrix* of size *N* is the square matrix with *N* rows, *N* columns and with ones on the diagonal and zeros everywhere else, as shown in Table 20.

1	0	0
0	1	0
0	0	1

Table 20: Identity matrix with N=3

In linear algebra, *identity matrices* act as the neutral element in dot-product operations. This property is going to prove useful when deriving the relational *matrix transpose* (converse) and the *dot product* operations.

5.1.2 *Matrix transpose*

In linear algebra, the transpose of a matrix is an operation that "mirrors the input matrix over its diagonal". In other words, for each element of that matrix, it switches the element row and column indexes.

Effectively, when defining relational *matrix transpose* in RMA, there are two problems that need to be solved: the need to keep contextual information consistent, and to perform the *matrix transpose* on the application part of the relation that is being transposed.

Given a relation *X*, the relational *matrix transpose* of *X* can be obtained by:

$$TRA(X) = CPD(X, ID) \tag{4}$$

In fact, once the equality of Equation (4) is expanded, one can see how from the relational matrix cross-product the relational *matrix transpose* operation is obtained:

$$\begin{aligned} &TRA(X) = CPD(X, ID) \ \ where \ \ CPD(X, Y) = X^{\circ} \cdot Y \\ &TRA(X) = X^{\circ} \cdot ID \\ &TRA(X) = X^{\circ} \end{aligned} \tag{5}$$

Let us now extrapolate this definition into the RMA system. Take the *jcode* relation bitmap from *Jobs* table and the relation that represents the *Identity* matrix with N=3, as shown in Table 21.

jcode	c1	c2	c3
GL	0	0	1
PR	1	0	0
SA	0	1	0

id3	c1	c2	c3
l1	1	0	0
l2	0	1	0
l3	0	0	1

Table 21: *jcode* bitmap from *Jobs* relation table and ID with N=3

First, one must create an *Identity* matrix that needs to be compatible with executing the *dot product* operation. Therefore, the *Identity* matrix needs to be defined with N = 3.

```
SELECT * FROM (jcode ON c1,c2,c3) cpd (id3 ON c1,c2,c3);
```

Listing 5.2: Compute the CPD between *jcode* and *ID*

Secondly, if we create the SQL script and execute it directly in RMA, as shown in Listing 5.2, one faces with the first problem mentioned before. Although the transposition of the application part and order part is successful, one part of the contextual information gets lost in the computation of CPD, the application schema, as can be seen in Table 22.

jcode	c1	c1	c1
c1	0	1	0
c2	0	0	1
c3	1	0	0

Table 22: Converse of *jcode* bitmap from *Jobs* relation table

This problem is solved by using *WITH..AS* clause of SQL and renaming the part to the correct application schema.

```
WITH
T1(jcode,GL,PR,SA) as (SELECT *
                       FROM (jcode ON c1,c2,c3) CPD (id3 ON c1,c2,c3,c4,c5));
SELECT * FROM T1;
```

Listing 5.3: Compute the CPD between *jcode* and *ID*

Listing 5.3 shows the proposed solution for defining relational *matrix transpose* and Table 23 is the result achieved when properly resolving the relational *matrix transpose* for the *jcode* relation bitmap of the *Jobs* relation.

$jcode^{\circ}$	GL	PR	SA
c1	0	1	0
c2	0	0	1
c3	1	0	0

Table 23: Converse of *jcode* bitmap from *Jobs* relation table

5.1.3 *Dot product*

The matrix-matrix multiplication, here referred as *dot product*, is an operation that multiplies a matrix with another matrix, using formally dot products. As seen in Section 2.3.1, there are some requirements for this operation to be successful, or in other words, matrices "*need to type*" according to TLA theory.

To define the relational matrix *dot product* in RMA, the use of the relational *matrix transpose* is going to be useful.

Given two relations *X* and *Y*, the relational matrix multiplication of *X* with *Y* can be obtained in RMA as follows:

$$MMU(X,Y) = CPD(CPD(X,ID),Y) \tag{6}$$

Once the equality in Equation (6) is expanded, one can see how from the relational matrix cross-product, the relational matrix multiplication operation is obtained:

$$\begin{aligned} &MMU(X,Y) = CPD(CPD(X,ID),Y) \;\; where \;\; TRA(X) = CPD(X,ID) = X^{\circ} \\ &MMU(X,Y) = CPD(X^{\circ},Y) \qquad\qquad where \;\; CPD(X,Y) = X^{\circ} \cdot Y \\ &MMU(X,Y) = X \cdot Y \end{aligned} \tag{7}$$

Let us now extrapolate this definition into the RMA system. Take the $jcode^{\circ}$ relation bitmap in Table 23 obtained from using the newly defined *TRA* operations on relation *jcode*.

Listing 5.4 shows the SQL script that is executed in RMA. Note that *T1* is the query result that comes from executing the SQL script in Listing 5.2.

$jcode^{\circ}$	GL	PR	SA
c1	0	1	0
c2	0	0	1
c3	1	0	0

ejob	c1	c2	c3	c4	c5
GL	0	0	1	0	0
PR	1	1	0	0	1
SA	0	0	0	1	0

Table 24: *jcode* relation bitmap from *Jobs* relation table and *ejob* relation bitmap from *Employees* relation table

```
WITH
T2 (jcodet,c1,c2,c3) as (SELECT * FROM (T1 ON gl,pr,sa) cpd (id3 ON c1,c2,c3)),
T3                   as (SELECT * FROM (T2 ON c1,c2,c3) cpd (ejob ON c1,c2,c3,c4,
    c5)),
SELECT * FROM T3;
```

Listing 5.4: Compute MMU between $jcode^{\circ}$ and *ejob*

Listing 5.4 shows the proposed solution for defining relational matrix multiplication and Table 25 is the result achieved when properly resolving the relational matrix multiplication in the $jcode^{\circ}$ and *ejob* relational bitmaps.

$jcode^{\circ} \cdot ejob$	c1	c2	c3	c4	c5
c1	1	1	0	0	1
c2	0	0	0	1	0
c3	0	0	1	0	0

Table 25: Converse of *jcode* bitmap from *Jobs* relation table

5.1.4 *Khatri-Rao product*

The Khatri-Rao multiplication between two matrices is an operation that distributes and multiplies each row of the first matrix by all the rows of the second matrix. As seen in Section 2.3.2, there are some requirements for this operation to be successful the same way that relational matrix dot product has requirements.

To define the relational matrix *Khatri-Rao product* in RMA, the use of the relational *CROSS JOIN* and scalar multiplication is going to be useful. There is no need to use any RMA operations to define these new operations, since relacional SQL already provides all needed operations.

Take the $jsalary \cdot jcode^{\circ} \cdot ejob$ relation bitmap in Table 26, obtained from using the newly defined *TRA* and *MMU* operations on relations *jcode*, *ejob* and *jsalary*. Take as well the *Identity matrix* relation defined with N=5, also in Table 26.

$jsalary \cdot jcode^{\circ} \cdot ejob$	c1	c2	c3	c4	c5
1	1000	1000	1333	1100	1000

id5	c1	c2	c3	c4	c5
l1	1	0	0	0	0
l2	0	1	0	0	0
l3	0	0	1	0	0
l4	0	0	0	1	0
l5	0	0	0	0	1

Table 26: $jsalary \cdot jcode^{\circ} \cdot ejob$ relation bitmap and *Identity matrix* with size N=5

Listing 5.5 shows the SQL script that is executed in RMA. For purposes of readability, relational bitmap $jsalary \cdot jcode^{\circ} \cdot ejob$ is going to be referred as *V*.

```
SELECT c1*c1,c2*c2,c3*c3,c4*c4,c5*c5 FROM V,id5;
```

Listing 5.5: Compute KR between *V* and *id5*

Listing 5.5 shows the proposed solution for defining relational matrix *Khatri-Rao product* and Table 27 is the produced result.

$V \bigtriangledown id5$	c1	c2	c3	c4	c5
l1	1000	0	0	0	0
l2	0	1000	0	0	0
l3	0	0	1333	0	0
l4	0	0	0	1100	0
l5	0	0	0	0	1000

Table 27: Converse of *jcode* bitmap from *Jobs* relation table

An important observation is that relational matrix Khatri-Rao can be used to transform a vector in a diagonalized matrix, as seen in the previous example.

5.1.5 *Hadamard product*

The *Hadamard product* between two matrices is an operation that multiplies each element of the first matrix by the correspondent element of the second matrix, as seen in Section 2.3.3.

To define the relational matrix *Hadamard product* in RMA, the use of the relational *JOIN* and scalar multiplication is going to be useful. As it happens in defining relational matrix *Khatri-Rao product*, here too there is no need to use any RMA operations to define this new operation, since relational SQL already provides all needed operations.

Take, for example, the *A* and *B* relations in Table 28.

A	c1	c2	c3
l1	2	3	1
l2	1	5	2
l3	3	4	1

B	c1	c2	c3
l1	4	1	5
l2	2	3	1
l3	6	1	3

Table 28: Example relations A and B

Listing 5.6 shows the SQL script that is executed in RMA.

```
SELECT a.c1*b.c1,a.c2*b.c2,a.c3*b.c3 FROM A JOIN C ON a.A=b.B;
```

Listing 5.6: Compute Hadamard product between A and B

Listing 5.6 shows the proposed solution to define the relational matrix *Hadamard product* and Table 29 is the produced result.

$A \times B$	c1	c2	c3
l1	8	3	5
l2	2	15	2
l3	18	4	3

Table 29: Hadamard product between relations A and B

5.2 A QUERY EXAMPLE

Back to the query example shown in Section 2.4, one can now think of implementing it on top of RMA equipped with the newly created operations.

The following query should run in RMA: get the total monthly salary per country/branch, ordering the result by countries. This query is in a TLA script shown below 8 that is extracted from the type diagram shown in Figure 20.

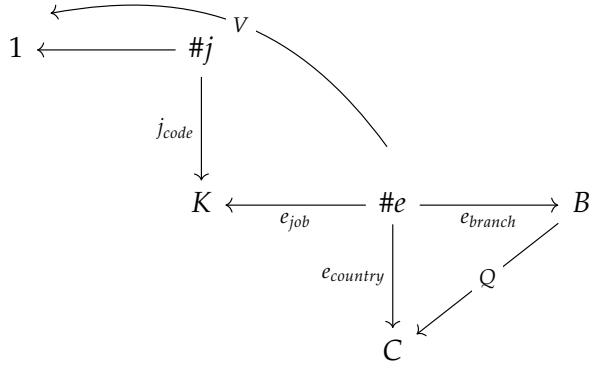

Figure 20: Query type diagram

$$Q = e_{country} \,.\, (V \bigtriangledown ID) \,.\, e^{\circ}{}_{branch} \quad where\ V = j^{salary} \,.\, j^{\circ}{}_{code} \,.\, e_{job} \tag{8}$$

Listing 5.7 shows the equivalent *SQL* query linked to the TLA script.

```
SELECT e_branch, e_country, sum(j_salary)
FROM empl, jobs
WHERE j_code = e_job
GROUP BY e_country, e_branch
ORDER BY e_country;
```

Listing 5.7: *Jobs* and *Employees* SQL query example

Database bitmaps

It should be stressed that TLA theory works with dimensions represented by bitmap matrices. This notion is ported to RMA by working on relations that are bitmaps, as previously shown when defining new RMA operations.

Tables 30 and 31 illustrate all the required relational bitmaps to solve this specific query, from the *Jobs* and *Employees* relations, respectively.

jcode	c1	c2	c3
GL	0	0	1
PR	1	0	0
SA	0	1	0

jsalary	c1	c2	c3
typeone	1000	1100	1333

Table 30: Relational bitmaps tables from *Jobs* relation needed to solve the query

ejob	c1	c2	c3	c4	c5
GL	0	0	1	0	0
PR	1	1	0	0	1
SA	0	0	0	1	0

ebranch	c1	c2	c3	c4	c5
Mobile	1	0	1	0	0
Web	0	1	0	1	1

ecountry	c1	c2	c3	c4	c5
PT	0	0	0	1	1
UK	1	1	1	0	0

Table 31: Relational bitmaps tables from *Employees* relation needed to solve the query

Solving the TLA script step-by-step

As the initial step for solving the TLA script, let us address the *where* clause section:

$$V = j^{salary} \cdot j^{\circ}{}_{code} \cdot e_{job} \tag{9}$$

Using *TRA* on relation bitmap *jcode*,

$jcode^{\circ}$	GL	PR	SA
c1	0	1	0
c2	0	0	1
c3	1	0	0

Table 32: *jcode* relation bitmap from *Jobs* relation table

applying MMU to $jcode^{\circ}$ and *ejob*,

$jcode^{\circ} \cdot ejob$	c1	c2	c3	c4	c5
c1	1	1	0	0	1
c2	0	0	0	1	0
c3	0	0	1	0	0

Table 33: $jcode^{\circ} \cdot ejob$

applying MMU again to *jsalary* and $jcode^{\circ} \cdot ejob$,

$j^{salary} \cdot jcode^{\circ} \cdot ejob$	c1	c2	c3	c4	c5
typeone	1000	1000	1333	1100	1000

Table 34: $j^{salary} \cdot jcode^{\circ} \cdot ejob$

and the first part of the script is now solved.

Advancing now to the main part of the TLA script, first we need to diagonalize the previous table.

To implement the diagonalization, several steps are applied, starting by the *KR* operation,

$V \triangledown ID$	c1	c2	c3	c4	c5
l1	1000	0	0	0	0
l2	0	1000	0	0	0
l3	0	0	1333	0	0
l4	0	0	0	1100	0
l5	0	0	0	0	1000

Table 35: $V \triangledown ID$

then applying TRA to *ebranch*,

$ebranch^{\circ}$	Mobile	Web
l1	1	0
l2	0	1
l3	1	0
l4	0	1
l5	0	1

Table 36: $ebranch^{\circ}$

followed by applying MMU to $(V \bigtriangledown ID)$ with ${e^{\circ}}_{branch}$,

$(V \bigtriangledown ID) \cdot ebranch^{\circ}$	Mobile	Web
l1	1000	0
l2	0	1000
l3	1333	0
l4	0	1100
l5	0	1000

Table 37: $(V \bigtriangledown ID) \cdot ebranch^{\circ}$

and finally by applying MMU to *ecountry* with $(V \bigtriangledown ID) \cdot {e^{\circ}}_{branch}$

$ecountry \cdot (V \bigtriangledown ID) \cdot ebranch^{\circ}$	Mobile	Web
PT	0	2100
UK	2333	1000

Table 38: $ecountry \cdot (V \bigtriangledown ID) \cdot ebranch^{\circ}$

Final SQL query

Listing 5.8 illustrates the SQL script that is executed in RMA.

```
with

t1(jcode,gl,pr,sa) as (SELECT * FROM (jcode ON a1,a2,a3) cpd (id3 ON i1,i2,i3)),

t2(jcodet,a1,a2,a3) as (SELECT * FROM (t1 ON gl,pr,sa) cpd (id3 ON i1,i2,i3)),
t3 as (SELECT * FROM (t2 ON a1,a2,a3) cpd (ejob ON a1,a2,a3,a4,a5)),

t4(jsalary,typeone) as (SELECT * FROM (jsalary ON a1,a2,a3) cpd (id1 ON i1)),
t5 as (SELECT * FROM (t4 ON typeone) cpd (t3 ON a1,a2,a3,a4,a5)),

t6(ebranch,mobile,web) as (SELECT * FROM (ebranch ON a1,a2,a3,a4,a5) cpd (id2 ON
    i1,i2)),

t7(ecountry,a1,a2,a3,a4,a5) as (SELECT ecountry,c1*a1,c2*a2,c3*a3,c4*a4,c5*a5
    FROM ecountry,t5),

t8(ecountry,pt,uk) as (SELECT * FROM (t7 ON a1,a2,a3,a4,a5) cpd (id2 ON i1,i2)),
t9 as (SELECT * FROM (t8 ON pt,uk) cpd (t6 ON mobile,web))

SELECT * FROM t9;
```

Listing 5.8: *Jobs* and *Employees* RMA query example

5.3 SUMMARY

This chapter was devoted to the developing and understanding the creation of new relational matrix operations: relational *matrix transpose*, *TRA*; relational matrix *dot product*, *MMU*; relational matrix *Khatri-Rao product*, *KR*; relational matrix *Hadamard product*, *HM*. After defining all the new operators, the RMA system is now ready to implement new scripts acording to TLA theory. The example explored in Section 2.4 was translated to a RMA script with a step-by-step explanation.

6

CONCLUSIONS

TLA (Chapter 2) is a new approach to querying database systems. In comparison to traditional systems it is a fresh, simple and straightforward way of interacting and querying a database. However, thus far its implementation lacks in performance when compared to MonetDB, as reported by Afonso (2018).

MonetDB (Idreos et al., 2012), a de facto standard in database performance, has thus become a comparison target for this project. It turns out that MonetDB's core system is inherently columnar, due to its data model (Binary Association Tables), and this brings it closer to the already developed LAQ engine. MonetDB exhibits great performance for analytical type queries due to being a highly optimized system tailored to a specific purpose with years of investment in development. The downside, in the author's view, is that it still carries around at frontend level the Relational Model supported by SQL, which is a breakaway from TLA theory and the LAQ engine.

Such conceptual mismatch of the software stack of MonetDB at frontend level has led to the discovery of Relational Matrix Algebra (Dolmatova et al., 2020a,b). As TLA does, RMA emphasizes the use of Linear Algebra operators in analytical querying. As of its first iteration, RMA implements four operations that are integrated and can be computed over MonetDB: relational matrix addition, relational matrix QR decomposition, relational matrix system solving and relational matrix cross-product.

A shortcoming of RMA with respect to TLA is the fact that important linear algebra operators such as matrix multiplication and transpose are not available in the frontend. Nevertheless, the relational matrix cross-product (CPD) is available, from which such operations could be derived and implemented. Matrix converse and matrix composition (i.e. multiplication) are essential to TLA because they allow a "navigation style" in queries, a kind of "follow the path in the type diagram" thinking. Other TLA operations followed the implementation of the previous operations, namely the relational matrix Khatri-Rao product and relational matrix Hadamard-Schur product, which were derived from the relational matrix cross-product operation that already can be found in the RMA system.

The implementation of such new TLA operations in RMA was not a straightforward task. RMA is a new and complex theory expressed in a somewhat convoluted notation. Moreover, RMA is still a prototype which runs on a specific OS subject to specific requirements that need to be understood and met.

Such difficulties delayed the implementation of TLA on top of RMA/MonetDB. The downside of this late success is that it prevented the author from fully automating the translation from TLA scripts to RMA scripts, which are an

extension of standard SQL, let alone benchmarking the whole framework against the TPC-H standard, as initially planned.

6.1 PROJECTED FUTURE WORK

As stated in the title of this dissertation, its main goal is to integrate TLA in the MonetDB stack. Although part of this objective was achieved (implement TLA queries over MonetDB), benchmarking the current implementation using industry standards will be a hard task without developing some process automation beforehand.

The main future work, therefore, revolves around the automation of a life-cycle that is currently done manually. Given a TLA script, this script must be parsed and translated into the corresponding RMA script to be loaded in the RMA system. There is a need to develop a parser for the DSL language defined by Afonso (2018) but this time tuned to the RMA target.

With such DSL parser tailored to LAQ scripting in hand, one can further improve the automation process with shell scripting in order to finally perform benchmarks in a HPC cluster.

BIBLIOGRAPHY

João M. Afonso, Gabriel D. Fernandes, João P. Fernandes, Filipe Oliveira, Bruno M. Ribeiro, Rogério Pontes, José N. Oliveira, and Alberto J. Proença. Typed linear algebra for efficient analytical querying. *CoRR*, abs/1809.00641, 2018. URL http://arxiv.org/abs/1809.00641.

João Afonso. Towards an efficient linear algebra based olap engine. Master's thesis, University of Minho, 2018.

Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, 1997. doi: 10.1145/248603.248616. URL https://doi.org/10.1145/248603.248616.

Anindya Datta and Helen M. Thomas. The cube data model: a conceptual model and algebra for on-line analytical processing in data warehouses. *Decis. Support Syst.*, 27(3):289–301, 1999. doi: 10.1016/S0167-9236(99)00052-4. URL https://doi.org/10.1016/S0167-9236(99)00052-4.

Oksana Dolmatova, Nikolaus Augsten, and Michael H. Böhlen. Preserving contextual information in relational matrix operations. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1894–1897. IEEE, 2020a. doi: 10.1109/ICDE48307.2020.00197. URL https://doi.org/10.1109/ICDE48307.2020.00197.

Oksana Dolmatova, Nikolaus Augsten, and Michael H. Böhlen. A relational matrix algebra and its implementation in a column store. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2573–2587. ACM, 2020b. doi: 10.1145/3318464.3389747. URL https://doi.org/10.1145/3318464.3389747.

Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012. URL http://sites.computer.org/debull/A12mar/monetdb.pdf.

Hugo Daniel Macedo and José Nuno Oliveira. A linear algebra approach to OLAP. *Formal Aspects Comput.*, 27(2):283–307, 2015. doi: 10.1007/s00165-014-0316-9. URL https://doi.org/10.1007/s00165-014-0316-9.

José Nuno Oliveira and Hugo Daniel Macedo. The data cube as a typed linear algebra operator. In Tiark Rompf and Alexander Alexandrov, editors, *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 6:1–6:11. ACM, 2017. doi: 10.1145/3122831.3122834. URL https://doi.org/10.1145/3122831.3122834.

Lucas Pereira and Tiago Baptista. LAQ-OLAP - design of a columnar database system based on linear algebra querying, 2019. LEI Final Report, M.Sc. in Informatics Engineering, University of Minho.

C. Salley and E. F. Codd. Providing OLAP to user-analysts: An it mandate. 1998.

Part I

APPENDICES

A

SUPPORT WORK

A.1 JOBS AND EMPLOYEES SCHEMA SCRIPT

```
START TRANSACTION;

CREATE TABLE "jobs" (
    "j_code"       char(15)     NOT NULL,
    "j_desc"       char(50),
    "j_salary"       decimal(15,2) NOT NULL
);

CREATE TABLE "empl" (
    "e_id"         integer    NOT NULL,
    "e_job"      char(15)     NOT NULL,
    "e_name"       char(15),
    "e_branch"       char(15)     NOT NULL,
    "e_country"      char(15)     NOT NULL
);

INSERT INTO "jobs" values ('PR', 'Programmer', 1000);
INSERT INTO "jobs" values ('SA', 'System Analyst', 1100);
INSERT INTO "jobs" values ('GL', 'Group Leader', 1333);

INSERT INTO "empl" values (1, 'PR', 'Mary', 'Mobile', 'UK');
INSERT INTO "empl" values (2, 'PR', 'John', 'Web', 'UK');
INSERT INTO "empl" values (3, 'GL', 'Charles', 'Mobile', 'UK');
INSERT INTO "empl" values (4, 'SA', 'Ana', 'Web', 'PT');
INSERT INTO "empl" values (5, 'PR', 'Manuel', 'Web', 'PT');

ALTER TABLE "jobs" ADD PRIMARY KEY ("j_code");
ALTER TABLE "empl" ADD PRIMARY KEY ("e_id");
ALTER TABLE "empl" ADD FOREIGN KEY ("e_job")
  REFERENCES "jobs" ("j_code");

COMMIT;
```

B

MAL ALGEBRA COMPENDIUM

B.1 MAL

B.1.1 *Operators*

MAL	Address	Comment
groupby	ALGgroupby	Produces a new BAT with groups indentified by the head column. (The result contains tail times the head value, ie the tail contains the result group sizes.)
find	ALGfind	Returns the index position of a value. If no such BUN exists return OID-nil.
fetch	ALGfetchoid	Returns the value of the BUN at x-th position with 0 <= x < b.count
project	ALGprojecttail	Fill the tail with a constant
projection	ALGprojection	Project left input onto right input.
projection2	ALGprojection2	Project left input onto right inputs which should be consecutive.

B.1.2 *BAT copying*

MAL	Address	Comment
copy	ALGcopy	Returns physical copy of a BAT.
exist	ALGexist	Returns whether 'val' occurs in b.

B.1.3 *Selecting*

The range selections are targeted at the tail of the BAT.

MAL	Address	Comment

Continued on next page

Continued from previous page

MAL	Address	Comment
select	ALGselect1	Select all head values for which the tail value is in range. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value.
select	ALGselect2	Select all head values of the first input BAT for which the tail value is in range and for which the head value occurs in the tail of the second input BAT. The first input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value.
select	ALGselect1nil	With unknown set, each nil != nil
select	ALGselect2nil	With unknown set, each nil != nil
selectNotNil	ALGselectNotNil	Select all not-nil values.
thetaselect	ALGthetaselect1	Select all head values for which the tail value obeys the relation value OP VAL. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.
thetaselect	ALGthetaselect2	Select all head values of the first input BAT for which the tail value obeys the relation value OP VAL and for which the head value occurs in the tail of the second input BAT. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.

B.1.4 *Sort*

MAL	Address	Comment
sort	ALGsort11	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort12	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.

Continued on next page

Continued from previous page

MAL	Address	Comment
sort	ALGsort13	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort21	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort22	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort23	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort31	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort32	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort33	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.

B.1.5 *Unique*

MAL	Address	Comment
unique	ALGunique2	Select all unique values from the tail of the first input. Input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value. The second input BAT is a list of candidates.
unique	ALGunique1	Select all unique values from the tail of the input. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value.

B.1.6 *Join operations*

Crossproduct

MAL	Address	Comment
crossproduct	ALGcrossproduct2	Returns 2 columns with all BUNs, consisting of the head-oids from 'left' and 'right' for which there are BUNs in 'left' and 'right' with equal tails

Joining

MAL	Address	Comment
join	ALGjoin	Join
join	ALGjoin1	Join; only produce left output
leftjoin	ALGleftjoin	Left join with candidate lists
leftjoin	ALGleftjoin1	Left join with candidate lists; only produce left output
outerjoin	ALGouterjoin	Left outer join with candidate lists
semijoin	ALGsemijoin	Semi join with candidate lists
thetajoin	ALGthetajoin	Theta join with candidate lists
bandjoin	ALGbandjoin	Band join: values in l and r match if r - c1 <[=] l <[=] r + c2
rangejoin	ALGrangejoin	Range join: values in l and r1/r2 match if r1 <[=] l <[=] r2
difference	ALGdifference	Difference of l and r with candidate lists
intersect	ALGintersect	Intersection of l and r with candidate lists (i.e. half of semi-join)

B.1.7 *Projection operations*

MAL	Address	Comment
firstn	ALGfirstn	Calculate first N values of B
reuse	ALGreuse	Reuse a temporary BAT if you can. Otherwise, allocate enough storage to accept result of an operation (not involving the heap)
slice	ALGslice$\backslash_{oid}$	Return the slice based on head oid x till y (exclusive).
slice	ALGslice	Return the slice with the BUNs at position x till y
slice	ALGslice$\backslash_{int}$	Return the slice with the BUNs at position x till y
slice	ALGslice$\backslash_{lng}$	Return the slice with the BUNs at position x till y
subslice	ALGsubslice$\backslash_{lng}$	Return the oids of the slice with the BUNs at position x till y

B.1.8 *Common BAT Aggregates*

These operations examine a BAT, and compute some simple aggregate result over it.

MAL	Address	Comment
count	ALGcount$\backslash_{bat}$	Return the current size (in number of elements) in a BAT.
count	ALGcount$\backslash_{nil}$	Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{nils}$==TRUE.
count	ALGcountCND$\backslash_{bat}$	Return the current size (in number of elements) in a BAT.
count	ALGcountCND$\backslash_{nil}$	Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{nils}$==TRUE.
count$_{nonil}$	ALGcount$_{no}\backslash_{nil}$	Return the number of elements currently in a BAT ignoring BUNs with nil-tail
count$_{nonil}$	ALGcountCND$\backslash_{no}\backslash_{nil}$	Return the number of elements currently in a BAT ignoring BUNs with nil-tail

B.1.9 *Default Min and Max*

Implementations a generic Min and Max routines get declared first. The @emph{min()} and @emph{max()} routines below catch any tail-type. The type-specific routines defined later are faster, and will override these any implementations.

- **cardinality** - ALGcard
- **min** - ALGminany, ALGminany$_{skipnil}$
- **max** - ALGmaxany, ALGmaxany$_{skipnil}$
- **avg** - CMDcalcavg

B.1.10 *Standard deviation*

The standard deviation of a set is the square root of its variance. The variance is the sum of squares of the deviation of each value in the set from the mean (average) value, divided by the population of the set.

- **stdeb** - ALGstdev
- **stdevp** - ALGstdevp
- **variance** - ALGvariance

- **variancep** - ALGvariancep
- **covariance** - ALGcovariance
- **covariancep** - ALGcovariancep
- **corr** - ALGcorr

INESCTEC Grant AE2021-0049