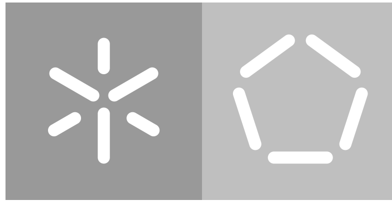
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Miguel Afonso

Towards an Efficient OLAP Engine based on Linear Algebra

November 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Miguel Afonso

Towards an Efficient OLAP Engine based on Linear Algebra

Master Dissertation
Master Degree in Computer Science

Dissertation supervised by
Alberto José Proença
José Nuno Oliveira

November 2018

ACKNOWLEDGEMENTS

This work was financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of project «UID/EEA/50014/2013».

I would like to express my deep gratitude to Professor Alberto Proença and Professor José Oliveira, my research supervisors, for their patient guidance, enthusiastic encouragement and useful critiques of this work.

To Bruno Ribeiro, João Fernandes, Gabriel Fernandes, Fernanda Alves, Filipe Oliveira, and Rogério Pontes, for their previous contributions in the project. More specially to João, with whom I worked together in (Afonso and Fernandes, 2017), and to Bruno, for always contributing with new ideas to implement the vision we all worked for.

Finally, to my family, whose continuous support has been essential to overcome the multiple obstacles I found during the way.

ABSTRACT

Relational database engines associated to the widely used Structured Query Language (SQL) are suffering unsatisfactory performance results in complex business queries, due to ever increasing volumes of stored data. To retrieve and process data in a more efficient way, Online Analytical Processing (OLAP) models have been proposed with an increased focus on attributes (measures and dimensions) over records.

OLAP is based on a row-oriented theory, while a columnar-oriented theory could considerably improve the performance of analytical systems. The Typed Linear Algebra (TLA) approach is an example of such theory: it encodes each database attribute in a distinct matrix. These matrices are combined in a single Linear Algebra (LA) expression to obtain the result of a query.

This dissertation combines concepts of relational databases, OLAP, TLA and performance engineering to design, implement and validate an efficient TLA-DB engine: SQL queries are converted into its equivalent LA expression, using Type Diagrams (TDs), which represent each matrix as an arrow pointing from the number of columns to the number of rows, TDs are converted to a LA expression encoded in Linear Algebra Query language (LAQ) and the LAQ script of a query is automatically coded in C Plus Plus (C++).

An efficient TLA-DB engine required the encoding of the sparse matrices in an adequate format, namely Compressed Sparse Column (CSC), while the operations specified in LAQ expressions had their performance improved by optimised algorithms and an optimised query processor.

The functionality of the resulting LAQ engine was validated with several TPC Benchmark H (TPC-H) queries for various dataset sizes. A comparative evaluation of the TLA-DB with two popular Database Management Systems (DBMSs), PostgreSQL and MySQL, showed that the developed framework outperforms both DBMSs in most TPC-H queries.

RESUMO

As melhorias de desempenho dos sistemas de gestão de bases de dados relacionais não têm sido suficientes para acompanhar o crescimento do volume de dados com que são utilizados. Para colmatar a consequente necessidade de soluções mais eficientes, a teoria OLAP foi proposta. Esta introduz as noções de medidas e dimensões, guardando pré-agregações das medidas baseadas nas últimas, de forma a acelerar o processo de análise de dados.

Contudo, ainda que com regras mais restritas, o OLAP está assente em álgebra relacional. A proposição de uma teoria orientada à coluna pode abrir portas a grandes melhorias de desempenho em consultas analíticas. A álgebra linear tipada é um bom exemplo. Segundo esta teoria, cada um dos atributos é convertido numa matriz independente, as quais são posteriormente combinadas através de uma expressão de álgebra linear que define o resultado da consulta.

Esta dissertação combina conceitos de bases de dados relacionais, OLAP, álgebra linear, teoria de tipos, e computação eficiente para projetar, implementar e validar um motor OLAP robusto e eficiente. Para tal, consultas em SQL são convertidas para a expressão de álgebra linear equivalente, usando diagramas de tipo que representam cada matriz como uma seta a apontar do número de colunas para o número de linhas da matriz. A expressão que deles resulta é então codificada em LAQ e automaticamente implementada em C++.

Para garantir a eficiencia da ferramenta desenvolvida, todas as matrizes foram guardadas num formato adequado, nomeadamente o CSC. Por sua vez, as operações especificadas na LAQ foram implementadas recorrendo a algoritmos optimizados.

A correção do sistema implementado foi garantida através da validação dos resultados de um grupo de consultas extraidas do TPC-H, executadas sobre bases de dados de multiplos tamanhos. Finalmente, a comparação com dois sistemas de bases de dados convencionais (o PostgreSQL e o MySQL) nas métricas de tempo de execução e memória utilizada, demonstrou a maior eficiencia da ferramenta desenvolvida na maioria das consultas.

CONTENTS

1 INTRODUCTION 1
1.1 The Relational Model 2
1.1.1 Relational Algebra 3
1.1.2 SQL 3
1.2 Database System Implementation 6
1.2.1 Query Compilation 6
1.2.2 Query Execution 9
1.3 Data Warehousing 10
1.3.1 Data Modeling 11
1.4 Challenges & Goals 12
1.5 Contribution 13
1.6 Dissertation Outline 13
2 TYPED LINEAR ALGEBRA FOR OLAP 14
2.1 Linear Algebraic Encoding of Data 14
2.1.1 Dense Vectors 14
2.1.2 Sparse Matrices 15
2.2 Type Diagrams 15
2.3 Linear Algebra Query language 16
2.3.1 Dot Product 17
2.3.2 Khatri-Rao Product 18
2.3.3 Hadamard-Schur Product 18
2.3.4 Filter 19
2.3.5 Fold 20
2.3.6 Lift 20
2.4 Conversion algorithm 21
2.4.1 The approach 21
2.5 Conversion Example 27
2.6 Summary 32
3 A TLA-DB ENGINE FOR RELATIONAL SQL 33
3.1 Matrix Representation 33
3.1.1 LIL 34
3.1.2 COO 35
3.1.3 CSC/CSR 35
3.1.4 Matrix labels 38

3.2 LA Operators 38
3.2.1 Hadamard-Schur Product 39
3.2.2 Khatri-Rao Product 39
3.2.3 Dot Product 43
3.2.4 Filter 44
3.2.5 Fold 45
3.2.6 Lift 46
3.3 "Streaming" approach 46
3.3.1 Dependencies in query processing 46
3.3.2 Execution order 47
3.4 SQL Driver 48
3.4.1 SQL Parser 48
3.4.2 Query Rewriter 49
3.4.3 SQL Converter 49
3.4.4 toString 50
3.5 LAQ Engine 50
3.5.1 LAQ Parser 50
3.5.2 Query Optimiser 51
3.5.3 Query Processor 51
3.5.4 Run-time Compiler 54
3.5.5 Query Execution 54
3.6 The framework manager 55
3.7 Summary 55
4 VALIDATION AND PERFORMANCE RESULTS 56
4.1 TPC Benchmark H 56
4.1.1 Benchmark modifications 57
4.2 Testbed environment 58
4.3 Results and discussion 59
4.4 Summary 63
5 CONCLUSIONS 64
5.1 Future work 66
5.1.1 Framework extensions 66
5.1.2 Horizontal scalability 70
5.1.3 Incremental querying 72

A TPC-H QUERIES - SQL AND LAQ VERSIONS 76
A.1 Query 3 76
A.2 Query 4 77

A.3 Query 6 78
A.4 Query 11 78
A.5 Query 12 79
A.6 Query 14 80

LIST OF FIGURES

Figure 1	TPC-H Query 3 – Relational Algebra representation	7
Figure 2	TPC-H Query 3 – RA optimized representation	8
Figure 3	System architecture	33
Figure 4	Detailed system architecture	34
Figure 5	A sparse matrix in CSC format	36
Figure 6	TPC-H query 6 - execution plan	48
Figure 7	TPC-H database schema	57
Figure 8	TPC-H queries 3 and 6 – preliminary performance results	59
Figure 9	TPC-H queries 3 and 4 performance results	60
Figure 10	Execution times (scale factor: 2^5)	61
Figure 11	Sequential and Parallel Query 6	62
Figure 12	Memory usage (scale factor: 2^5)	62
Figure 13	TPC-H queries 3 and 4 performance results (MonetDB)	69
Figure 14	Execution times (scale factor: 2^5)	69
Figure 15	Memory usage (scale factor: 2^5)	69
Figure 16	Distributed dot product - replicate A	70
Figure 17	Distributed dot product - rotate A	71

LIST OF TABLES

Table 1 Relational Model terminology 2
Table 2 TPC-H relation Lineitem 2
Table 3 TPC-H relation Orders 2
Table 4 Relational Algebra operators 4
Table 5 OLTP *vs* Data Warehousing systems 11
Table 6 Properties of CSC block types 37
Table 7 Testbed environment 59

ACRONYMS

A

AL Aggregations List. 4
API Application Programming Interface. 52, 55

B

BI Business Intelligence. 1

C

C++ C Plus Plus. ii, iii, 13, 51, 54, 55, 65
CBLAS C language Basic Linear Algebra Subprograms. 38, 45, 64
CFG Context-Free Grammar. 21, 50, 68
COO Coordinate List. 34–36, 64, 67
CRUD Create Read Update Delete. 10
CSC Compressed Sparse Column. ii, iii, 34–36, 40, 42, 43, 46, 64, 67
CSR Compressed Sparse Row. 34, 35, 64
CSV comma-separated values. 34

D

DBMS Database Management System. ii, 1, 6, 12, 13, 38, 48, 54, 56, 58, 60, 62, 65–67
DCL Data Control Language. 3
DDL Data Definition Language. 3
DML Data Manipulation Language. 3
DQL Data Query Language. 3
DSL Domain Specific Language. 14, 16, 32, 64
DTL Data Transaction Language. 3
DW Data Warehouse. 11

G

GA Grouping Attributes. 4
GMP GNU Multi Precision. 65

I

IDC International Data Corporation. 1
Intel MKL Intel Math Kernel Library. 38, 45, 64

L

LA Linear Algebra. ii, 12–14, 16, 21, 27, 31–33, 38, 51, 55, 57, 58, 60–67, 69, 70
LAQ Linear Algebra Query language. ii, iii, 13, 16, 21, 27, 31, 33, 37, 38, 45–47, 49–53, 55, 58, 60, 64–66, 68, 70, 72
LIL List of Lists. 34, 35, 64

O

OLAP Online Analytical Processing. ii, iii, 11, 12, 16, 21, 32, 45, 56, 65, 68
OLTP Online Transaction Processing. 10, 11
OS Operating System. 67

P

PU Processing Unit. 59, 61

R

RA Relational Algebra. 3, 5, 6, 12, 21, 22
RAM Random Access Memory. 58, 59, 62, 63
RC Relational Calculus. 3
RDBMS Relational Database Management System. 2, 33, 49, 69
RM Relational Model. 1, 2

S

SQL Structured Query Language. ii, iii, 1, 3, 5, 6, 9, 12–14, 16, 20, 21, 23, 26–28, 32, 33, 37, 45, 48–50, 53, 55, 56, 58, 65, 66, 68

T

TD Type Diagram. ii, 15–17, 19, 20, 23, 28, 32, 49, 65, 68
TLA Typed Linear Algebra. ii, 13, 14, 32, 49
TPC Transaction Processing Performance Council. 56
TPC-DS TPC Benchmark DS. 56
TPC-H TPC Benchmark H. ii, iii, 2, 21, 47, 50, 56, 57, 59, 60, 63, 65, 68, 72
TSV tab-separated values. 34

1

INTRODUCTION

"From paper-based files to the electronic era, there is not one aspect of modern business that has avoided the need to collect, collate, organise and report upon data" Lake and Crowther (2013).

The importance of data in the business context is unquestionable, and the amount of electronic data that companies have in hand is growing exponentially, developing the ideal environment for high profits.

However, a study developed by the International Data Corporation (IDC) (Gantz and Reinsel, 2012) revealed that only 0.5% of this data is actually analysed. This slight percentage reveals that the improvements in data processing systems and methodologies are not enough to compete with the data growing rate.

In fact, the development of a DBMS is certainly among the most complex projects in computer science. Not only all the theoretical basis must be carefully and extensively researched, but all the system components must be efficiently implemented and tested under a wide range of queries and datasets.

It is true that many recent systems have been built for more special purpose tasks, slightly bypassing the establishment of a formal basis. But is also true that most of these systems have failed to compete with the standard solutions.

The best example is probably Map-Reduce. Purposely built in 2005 to support Google's crawl database, it was replaced a few years later by Big Table, and only now the world is realising that even a software built by Google can have no practical applicability (Stonebreaker et al., 2015, p. 4).

Still in the same mindset, the basis of all major system is, or is prone to become, relational SQL (Stonebreaker et al., 2015, p. 5). Considering that the relational theory has been on the market for almost fifty years, and is still the standard approach to databases is the proof that formal theories tend to support the development of better software.

The next sections introduce the reader the key issues in the evolution of database systems, from Codd's Relational Model (RM) to modern implementations considering Business Intelligence (BI) methodologies. This context is relevant to better comprehend the core topics addressed by this work.

1.1 THE RELATIONAL MODEL

In his paper "*A Relational Model of Data for Large Shared Data Banks*" (Codd, 1970), Edgar Codd proposed the database model which would become the theoretical foundation of all Relational Database Management Systems (RDBMSs) (Connolly and Begg, 2014, p. 149).

As a result of the successive adjustments in these systems and the subsequent deviations from the original model, the terminology of the RM is quite confusing. Some concepts have multiple designations, so a short summary is presented in Table 1.

Formal Terms	Alternative 1	Alternative 2
Relation	Table	File
Tuple	Row	Record
Attribute	Column	Field

Table 1.: Relational Model terminology. Based on Connolly and Begg (2014, p.154)

As the name suggests, the RM is based on the mathematical concept of relation. Relations serve to keep all the information stored in the database. They are represented as bi-dimensional tables in which the rows of the table correspond to individual records (tuples) and the table columns correspond to attributes (Connolly and Begg, 2014, p. 152).

For example, the Tables 2 and 3 respectively depict the relations *Lineitem* and *Orders*. Analyzing the tables, one can identify that the relation *Lineitem* is composed by the attributes *Quantity* and *OrderKey*, while the relation *Orders* incorporates the attributes *OrderKey*, *OrderDate* and *ShipPriority*. Furthermore, the **cardinality** (number of records) of both relations is the same.

Lineitem	
Quantity	OrderKey
28	32
44	32
13	34

Table 2.: TPC-H relation lineitem [1]

Orders		
OrderKey	OrderDate	ShipPriority
32	1995-07-16	1-URGENT
33	1993-10-27	2-HIGH
34	1998-07-21	1-URGENT

Table 3.: TPC-H relation orders [1]

The RM states that there can be no duplicate tuples within a relation. To ensure this property, it is necessary to properly identify each tuple of the relation with a **primary key**. This key is an attribute (or set of attributes) from the relation where all the records are distinct.

1 To maintain the document consistency, all the database examples used in this dissertation are based on TPC-H. This benchmark will be described in section 4.1.

When an attribute exists in more than one relation, it usually represents a relationship between tuples of the two relations (Connolly and Begg, 2014, p. 159). For example, the inclusion of *OrderKey* in both *Orders* and *Lineitem* relations connects each order to the items that compose it. In the Orders relation, *OrderKey* is the primary key. However, in the *Lineitem* relation, the *OrderKey* attribute is a **foreign key**.

These keys are extremely important because they are the only way of relating information spread across multiple tables. The law of referential integrity specifies that every non-null value in a foreign key must match a primary key value of some tuple in its home relation (Connolly and Begg, 2014, p. 162).

1.1.1 *Relational Algebra*

More than the definition of a structure for the database and its data, any database model must specify a set of operations on the information stored in the database (Connolly and Begg, 2014, p. 93). To fulfill this purpose, Codd (1971) introduced Relational Algebra (RA) and the Relational Calculus (RC).

As defined by Connolly and Begg (2014, p. 168), "*The relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation(s)*". This type similarity between the input and output of relational expressions allows their composition in an analogous way to arithmetic operations.

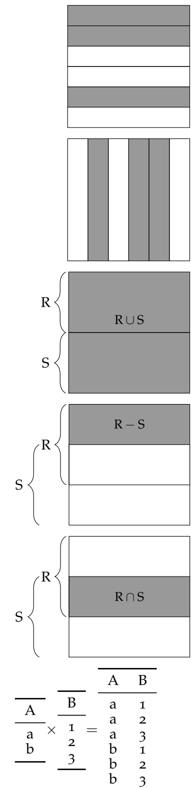
There are five key operations in RA: selection, projection, Cartesian product, union and set difference. Based on them, other operations have been defined, for example the join and intersection, Connolly and Begg (2014, p. 168).

Both RA and RC are too broad to be exhaustively described in this document. Table 4 contains a summary of the enumerated operations, based on Connolly and Begg (2014, pp. 169–180). Please consult Connolly and Begg (2014) and Molina et al. (2008) for further information.

1.1.2 *SQL*

RA is a mathematical language, difficult to use in database manipulation tasks. To achieve a higher level of abstraction and facilitate this tasks, other RA based languages have been proposed, being SQL one of them.

The SQL language can be divided in five major groups: Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), Data Control Language (DCL), and Data Transaction Language (DTL). The present analysis will be summarised and restricted to the DQL, more specifically to the `SELECT` statement.



Selection $(\sigma_{\text{predicate}}(R))$ - an unary operation with a single relation (R) that produces a relation containing only those tuples of R that satisfy the specified condition (predicate).

Projection $(\pi_{a1,\ldots,an}(R))$ - an unary operation that produces a relation containing a vertical subset of R with the specified attributes and eliminating duplicate tuples.

Union $(R \cup S)$ - a binary operation that produces a relation containing all tuples of R and S, without duplicate tuples. R and S must be union-compatible.

Set difference $(R - F)$ - produces a relation with all tuples in R that do not exist in S. R and S must be union-compatible.

Intersection $(R \cap S)$ - produces a relation containing all tuples present in both R and S. R and S must be union-compatible.

Cartesian product (Cross Join) $(R \times S)$ - produces a relation that is the concatenation of every tuple of relation R with every tuple of relation S.

Theta join $(R \bowtie_F S)$ - produces a relation that contains tuples satisfying the predicate F from the Cartesian product of R and S, that is, $R \bowtie_F S = \sigma_F(R \times S)$. The resultant predicate is of the type $R_{\text{attribute}}\,\theta\,S_{\text{attribute}}$, where θ must be a comparison operator: $<, >, \leq, \geq, =, \neq$

Equijoin $(R \bowtie_F S)$ - similar to theta join, produces a relation that contains tuples satisfying predicate F, while restricting the comparison operators to equality, $R_{\text{attribute}} = S_{\text{attribute}}$.

Natural join $(R \bowtie S)$ - equivalent to an equijoin of two relations over all common attributes. Only a single occurrence of each attribute is kept. As illustrated, the natural join of the Tables 2 and 3 with abbreviated attribute names.

Lineitem ⋈ Orders

Q	OK	OD	SP
28	32	1995-07-16	1-URGENT
44	32	1995-07-16	1-URGENT
13	34	1998-07-21	1-URGENT

(Right) Outer join $(R ⟖ S)$ - a natural join where tuples from S that do not match any values in the common attributes of R are also included in the output relation. The presented table contains the right outer join of the Tables 2 and 3 (Lineitem and Orders).

Lineitem ⟖ Orders

Q	OK	OD	SP
28	32	1995-07-16	1-URGENT
44	32	1995-07-16	1-URGENT
	33	1993-10-27	1-URGENT
13	34	1998-07-21	1-URGENT

Semijoin $(R \triangleright_F S)$ - operates on the relations R and S, producing a relation that includes only attributes from R. The obtained tuples are the ones in R that participate in the join of R with S, satisfying predicate F. This way, $R \triangleright_F S = \pi_A(R \bowtie_F S)$

Lineitem ▷ Orders

Quantity	OrderKey
28	32
44	32
13	34

Aggregate $(_{AL}(R))$ - applies an Aggregations List (AL) to the relation R, defining a new relation with the results of the aggregations. AL contains one or more pairs of attributes and aggregate functions, e.g., SUM, COUNT, AVG, MIN and MAX.

Grouping $(\gamma_{GA\ AL}(R)$, as in Molina et al. (2008, p. 778)) - groups the tuples of R by the Grouping Attributes (GA). The values in each created group are then combined with the aggregation operators specified in AL. The output relation contains the GA, along with the results of all the aggregate functions.

Table 4.: Relational Algebra operators

The SELECT statement aims to retrieve data from the database, being the most used SQL command. As explained by Connolly and Begg (2014, p. 197), the basic structure of this instruction is the following:

```
SELECT     [DISTINCT | ALL] {* | [columnExpression [AS newName]] [,...]}
FROM       TableName [alias] [,...]
[WHERE     condition]
[GROUP BY  columnList] [HAVING condition]
[ORDER BY  columnList]
```

It has two mandatory clauses: SELECT and FROM and three optional ones: WHERE, GROUP BY and ORDER BY.

The SELECT statement is similar to the relational projection, although the selected attributes are kept intact, that is, the repeated tuples are not removed. The attributes contained in this clause can be involved by an aggregation function.

The FROM clause is the simplest one, because it just enumerates the tables that will be used in the query. Using the presented syntax the table joins are always implicit, avoiding the inclusion of multiple statements : [INNER | [LEFT | RIGHT | FULL] [OUTER]] JOIN. All of them can be recreated by combining implicit joins and WHERE clauses.

The WHERE statement corresponds to the RA selection operation, being used to restrict the rows to be retrieved. It must be followed by a condition (or predicate), used to filter the desired records. Connolly and Begg (2014, p. 201) specifies five basic conditions:

1. The direct **comparison** of two expressions, e.g. $a \geq b$;
2. The value of an expression is within a defined **range**, e.g. $a\ BETWEEN(x, y)$;
3. The value of an expression equals an element in a **set**, e.g. $a\ IN(x, y, ...)$, where $x, y, ...$ can be tuples with one or more elements;
4. The value of an expression matches a predefined **pattern**, e.g. $a\ LIKE\ \text{"}pattern\text{"}$
5. The value is **null**.

Corresponding to the relational grouping operator, the GROUP BY clause can be used to calculate sub-aggregations of data, based on the similarity of records extracted from a specified set of columns.

However, for the SQL query to be valid, the GROUP BY and the SELECT clauses must be integrated under some rules. For example, all attributes in the SELECT statement must either be present in the GROUP BY clause or be involved by an aggregation function (Connolly and Begg, 2014, p. 210).

The ORDER BY clause is self explanatory, simply sorting the records based on the specified attributes.

Although the presented structure of the SELECT only represents a narrow view of the command, it can be used to describe most of the SQL queries.

1.2 DATABASE SYSTEM IMPLEMENTATION

One of the main goals when designing user-friendly applications is to guarantee short response times. In the database environment, this translates to a demand for real-time query processing of always increasing volumes of data.

To pursue this objective, DBMS developers had to consider performance in high priority, optimizing all critical components of a system. However, considering that the algorithms to be executed are strongly tied to each SQL query, a less efficient implementation can lead to hours or even days to complete a query. To overcome this limitation, a new module was added to the SQL compiler: a query optimiser.

1.2.1 *Query Compilation*

The process of compiling a query encompasses three major phases: the query parsing, the optimising phase and the code generation. While the code generation is a simple translation of an execution plan to machine code, the other two phases are more complex, deserving a more in-depth analysis.

Query parsing

Like the parser of many other programming languages, the only role of the SQL parser is to receive queries in a textual format and convert them to a parsing tree. Although it seems a trivial task, the complexity inherent to the language raises some obstacles, and thus leads to some deviations from the standards in the majority of the implemented systems.

Still in the parsing context, the generated parsing tree will then be scanned by the preprocessor, which compares it to the data dictionary of the database. This way, it ensures that all the tables and attributes used in the query are valid and used correctly, and that the user has enough permissions to consult them.

Considering the similarity between SQL and RA, the conversion between both notations is quite straightforward. For instance, Listing 1 represents an example SQL query and Figure 1 its translation to RA.

```sql
SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'MACHINERY'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < '1995-03-10'
    AND l_shipdate > '1995-03-10'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority;
```

Listing 1: TPC-H Query 3 [Adapted]

The `FROM` clause specifies three tables: *Customer*, *Orders*, and *Lineitem*. Since the query has no explicit join statement, the tables have to be combined using the Cartesian product. After that, the necessary selections and groupings are applied, to finally do the projection as stated in the `SELECT` clause.

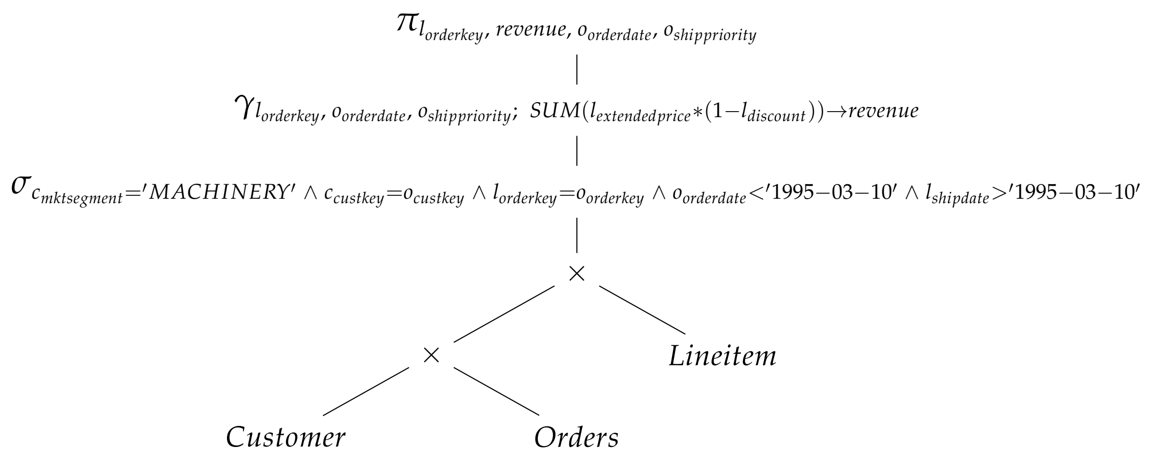


Figure 1.: TPC-H Query 3 – Relational Algebra representation [3]

3 Note the usage of t_{attr} notation to represent the attribute *Attr* from the table *Table*, where "*t*" is the first letter of the table name

Query optimization

As shown in Figure 1, two Cartesian products must be calculated before any relation has been filtered, either by selection or grouping. Thus, it is necessary to calculate a massive intermediate table, certainly incomputable even for medium-sized databases.

It is the role of the query optimizer to overcome situations like this. To do so, it counts with a rule-based system for query rewriting, complemented with a cost-based estimator to choose the most efficient execution path.

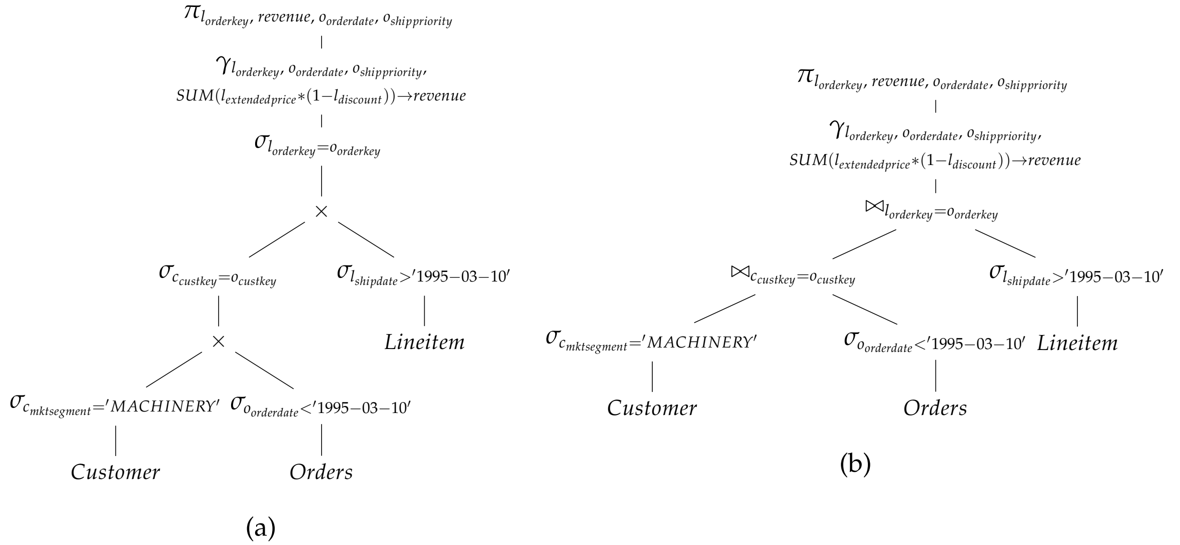


Figure 2.: TPC-H Query 3 – RA optimized representation

Figure 2 shows two possible optimizations over the parsing tree in Figure 1. Considering the diagram (a), the selection operator have been spread across the diagram by applying the commutative property on the selection and the Cartesian product: $\sigma_{\text{predicate}}(\text{R} \times \text{S}) = (\sigma_{\text{predicate}}(\text{R})) \times \text{S}$. This change to the query is in conformity with the idea that every table should be filtered as soon as possible, thus reducing the data to be processed in later operations.

As explained in Table 4, whenever a selection is applied to the Cartesian product of two relations, it can be replaced by a theta-join, $\sigma_{\text{predicate}}(\text{R} \times \text{S}) = \text{R} \bowtie_{predicate} \text{S}$. By employing this property the diagram in (a) can be converted to the one in (b). This optimization removed the need to compute the Cartesian product.

These two optimizations demonstrate how the volume of data should be reduced as early as possible and the relevance of calculating only the unavoidable operations.

After these and other optimizations rules being applied, the query reaches its logical plan state. However, it can still be optimized in many cases. To reach this level of optimization, the properties of each relation and the data they contain should be analyzed.

For instance, consider the parsing tree in Figure 2 (b). It is predictable that the selection applied to the table *Customer* is more restrictive than the ones in the tables *Orders* and *Lineitem*. Also, as it will be explained in Section 4.1, the table *Lineitem* is the one with the highest number of records, while the *Customer* is the shortest among the three. This way it is predictable that if the order of the two joins were changed, for example $(Customer \bowtie Orders) \bowtie Lineitem \Rightarrow Customer \bowtie (Orders \bowtie Lineitem)$, the query would take much longer to complete, since a join between the two largest tables had to be computed.

Additionally, several algorithms can complete the join operations and it is up to the query optimiser to select the most efficient one.

1.2.2 *Query Execution*

SQL operations may have multiple implementations on a single system. For instance, consider the equijoin and natural join operations, the most common join operations. They can be completed using one of three distinct algorithms: the nested loop join; the merge join, and the hash join.

Nested loop join

The simplest version of the nested loop join iterates through all the combinations of tuples from the relations to be joined. Whenever the records of both tables have matching keys, they are combined in a new tuple, which is added to the result table.

As explained in Molina et al. (2008, p. 719), the join $R(X,Y) \bowtie S(Y,Z)$ is processed according tho the pseudocode in Listing 2.

```
1 for each tuple s in S:
2     for each tuple r in R:
3         if r and s join to make a tuple t:
4             output t
```

Listing 2: Nested loop join - pseudo-code

Considering that the tables R and S respectively contain N and M records, the complexity of the algorithm can be defined as $\mathcal{O}(N \times M)$, fitting in the category of the quadratic algorithms.

If none of the relations R and S fits in the main memory, the data access patterns of the algorithm will imply that nearly $N \times M$ records must be loaded from disk, completely annihilating the algorithm performance. One possible solution is to take advantage from the memory hierarchy by using a blocked version of the algorithm, denoted block nested loop join (Connolly and Begg, 2014, p. 752).

Merge join

If it is ensured that the relations to be joined are sorted by the attributes they have in common, for instance, "Y" in the previous example, the process of joining the tables can be completed in linear time. This is achieved by employing the merge algorithm from the merge sort, hence the name of this algorithm.

However, if required sorting property is not guaranteed, a sorting phase must be computed before the merge. These two phases of data processing are the reason why this algorithm belongs to the two-pass category.

Considering the relations R and S from the previous example, as both of them must be sorted and then merged, the total cost of this algorithm can be estimated as $\mathcal{O}(N \times \log N + M \times \log M + N + M)$.

Hash join

The hash join is also a two-pass algorithm. In its first phase, it groups the tuples of both relations in distinct subsets, according to a predefined hash function ("h").

If two relations R and S are respectively scattered among the $R_1, R_2, ..., R_n$ and $S_1, S_2, ..., S_n$ groups, the hash property ensures that a tuple in the group R_x can only be joined with tuples in S_x, that is, if $h(R_{attr}) \neq h(S_{attr})$, then $R_{attr} \neq S_{attr}$. Note that the opposite is not necessarily true, if $h(R_{attr}) = h(S_{attr})$, there is no guarantee that $R_{attr} = S_{attr}$, as the hash function can have the same result for distinct inputs (Connolly and Begg, 2014, p. 754).

The second phase is to iterate over the pairs of R and S partitions, joining them, for example, with a nested loop join. Despite the low efficiency of the nested loop algorithms, since in most of the tuples the join condition will be true, the final relation will have approximately $R_x \times S_x$ tuples, making the algorithm's complexity linear in the output size.

This way, the complexity of the algorithm is $\mathcal{O}(\alpha \times (N + M))$, where $\alpha = 2$ for the two phases of the algorithm or $\alpha = 3$ if the step of writing data back to disk between the two phases is considered.

1.3 DATA WAREHOUSING

Relational systems are well suited to process large volumes of short and simple Create Read Update Delete (CRUD) transactions. Their performance in this so-called Online Transaction Processing (OLTP) is powered by how they store and operate over data. For example, the insertion of a new entry in a database is processed with the construction of a single record and its attachment at the end of the correspondent relation, having little to none interference with the remainder of the database.

An organization will normally have multiple OLTP databases, each one oriented to a distinct branch of the company, for instance storing inventory data or sales data per point-

of-sale (Connolly and Begg, 2014, p. 1227). The operational data in these databases is highly volatile, requiring frequent updates and deletes, which are responsible to keep the database short.

Contrasting with OLTP, typical business queries tend to operate over multiple relations, performing complex data manipulation tasks. These are called OLAP queries.

In the enterprise environment it is interesting to analyze and compare both recent and historical data to obtaining better insights on the business process. This way, data is usually moved from the transactional systems to a central analytical database known as Data Warehouse (DW). This system typically encapsulates all the data of the organization. Since the only common operation in addiction to the data consulting is the refreshment of the database with data coming from the OLTP databases, they are considered non-volatile systems. The distinction between the two presented branches of data storage is summarized in Table 5.

CHARACTERISTIC	OLTP SYSTEMS	DATA WAREHOUSING SYSTEMS
Main purpose	Support operational processing	Support analytical processing
Data age	Current	Historic (but trend is toward also including current data)
Data latency	Real-time	Depends on length of cycle for data supplements to warehouse (but trend is toward real-time supplements)
Data granularity	Detailed data	Detailed data, lightly and highly summarized data
Data processing	Predictable pattern of data manipulation. High level of transaction throughput.	Less predictable pattern of data queries; medium to low level of transaction throughput
Reporting	Predictable, one-dimensional, relatively static fixed reporting	Unpredictable, multidimensional, dynamic reporting
Users	Serves large number of operational users	Serves lower number of managerial users (but trend is also toward supporting analytical requirements of operational users)

Table 5.: OLTP *vs* Data Warehousing systems. Based on Connolly and Begg (2014, p. 1227)

1.3.1 *Data Modeling*

The complete analysis of DW design and implementation methodologies is out of the scope of this dissertation. However, the interested reader can consult Kimball and Ross (2011) and

Inmon et al. (2002) to understand the two main distinct approaches. This document is based on Kimball's method.

Kimball's business dimensional life-cycle defines a dimensional modeling phase aiming to convert data into a standard and intuitive format, where data can be more efficiently accessed (Connolly and Begg, 2014, p. 1261).

Kimball defines two distinct sets of attributes: measures and dimensions. Measures should be contained in a central table (the fact table); they correspond to aggregable data, like monetary values or number of sales. Dimensions are the context that help to understand the meaning of those measures.

As stated, the fact table should be the central part of the database, surrounded by dimension tables to form a star schema (Connolly and Begg, 2014, p. 1261). If dimension tables are linked to other dimension tables, the schema is denoted snowflake.

The concepts presented in this section are the base of OLAP theories. However, the simple differentiation of transactional and analytical queries is enough to support the comprehension of the conducted work. Multidimensional analysis techniques can also be used with the approach presented in Section 2, but its implementation is out of the scope of this dissertation.

1.4 CHALLENGES & GOALS

Macedo and Oliveira (2015) proposed a way to replace RA by LA to encode and resolve OLAP queries. Pontes (2015) tested the efficiency of such theories in a distributed environment built with the Hadoop framework (Shvachko et al., 2010). Considering the unsatisfactory, but promising results, Oliveira and Caldas (2016) implemented a simple query in shared memory, already outperforming PostgreSQL.

To extend the test suite from a single query to a larger, and then more solid group of queries, Ribeiro et al. (2017) worked on the implementation of multiple and more complex queries. Meanwhile, Afonso and Fernandes (2017) provided them the LA scripts of the tested queries, obtained when searching for a generic algorithm to convert a SQL query to LA equivalent.

This dissertation aims the consolidation of all previous work in a single piece of software. The main goals are the development of a DBMS based on the LA approach, as well as its validation with an industry standard benchmark suite and its performance evaluation through a comparison with standard market solutions.

The key challenges the author had to face included:

- the choice of an adequate format to represent and process very large sparse matrices;
- the performance evaluation of the alternative algorithms and implementations to execute the LA operations;

- the selection of a set of representative queries to validate the LA engine;
- the setup of a testbed environment with the competitive DBMS and the consequent reliable performance measurements.

1.5 CONTRIBUTION

This dissertation is a follow-up of the stated projects to build a new integrated framework to aid the efficient use of a LA DBMS. To identify the key contributions is not straightforward task: the major one was certainly the proposal of a modular architecture to address queries encoded in SQL, while the other ones are mostly related to an efficient implementation of these modules. The following contributions should be outlined:

- the complete re-design of the previous kernel of LA operations;
- the development of a code generator that takes LAQ scripts as input and produces the equivalent C++ versions;
- the validation of the code generator with several inputs from a standard benchmark suite;
- the performance tuning of the LAQ approach and a performance evaluation with standard DBMSs.

One of the key outcomes of this work is the paper *''Typed Linear Algebra for Efficient Analytical Querying''* (Afonso et al., 2018) submitted to VLDB. The received reviews gave very relevant clues for a further improvement of the work, which are included and discussed in the Future Work section in the last Chapter of this dissertation.

1.6 DISSERTATION OUTLINE

Chapter 2 presents TLA and how it can be used to power a database systems and encode relational queries. Chapter 3 presents the architecture and implementation details of the developed framework. Chapter 4 specifies how the system was validated and benchmarked, also comparing its performance with other systems. Finally, Chapter 5 contains some considerations on the developed work, as well as relevant information on how the system can be extended.

It is also noteworthy that some parts of the document were extracted from previous publications for which the author has contributed. More specifically, Chapter 2 contains parts of a previous report by Afonso and Fernandes (2017), and Chapter 4 reproduces the analysis made for Afonso et al. (2018), which was carried in parallel with the work being described in this dissertation.

2

TYPED LINEAR ALGEBRA FOR OLAP

As stated by Stonebreaker et al. (2015), the "*SQL Standard is both ambiguous and underspecified*". Despite all its drawbacks, SQL is undoubtedly the most used query language in the market, and the recommended interface for any new implemented system. This way, a LA solution will be introduced as an alternative encoding for relational SQL.

Afonso and Fernandes (2017) developed a novel attempt to define a Domain Specific Language (DSL) to describe the LA operators, derived from SQL queries. Although the proposed solution is not full SQL compliant, it contains an objective introduction to the TLA concepts, forming the basis of the current analysis.

2.1 LINEAR ALGEBRAIC ENCODING OF DATA

Since LA works mainly with matrices, it is crucial to understand how information, initially encapsulated in relations, gets converted into matrices.

The LA encoding must be done separately for each attribute in the database, taking advantage of the columnar data access and making it possible to ignore useless attributes for each computed query.

2.1.1 *Dense Vectors*

The first data representation model is a straightforward extraction of an attribute from the data table. Its representation is a dense vector, as shown in (1). The *ShipPriority* attribute from the *Orders* table will be used and, from now on, named $o_{shippriority}$.

$$o_{shippriority} = \begin{matrix} 1 & 2 & 3 & 4 \\ \end{matrix} \atop \begin{bmatrix} 1 & 2 & 1 & 3 \end{bmatrix} \tag{1}$$

2.1.2 *Sparse Matrices*

With the attribute represented in a dense vector format, it is now necessary to convert it into a matrix. As seen in (2), the matrix contains the same information as the vector (1), being a direct conversion from it. The row labels are the set of distinct values that constitute the previous vector. Obviously, repeated values only need to be represented once. The number of columns of the defined matrix matches the length of the dense vector.

$$o_{shippriority} = \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & - & 1 & - \\ - & 1 & - & - \\ - & - & - & 1 \end{bmatrix} \end{array} \begin{array}{l} \\ \text{1-URGENT} \\ \text{2-HIGH} \\ \text{3-MEDIUM} \end{array} \tag{2}$$

Each value ("1") in the matrix means that the record defined by the respective column number contains the value specified by the matrix row. For instance, a "1" in column "4" at the row labelled "3-MEDIUM" means that the 4th element of the array is a "3-MEDIUM".

This conversion process creates functional matrices composed by, at most, a single value per column, that can be efficiently represented in a sparse format, as accomplished by Ribeiro et al. (2017).

2.2 TYPE DIAGRAMS

The concept of TD is also crucial in this approach. In (3), a diagram representing a single matrix of width $\#o$ and height SP can be seen. It shows the type of matrix (2) that represents the attribute $o_{shippriority}$.

$$SP \xleftarrow{o_{shippriority}} \#o \tag{3}$$

Afonso and Fernandes (2017) also used a standard notation for naming the dimensions of the matrix. The number of records in the table is represented by the character # followed by the first letter of the tables name (in this case, the table is named *Orders*, hence $\#o$).

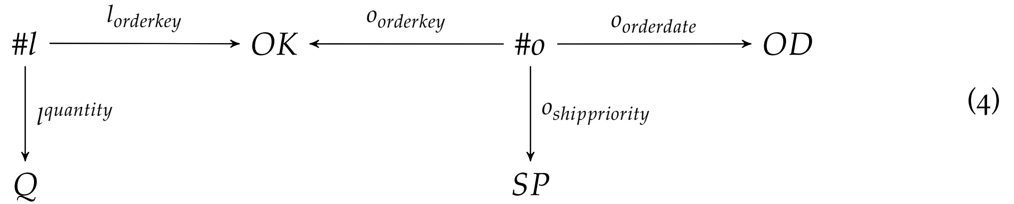
The number of rows in the matrix is the cardinality of type SP, totalising the distinct values in the attribute $o_{shippriority}$. This abbreviation comes from the combination of the first letter of each word in the attribute's name.

It is also relevant to mention that this arrow notation is typically used to represent functions. In this case, it receives an argument of type $\#o$ and returns a value of type SP.

The created matrix does just that. For a given row of the data table ($\#o$) its corresponding column in the matrix identifies its attribute value (SP), by checking the row in which the

"1" is contained in the bitmap. This creates this notion that a matrix can be thought of as a function.

This notation can be scaled and composed to represent a full database schema. In (4) the attributes in Tables 2 and 3 are placed in the equivalent TD.



(4)

The first thing to notice are the two tables. These are characterised in the diagram by all the attributes that compose them. Obviously, attributes of the same table have the same number of registers, hence deriving from the same point in the TD.

Another aspect introduced with this diagram are the joins between tables. In this case, the table *Lineitem* has an attribute ($l_{orderkey}$) that is a foreign key pointing to the primary key of the table *Orders*.

Since null values are not allowed in OLAP databases, every single value of the foreign key attribute ($l_{orderkey}$) must be present in the primary key ($o_{orderkey}$).

This validates the invariant that the cardinality of the foreign key is lower or equal to the one of the primary key.

As this representation abstracts the table's data, the bitmaps containing the foreign keys will have as many rows as the primary key, culminating in the same dimension in the diagram (*OK*).

Finally, note how in $l^{quantity}$ the attribute name was intentionally positioned over the matrix first letter. This notation was extracted from Oliveira and Macedo (2017), allowing the differentiation between OLAP measures and dimensions.

2.3 LINEAR ALGEBRA QUERY LANGUAGE

The LA approach presented by (Afonso and Fernandes, 2017) not only introduced a distinct way of encoding data, but also specified a set of operations capable of reproducing typical database queries. To formalise this set of operations in such a way that makes it suitable to be used by any developer, and capable of supporting an automatic conversion from SQL queries, a new DSL was defined and named LAQ.

This new language includes three key algebraic operators, from which all the LAQ operations are implemented: Dot, Khatri-Rao and Hadamard-Schur products. Three other derived operations are introduced to cover the complex syntax of SQL: the Filter, Fold, and Lift operators.

2.3.1 *Dot Product*

Mathematically, this product corresponds to the matrix multiplication, where $A \times B = C$, being only applicable if the number of columns of the matrix A matches the number of rows of the matrix B.

It can be defined as follows: for any entry v of C in the position $[x, y]$ (where $x \leq i \wedge y \leq k$), v is calculated by summing the product of all j values in the x row of A by the j values in the column y column of B. In (5) a sample product is displayed.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \times 1 + 2 \times 4 & 1 \times 2 + 2 \times 5 & 1 \times 3 + 2 \times 6 \\ 3 \times 1 + 4 \times 4 & 3 \times 2 + 4 \times 5 & 3 \times 3 + 4 \times 6 \\ 5 \times 1 + 6 \times 4 & 5 \times 2 + 6 \times 5 & 5 \times 3 + 6 \times 6 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 15 \\ 11 & 26 & 33 \\ 29 & 40 & 51 \end{bmatrix} \tag{5}$$

Again, thinking of a matrix as a function, leads to the use of its specific properties. One of the most useful is the function composition. As seen in (6), the requirements for function composition and matrix multiplication are the same, which allows a similar representation.

(6)

This property can be visualised in the TD notation. Considering the multiplication of two matrices (N and M), their composition can be represented by ($M.N$), thus the "dot" in the product's name.

Another interesting aspect of this property is that the dimensions have to align, just like types in function composition. It is also very perceptible to see the result matrix dimensions in the diagram. In this case, they are ($i \times k$).

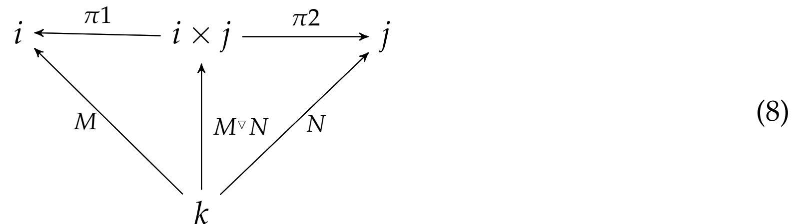
2.3.2 *Khatri-Rao Product*

The Khatri-Rao product of two matrices is represented by $A \triangledown B = C$. In this product, each row of the matrix A is multiplied by the whole matrix B (row by row), and the result is stored in the final matrix C.

If the initial matrices have dimension of $(i \times k)$ and $(j \times k)$ respectively, the result matrix will have $((i * j) \times k)$ dimension. In (7) its application is displayed.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \triangledown \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 \times 1 & 2 \times 2 \\ 1 \times 3 & 2 \times 4 \\ 1 \times 5 & 2 \times 6 \\ 3 \times 1 & 4 \times 2 \\ 3 \times 3 & 4 \times 4 \\ 3 \times 5 & 4 \times 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 8 \\ 5 & 12 \\ 3 & 8 \\ 9 & 16 \\ 15 & 24 \end{bmatrix} \tag{7}$$

In (8) it is possible to establish a parallelism between this operator and a function split, defined by $(f \triangledown g)x = \langle f(x), g(x) \rangle$. One of the important properties of this operator (also depicted in the diagram) is that no information is lost. From the result of a Khatri-Rao, one or both of the initial matrices can always be retrieved.



(8)

2.3.3 *Hadamard-Schur Product*

The Hadamard-Schur product is defined only for matrices of the same type, i.e. matching dimensions. $A \times B$ is implemented by multiplying every element in A by the corresponding element in B and storing it in the same position on matrix C. This way, the resulting matrix matches the dimensions of the original matrices, as seen in (9).

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 & 2 \times 6 \\ 3 \times 7 & 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix} \tag{9}$$

2.3.4 *Filter*

The attribute filter is the equivalent operator to the relational selection, which filters the columns of a matrix based on the labels of its corresponding rows. Specifically, this operator has, as input, both a predicate and a matrix. The predicate is directly applied to the labels of the matrix, generating a boolean vector that states which labels comply with the predicate. After this, a dot product between the result vector and the initial matrix is made. This generates another boolean vector, which identifies the columns that have a row satisfying the predicate. The TD (10) represents this clearly.

$$\begin{array}{ccc} i & \xleftarrow{M} & j \\ {\scriptstyle f}\downarrow & \swarrow {\scriptstyle f.M=\sigma(M_f)} & \\ 1 & & \end{array} \tag{10}$$

To understand this, an example is given. The matrix in (11) shows the representation of the attribute $o_{shippriority}$ in a matrix format. The applied filter (12) is a predicate that selects non-urgent shipments.

$$o_{shippriority} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} 1 & - & 1 & - \\ - & 1 & - & - \\ - & - & - & 1 \end{bmatrix} \end{array} \begin{array}{l} \text{1-URGENT} \\ \text{2-HIGH} \\ \text{3-MEDIUM} \end{array} \tag{11}$$

$$\neq \text{'1-URGENT'} = \begin{array}{c} \begin{matrix} \text{1-URGENT} & \text{2-HIGH} & \text{3-MEDIUM} \end{matrix} \\ \begin{bmatrix} - & 1 & 1 \end{bmatrix} \end{array} \tag{12}$$

The matrix in (13) is the result of the dot product of (12) by (11). It is a sparse boolean vector that specifies whether each column in the original matrix has a row that corresponds or not to a valid predicate.

$$\sigma(o_{shippriority} \neq \text{'1-URGENT'}) = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{bmatrix} - & 1 & - & 1 \end{bmatrix} \end{array} \tag{13}$$

2.3.5 *Fold*

A “!” is a vector of arbitrary length filled with "1"s. It is typically used to condense information, by reducing one of the matrix dimensions to the unit when both are multiplied. The TD in (14) exemplifies this process.

$$\begin{array}{ccc} i & \xleftarrow{M} & j \\ & \nwarrow^{fold(M)=M.!^o} & \uparrow !^o \\ & & 1 \end{array} \quad (14)$$

This way, a new vector is obtained, having as many rows as the initial matrix. Each value equals the fold of all of the elements of the corresponding row in the original matrix using the aggregation function of the SQL language, for instance: sum, count, avg, min, and max.

By directly applying the composition in (14), the sum and count aggregation functions can be implemented. Furthermore, the average operator can be theoretically defined by the division $sum/count$. A practical implementation can be optimised to do this division in a single operation, encapsulated in the fold operator. The example in (15) illustrates the explained operator. It is applied to obtain the sum of $l_{quantity}$.

$$sum(l^{quantity}) = \begin{bmatrix} 28 & 44 & 13 & 35 \end{bmatrix} . \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 120 \quad (15)$$

2.3.6 *Lift*

The lift operator applies a mathematical expression, defined by a lambda function, to a set of vectors, corresponding to the function arguments. This is done for each element in the provided vector(s), creating a new one with the obtained results, as exemplified in (16).

$$lift(2 \times l^{quantity}) = 2 \times \begin{bmatrix} 28 & 44 & 13 & 35 \end{bmatrix} = \begin{bmatrix} 56 & 88 & 26 & 70 \end{bmatrix} \quad (16)$$

2.4 CONVERSION ALGORITHM

The current system is oriented towards analytical querying. Considering the complexity of such queries, translating a SQL query into a series of LA operations has a negligible performance overhead in the complete query run-time. Yet, there are several paths that can be taken to reach the same query output. This way, certain decisions can be made to produce the most efficient one.

Although a possible approach would be to produce a range of viable alternatives and then pick the best one based on a predicted cost, Afonso and Fernandes (2017) developed a theoretically sustained algorithm, capable of providing an efficient LA expression for any supported OLAP query.

This way, as performance is a priority, it is crucial to have a precise idea of the cost of each operation to prioritise the most efficient ones during the conversion phase. A brief study conduced by Afonso and Fernandes (2017) revealed the dot product as the most costly operator[1], so the full development process was oriented to avoid its usage. Also, filtering an attribute results in a reduction of the number of non-zero elements in the filtered matrix. Computing these operations in the initial stages of the query processing results in a more efficient solution, thus making it another priority.

2.4.1 *The approach*

Afonso and Fernandes (2017) found the implementation of a generic algorithm to convert any given SQL query to its LAQ counterpart a complex task.

This complexity raises from two major obstacles. The first is the high complexity of SQL syntax and the variety of possible queries it can describe. Another great obstacle is that RA and LA are completely different paradigms, leading to a conversion process that does not have a direct "one to one" translation, achievable using only a Context-Free Grammar (CFG).

Like with any other complex problem, its solution is not immediately discoverable, but reached progressively. Afonso and Fernandes (2017) constantly used the queries from the TPC-H benchmark to extend the algorithm's capabilities. As more and more queries were considered, the algorithm gradually evolved to support them. The benchmark is also used in this dissertation to validate, and measure the efficiency, of the query processing.

The methodology proposed by Afonso and Fernandes (2017) is as follows:

1 Although this statement is not valid for every query, the dot product still has a negative impact in the streaming approach presented in Section 3.3

1) Combine independent predicates filtering the same attribute
2) Replicate the type diagram if there are disjunctions between different tables
3) Filter the attributes using the dot product
4) Combine all the filters from the same table
5) Join the tables respecting referential integrity
6) While there are more than one functional bitmap:
    6.1) Compose matrices whenever possible
    6.2) Khatri-Rao between all bitmaps (or Hadamard if both are vectors)
7) Merge the replicated diagrams respecting the logical tree
8) Perform the necessary operations depending on the aggregation functions
9) Sort the data as stated in the `ORDER BY` clause

Listing 3: SQL to LAQ convertion algorithm

1) Combine independent predicates filtering the same attribute

In RA, after the Cartesian product between all the tables involved in the query is made, the conditional expression in the `WHERE` clause is evaluated row by row, filtering the records.

In a columnar approach, each attribute must be evaluated separately. If the conditional expression can be divided into a set of conjunctive sub-expressions, each one only applied to a single attribute, the predicates in each set can be immediately combined, creating a new boolean vector. Otherwise, it is necessary to build the logical tree that represents the conditional expression.

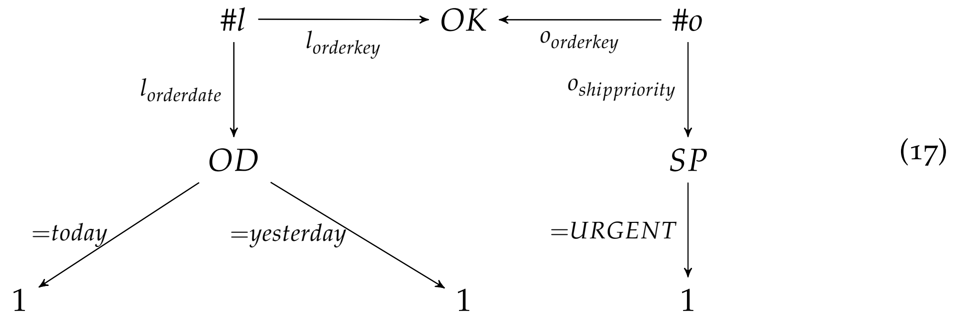
2) Replicate the type diagram if there are disjunctions between different tables

If the logical tree can be fully simplified, 1) will reduce it to a single root node and this step will not change the type diagram. When this is not possible, it is necessary to replicate the type diagram.

```
1  WHERE l_orderkey = o_orderkey
2    AND l_orderdate = 'yesterday'
3    OR (l_orderdate = 'today' AND o_shippriority = '1-URGENT')
```

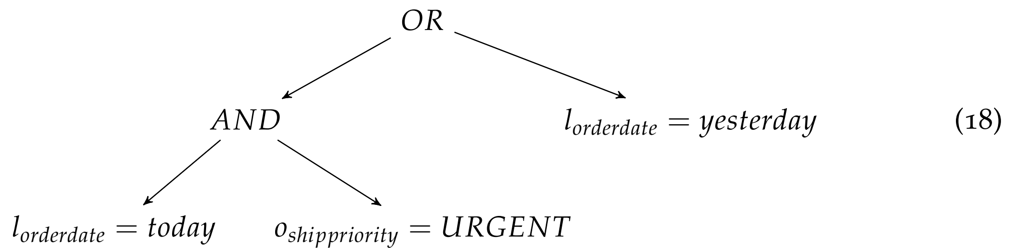
Listing 4: Example of a SQL `WHERE` clause

The SQL statement in Listing 4 exemplifies a situation where this problem arises. The predicates over $l_{orderdate}$ cannot be conjugated as stated in step 4), since it does not respect the logical expression



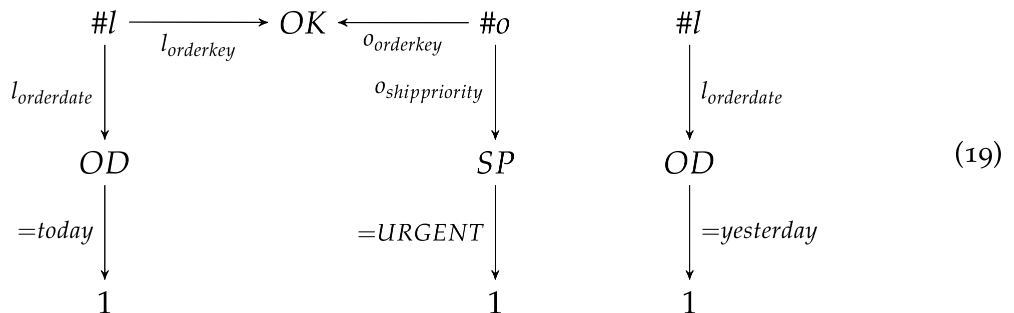
(17)

The type diagram (17) represents the presented SQL code. Since this notation cannot illustrate the difference between conjunctions and disjunctions, the logical tree in (18) was also built.



(18)

The necessary conditions are now met to proceed with the replication process, creating as many new TDs as the number of ORs in the logical tree, each one of them capable of being independently solved.



(19)

In this particular case, since there is only one OR conditional operator, only a new type diagram is formed, splitting the conditions among the two, as shown in (19).

3) Filter the attributes using the dot product

The goal is to obtain the vector that selects which columns of the bitmap will be kept, removing the others from the sparse representation. This reduces the amount of data in memory, which further improves the efficiency of the succeeding operations. Achieving this is done by composing the matrices of the filter and the attribute, as seen in (20).

$$
\begin{array}{ccccccc}
\#l & \xrightarrow[l_{orderkey}]{} & OK & \xleftarrow[o_{orderkey}]{} & \#o & & \#l \\
\downarrow{\scriptstyle f_{td}=filter(l_{orderdate}=today)} & & & & \downarrow{\scriptstyle f_1=filter(o_{shippriority}=1)} & & \downarrow{\scriptstyle f_{yd}=filter(l_{orderdate}=yesterday)} \\
1 & & & & 1 & & 1
\end{array}
\tag{20}
$$

4) Combine all the filters from the same table

As explained in 2), the conditions in each type diagram are independent from each other, so the diagrams can be further simplified.

At this stage, only the conditions relative to the same table can be resolved. Effectively, they can be "glued" by respecting the logical operators in the initial condition. For instance, if the initial expression was composed by an AND between two conditions, the logical conjunction would have to be performed, element by element, between the two condition vectors.

5) Join the tables respecting referential integrity

Referential integrity is a very common constraint in a relational database. This guarantees that, when a table has data that points to another table, all the made references are valid. For instance, if a given table has a foreign key that is relative to another table's primary key, this foreign key always takes a valid value that exists in the second table's primary key.

To guarantee this integrity, the proposed approach has to make sure that the cardinality of both attributes is the same. This may not be true if the foreign key attribute does not reference all the primary keys. Padding the foreign key's bitmap with rows full of zeros, in all the elements that originally do not occur in it makes them both have the same cardinality, solving this problem.

In theory, joining the tables can be achieved by calculating the dot product between the transpose of the primary key and the foreign key matrices, as illustrated in (21).

$$\begin{array}{ccc} OK & \xrightarrow{o_{orderkey}{}^{o}} & \#o \\ {\scriptstyle l_{orderkey}}\uparrow & \nearrow{\scriptstyle join=o_{orderkey}{}^{o}.l_{orderkey}=id^{o}.l_{orderkey}=l_{orderkey}} & \\ \#l & & \end{array} \tag{21}$$

Although this seems like a straightforward matrix composition, if the process of loading the data into the bitmaps is made correctly, the primary key of each table will always be an identity matrix, rendering the matrix multiplication redundant.

$$o_{orderkey} = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} 1 & - & - & - & - \\ - & 1 & - & - & - \\ - & - & 1 & - & - \\ - & - & - & 1 & - \\ - & - & - & - & 1 \end{bmatrix} \end{array} \begin{matrix} 21 \\ 22 \\ 23 \\ 24 \\ 25 \end{matrix} = o_{orderkey}{}^{o} = id \tag{22}$$

The matrix in (22) contains the loaded data for the primary key of the table *orders*: $o_{orderkey}$. If two rows from the matrix and its respective labels were swapped, it would represent the exact same information, but it would not be the identity. Thus, this property can only be applied if the loading process is done correctly.

By employing this method, the type diagram in (23) is achieved.

$$\begin{array}{ccccc} \#l & \xrightarrow[l_{orderkey}]{} & \#o & & \#l \\ {\scriptstyle f_{td}}\downarrow & & \downarrow{\scriptstyle f_{1}} & & \downarrow{\scriptstyle f_{yd}} \\ 1 & & 1 & & 1 \end{array} \tag{23}$$

6) While there are more than one functional bitmap:

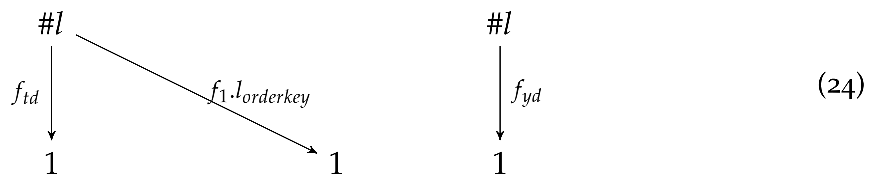
The next step to complete any query is to obtain a single functional bitmap. This bitmap specifies which values of the measure(s) should be considered, ignored or aggregated. The cardinality of this bitmap matches the number of rows of the query result, which is the same as the product of the cardinalities of all attributes involved in the `GROUP BY` clause. Its number of columns is equivalent to the number of records in the fact table.

Over this bitmap, from now on named function, all the aggregation operations can be applied. In this specific case, the translation process is being made over a simple SQL snippet, without a `GROUP BY` statement, thus the obtained function is a vector.

Since the practical meaning of the data stored in intermediate matrices starts to be quite ambiguous, the final steps to achieve the query solution are less intuitive than the previous ones. In summary, the Khatri-Rao product is used to merge the attributes in the `GROUP BY` clause, and the dot product to filter attributes using specified predicates, as well as join tables.

6.1) Compose matrices whenever possible

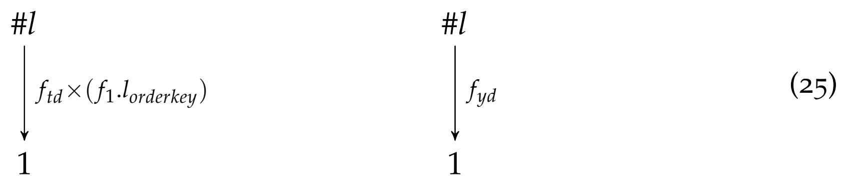
By applying the dot product, the composition in (23) is simplified and the type diagram in (24) is obtained.



(24)

6.2) Khatri-Rao between all bitmaps (or Hadamard if both are vectors)

Like mentioned, the Hadamard product can be used instead of the Khatri-Rao. In this case, after applying the dot product, a bifurcation was established, creating an opportunity for the usage of this rule. The type diagram (25) demonstrates the end result.



(25)

7) Merge the replicated diagrams respecting the logical tree

Using the logical tree calculated in 2), it is possible to join the functions of all type diagrams in a single one, using the logical disjunction element by element.

The functional bitmap in (26), represents the full LA translation of the initial `WHERE` statement.

$$1 \xleftarrow[function=f_{yd} \vee (f_{td} \times (f_1.l_{orderkey}))]{} \#l \tag{26}$$

8) Perform the necessary operations depending on the aggregation functions

An aggregation function will produce a dense vector, with as many rows as the `SELECT` statement cardinality. The value of each element depends on the applied aggregation function. These functions reduce every row of the matrix to a single value and, as explained in section 2.3.5, are globally defined as the fold LAQ operator.

Knowing that both a measure vector and a functional bitmap are available, the necessary conditions are met to determine the last set of LA operations that translate the query. The five major aggregation functions are supported (count, sum, average, min and max).

The count operation can be calculated using only the functional bitmap. This makes it so that the measure vector does not have to be loaded into memory, making it the most efficient of the aggregation functions.

Note that the count of the desired records is made by "summing" all the "1"s in the bitmap. Thus, to implement the sum operation, it is only necessary to put the measure values in the place of the "1"s before they are aggregated, using the Khatri-Rao product (or the Hadamard if the function is a vector).

Although the average is the more complex of the three operators, it can be easily obtained by dividing the result of the other ones, using the formula: $average = sum/count$. If the sum and/or count are not needed, it can also be incrementally calculated, storing the actual average and count.

The min and max functions follow a similar strategy, searching all the elements of an attribute for the min and max values.

9) Sort the data as stated in the `ORDER BY` clause

Without going through the contents of the matrices, it is impossible to perform a sort. For this reason, the content of the `ORDER BY` clause is directly transcribed to the LAQ script.

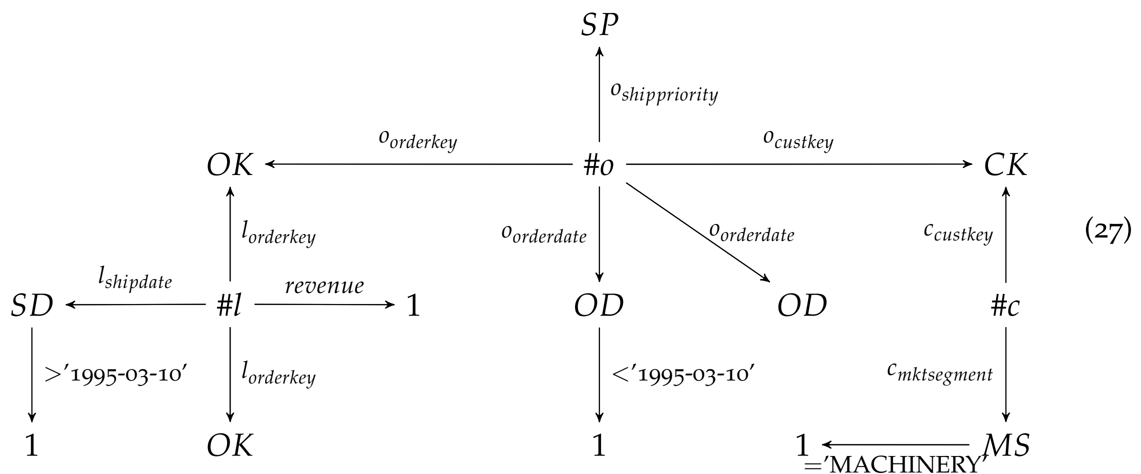
2.5 CONVERSION EXAMPLE

To consolidate all previous concepts and to better understand the utility of the introduced operators, a SQL query will be translated to a LAQ script. The first step to solve any query

is to build the correspondent TD. Using the query in Listing 1 as example, the TD that represents it is depicted in (27).

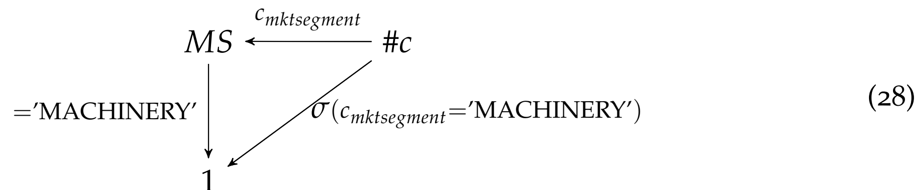
This diagram stands as a visual representation to support the query conversion process. It does not demand any construction rules that have not been introduced in Section **??**. However, the following conventions have been adopted:

- all attributes are represented as many times as they appears in the SQL script;
- each operation to be performed removes all the input matrices from the diagram, and adds the resultant one;
- revenue represents a vector containing the result of the provided arithmetic expression applied element-wise $revenue = lift(l^{extendedprice} * (1 - l^{discount}))$, and not the filtered and aggregated values



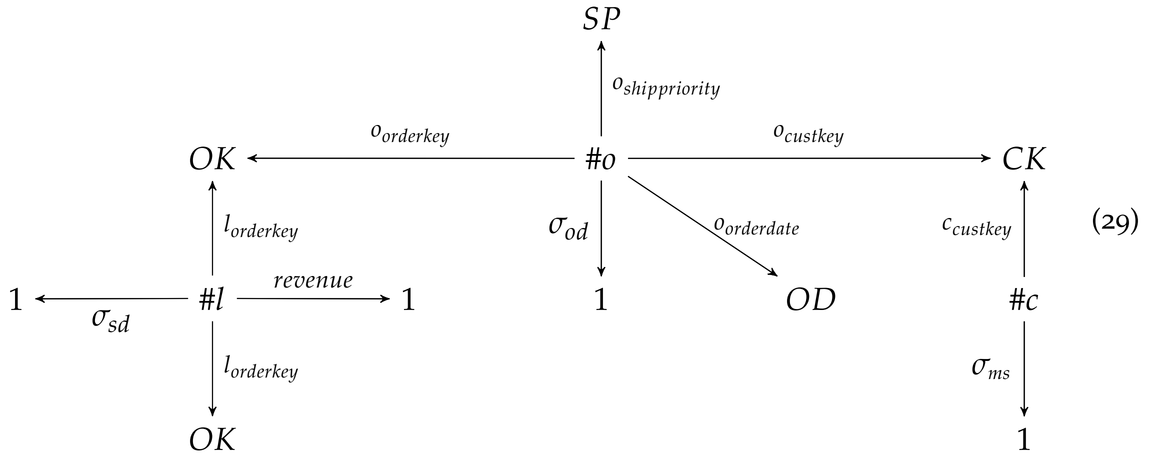
(27)

The first step in the query translation will be the composition of the filters. The Type Diagram in (28) illustrates the practical application of (10) to filter urgent priority records out of the $c_{mktsegment}$ attribute.



(28)

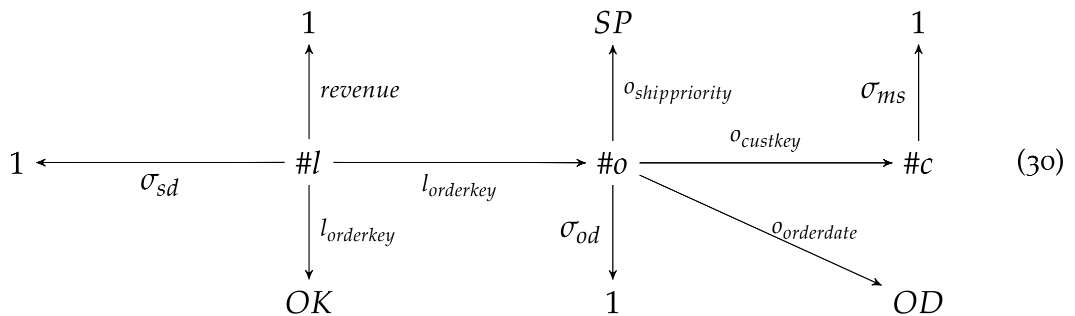
Applying this formula to all the filters in the query Type Diagram (27), results in (29), where $\sigma_{sd} = \sigma(l_{shipdate} > '1995\text{-}03\text{-}10')$, $\sigma_{od} = \sigma(o_{orderdate} < '1995\text{-}03\text{-}10')$, and $\sigma_{ms} = \sigma(c_{marketsegment} = \text{'MACHINERY'})$.



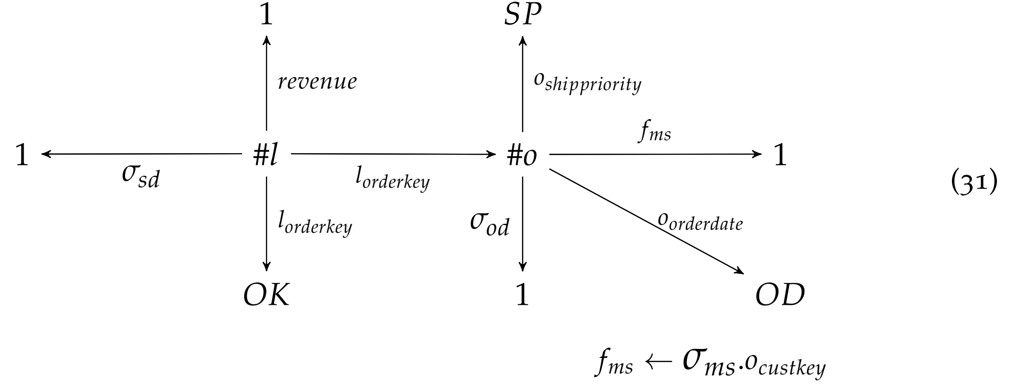
The joins between the tables involved in the query (*Lineitem*, *Orders*, and *Customers*), although partially hidden, are also solvable matrix compositions.

As explained in (21), if the $o_{orderkey}$ attribute is transposed, it can be multiplied by $l_{orderkey}$, obtaining a single matrix as presented in (30). This matrix makes the correspondence between the records in *Lineitem*, as foreign keys, and the ones in *Orders*, as primary keys.

Also, this composition is taken for free on behalf of the id property.

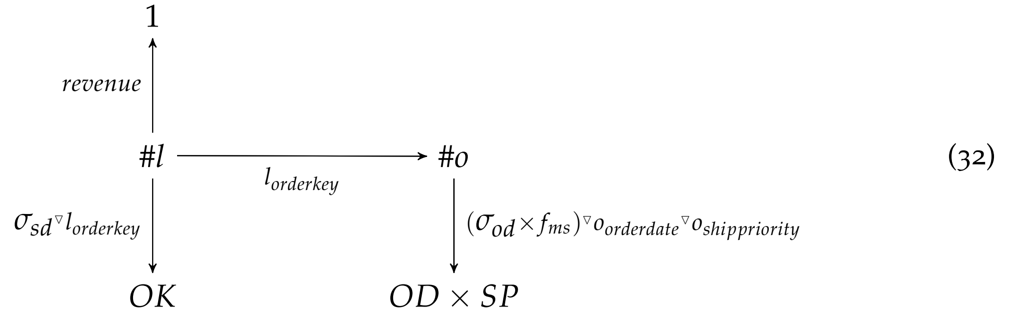


As shown in (30), there is still a composition to be made, after which the diagram (31) can be obtained.

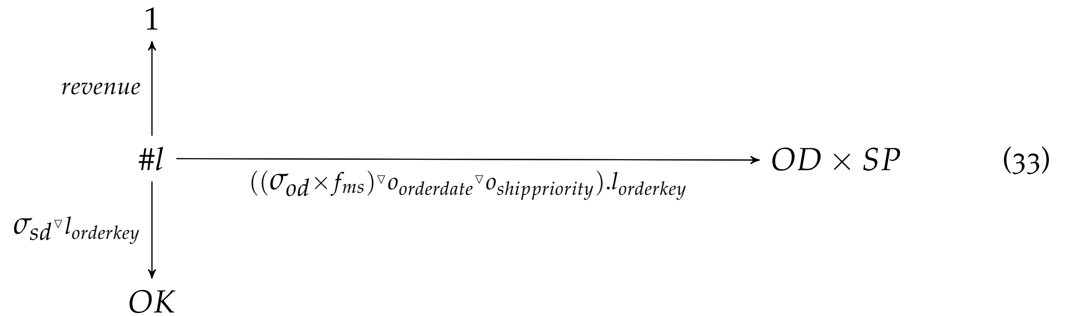


$$f_{ms} \leftarrow \sigma_{ms} . o_{custkey}$$

The matrices σ_{od} and f_{ms} have the same dimensions, thus being combined using the Hadamard product. Many other matrices share a single dimension, for instance #*l* and #*o*, being merged with the Khatri-Rao product. The diagram containing these modifications is depicted in (32).



Once the `GROUP BY` of the attributes in *Orders* is calculated, it can be combined with the join of the relations *Orders* and *Lineitem*, producing the diagram in (33).



Now, the final element of the `GROUP BY` clause, that is, the filtered matrix representing the attribute $l_{orderkey}$, can be merged with the rest of the `GROUP BY` elements.

$$
\begin{gathered}
1 \xleftarrow[revenue]{} \#l \xrightarrow[function]{} OK \times OD \times SP \\
function \leftarrow \sigma_{sd} \triangledown l_{orderkey} \triangledown (((\sigma_{od} \times f_{ms}) \triangledown o_{orderdate} \triangledown o_{shippriority}).l_{orderkey})
\end{gathered} \tag{34}
$$

As shown in (34), the diagram at this stage is composed by a single functional matrix, and a set of measures: in this case *revenue* (derived from $l^{extendedprice}$ e $l^{discount}$).

One possible way to combine all the information in a single vector, named query, is to calculate the Khatri-Rao between the measure and the function derived from the dimensions, then applying the needed fold operation: sum, to perform the row-wise aggregation of the matrix.

Compiling all the defined variables together with the final LA expression in a single equation results in the expression (35).

$$
\begin{cases}
query = sum(revenue \triangledown function) \\
revenue = lift(l^{extendedprice} * (1 - l^{discount})) \\
function = \sigma_{sd} \triangledown l_{orderkey} \triangledown (((\sigma_{od} \times f_{ms}) \triangledown o_{orderdate} \triangledown o_{shippriority}).l_{orderkey}) \\
\sigma_{sd} = \sigma(l_{shipdate} > \text{'1995-03-10'}) \\
\sigma_{od} = \sigma(o_{orderdate} < \text{'1995-03-10'}) \\
f_{ms} = \sigma_{ms}.o_{custkey} \\
\sigma_{ms} = \sigma(c_{marketsegment} = \text{'MACHINERY'})
\end{cases} \tag{35}
$$

Encoding this expression in the LAQ syntax, the code snippet in Listing 5 is obtained, this way completing the translation process.

```
1  A = filter( o_orderdate < "1995-03-10" )
2  B = krao( A, o_orderdate )
3  C = filter( c_mktsegment = "MACHINERY" )
4  D = filter( l_shipdate > "1995-03-10" )
5  E = dot( C, o_custkey )
6  F = krao( l_orderkey, D )
7  G = krao( B, E )
8  H = krao( G, o_shippriority )
9  I = dot( H, l_orderkey )
10 J = krao( F, I )
11 K = lift( l_extendedprice * (1 - l_discount) )
12 L = krao( J, K )
13 M = sum( L )
14 return( M )
```

Listing 5: TPC-H Query 3 - LAQ version

2.6 SUMMARY

This chapter extensively described the TLA approach to data querying. It started with the analysis of the matrices used to encode OLAP measures and dimensions. Then, the concept of TD was introduced as a visual way of representing queries. Finally, a DSL designed to encode LA queries was specified, as well as a methodology to produce it from its SQL counterpart.

The following chapters contain the full details of the implementation of these ideas in a single framework, followed by its validation and performance evaluation.

3

A TLA-DB ENGINE FOR RELATIONAL SQL

The developed framework in this work is based on LA and in the type theory placed above it. Time has shown how mathematical interfaces tend to lose out for simpler and easier-to-use implementations, while the success of RDBMSs is strongly linked to the higher level of abstraction and robustness they provide, comparatively to the previously implemented network and hierarchical systems.

As stated by Stonebreaker et al. (2015, p. 6), "*SQL will be the COBOL of 2020, a language we are stuck with that everybody will complain about*". SQL is here to stay, and even the non-relational approach proposed in this dissertation should provide a SQL interface.

The key goal of the developed framework is to perform analytic querying on databases. Once identified the two main requirements of the database system, the correspondent software components can be defined: a SQL driver that reads SQL queries and translates them to their LAQ equivalent, and a database engine that executes the LAQ queries (Figure 3).

Figure 3.: System architecture

This chapter addresses the detailed analysis of the two framework components (Figure 4), starting with the key issues, namely an efficient data representation, the required LA operations and a streaming approach, followed by the description of the software components in the overall architecture.

3.1 MATRIX REPRESENTATION

Typical OLAP implementations tend to be supported for at least one transactional database, and data is loaded into the data warehouse at specific periods of time (the so-called windows of opportunity). Fast updates and single record insertions are not essential; for now, the only implemented method for data loading is the SQL's `COPY FROM`.

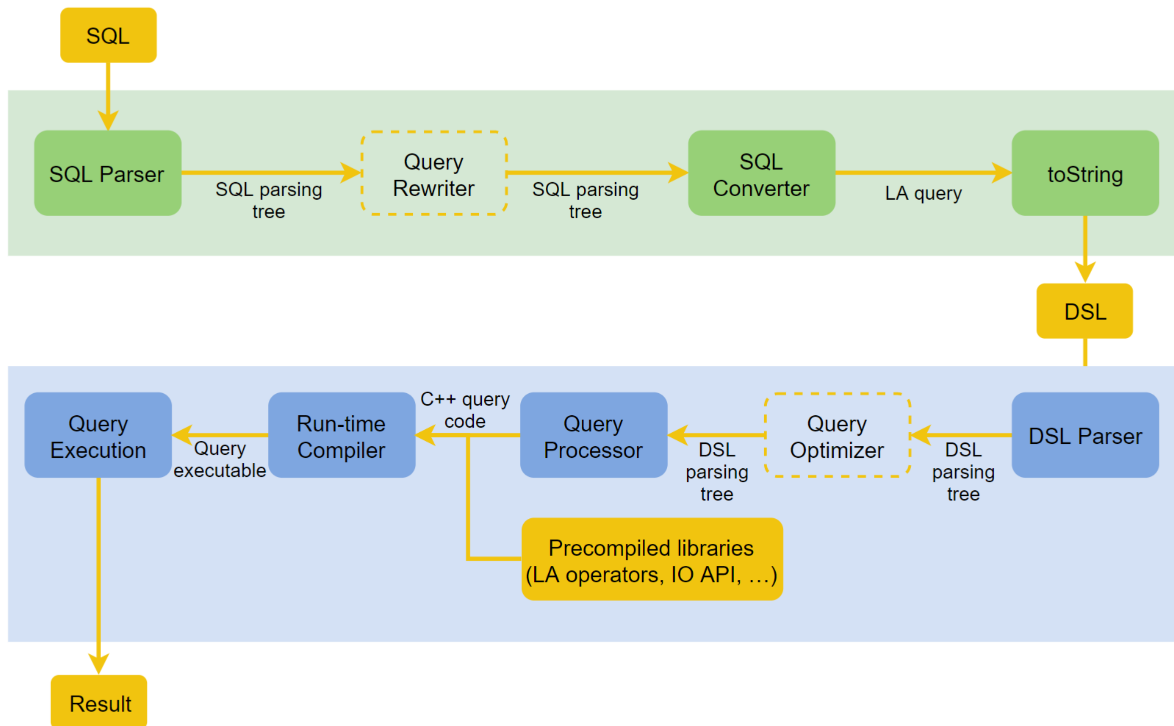


Figure 4.: Detailed system architecture

The implemented `COPY FROM` operation takes any text file with delimited columns (such as comma-separated values (CSV), tab-separated values (TSV), ...), its delimiter, and a map with the desired column indexes and the corresponding attribute names, loading their data into the specified attributes.

Having a way of loading data, it is necessary to define an efficient way of encoding it. Since attributes are matrices, efficient matrix representation formats were required.

A matrix in memory can be represented in multiple ways. Since dimensions are stored in bitmaps (matrices containing either zeros or ones), and the number of zeros is orders of magnitude higher than the number of ones, dense representations can be immediately excluded. Three formats for sparse matrix encoding have been analysed, namely List of Lists (LIL), Coordinate List (COO) and Compressed Sparse Row (CSR)/CSC.

3.1.1 *LIL*

Similar to a pointer based representation of a matrix, the LIL format is composed by an array with as much elements as the number of rows in the matrix, where each element is a list of pairs (column, value).

Considering that the Khatri-Rao operation, for example, can easily produce matrices with billions of rows, the representation of the main array can easily scale to gigabytes of unnecessary data, making this format inadequate to encode this kind of matrices.

3.1.2 *COO*

This matrix encoding format simply stores tuples (value, row, column). Since no metadata is attached to this representation, it is the most memory efficient of the presented ones when applied to functional matrices.

This format ensures that all matrix elements can be iteratively accessed, thus powering algorithms with $\mathcal{O}(NNZ)$ complexity. Considering that some of the algorithms presented in Section 3.2 require column-wise accesses to the matrix, the tuples should be ordered by their column indexes.

It is also noteworthy that the matrix transposition corresponds to a simple sort of the matrix records based on its row indexes, having an estimated cost of $\mathcal{O}(NNZ \times \log NNZ)$.

However, this format has a huge drawback. In query processing data tends to be filtered, producing matrices with few columns of data. When performing operations like the `GROUP BY`, by combining these matrices with database attributes that have a value in each column, both COO matrices must be iterated.

3.1.3 *CSC/CSR*

Both CSC and CSR formats are very similar. The only difference is if the compressed vector keeps the row or column indexes. Since CSR has the same memory problem than LIL, and considering that most of the algorithms of this approach require column indexing, the CSC format seems more adequate and it was the selected for this work.

This format consists in three basic 1D-arrays.

- `values`, with the values of each non-zero element to be represented;
- `row_indices`, with the row indices of each non-zero value;
- `column_pointers`, with size `n_columns+1`, that specifies the position of the non-zero elements of a given column in the other arrays, together with the number of non-zero values of each column; the 1st value is 0 and the `column_pointer[column]` is given by adding the number of non-zero values in [column-1] to the `column_pointer[column-1]`.

Figure 5 illustrates how a matrix is compressed in CSC.

Sample matrix

$$\begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Same matrix in CSC format

values	$\begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$
row_indices	$\begin{bmatrix} 0 & 0 & 2 & 1 \end{bmatrix}$
column_pointers	$\begin{bmatrix} 0 & 1 & 1 & 3 & 4 \end{bmatrix}$

Figure 5.: A sparse matrix in CSC format

The compression achieved by this representation comes from the establishment of a fixed size array for the column indexes, having as many elements as columns in the matrix. Each element of this vector contains the cumulative counting of the elements per column.

The compression is achieved because only one value per column is stored in the column pointer array. However, since functional matrices have at most one value per column, this is not an optimization over COO, but its worst case scenario.

Despite its higher memory usage, this encoding format has an advantage in operations with random accesses to the matrix columns. This happens because the column pointer array serves as an hash from the element in a given column to the position in the arrays containing its value and row. This way, an otherwise quadratic algorithm can be run in linear time. That is the main reason why CSC was adopted.

Having chosen CSC as the encoding format, its use can be further optimised regarding data properties:

1. Dimensions are Bitmaps, so every value in the matrix is a "1", thus they are not stored.
2. Measures are uni-dimensional (horizontal) Decimal Vectors, thus the row array is not needed.
3. Both measures and dimensions have a single element per column, thus the column pointer array will be a subset of $\mathbb{N}_0$; since it can be easily derived, it is not necessary to represent it.
4. To benefit from CSC properties, all vectors will be represented horizontally, despite their original orientation.

During query processing, intermediate matrix representations are neither measures or dimensions. In fact, every combination of the columns, rows and values arrays has its utility, which is summarised in Table 6.

With these seven types of matrices, disk and memory usages are highly optimised. The next natural step is to find the best format to power parallel data processing.

Considering a set of random operations, a simple way to explore parallelism would be to use a distinct processing unit for each operation in the query script, creating a data flow from disk to the query result.

However, this strategy is strongly tied to the query topology, raising two major issues:

Block type	Application example	Values	Rows	Columns
Bit-vector (bang)	No actual usage	–	–	–
Decimal Vector	Stored measures	✓	–	–
Bitmap	Stored dimensions	–	✓	–
Filtered Bit-vector	Result of filter operations	–	–	✓
Decimal Map	Query with a `GROUP BY` clauses before aggregations, and after doing the *measure* ▽ *function* operation	✓	✓	–
Filtered Decimal Vector	Query with a `WHERE` clause before aggregations Result of a query containing `WHERE` and `GROUP BY` clauses	✓	–	✓
Filtered Bitmap	Function containing (part of) the `WHERE` and `GROUP BY` clauses	–	✓	✓
Filtered Decimal Map	Query with `WHERE` and `GROUP BY` clauses before aggregations	✓	✓	✓

Table 6.: Properties of CSC block types

- if the processing units are spread across multiple servers, data will need to be transferred after most operations; to minimise communication overheads, task attribution would be a complex process;
- the number of tasks is the same as the number of operations in the LAQ script, leaving very few opportunities to optimise load balances; for instance, computing a query in a many-core, GPU, or even a recent multi-core device, would result in fewer tasks than the available processing units in the chip, most of them doing no work.

To overcome this situation matrices were divided into multiple blocks of fixed size, and each processing unit computes all the operations in the LAQ script for a group of blocks.

This division works perfectly for operations that can be divided by column, that is, $[A|B] \otimes [C|D] = [A \otimes C|B \otimes D]$, where $\otimes$ represents any operation. This way, if a given matrix is stored in multiple servers, the operations can be computed in those servers, avoiding costly data transfers.

However, some operations have random accesses to the columns of the matrices, creating the necessity of having the full matrix loaded in the main memory. Making a connection with SQL, these operations are either part of an aggregation after a `GROUP BY` or part of a `JOIN`, thus suffering from the same problems of their relational counterparts.

3.1.4 *Matrix labels*

Recall that each row of the projection matrix of an attribute corresponds to a data label. The number of rows is the number of distinct labels that the attribute has. One of the challenges of the LA approach to data representation is how to represent such data labels, that is, how to map attribute values to unique matrix indices.

To ensure that every label maps to a distinct matrix row, they are dynamically inserted in an hash table, incrementally taking the first available integer value, in an `AUTO INCREMENT` fashion, as in other DBMSs.

This hash table has $\mathcal{O}(1)$ complexity when retrieving the row of a given record, as needed during data insertion. A double hashing approach could be used to improve performance of consulting queries, since it leads to a complexity of $\mathcal{O}(1)$ on both directions of the association.

However, the computation of an hash function for every row of the matrix is an undesired overhead. As the keys of the second hash are the row numbers, it is encoded as an array where the position represents the row number. Both structures are filled during data insertion, but only the array is loaded during consulting queries. To further optimise the memory usage, this array was also divided in multiple blocks.

This strategy is applied independently for each attribute. However, for referential integrity to hold among all database tables, such structures are shared by both primary and foreign key attributes. Primary key attribute values are always mapped first. Since these keys must be `DISTINCT` and `NOT NULL`, there is a bijection between row numbers and key values.

3.2 LA OPERATORS

Improved versions of the algorithms behind the results presented by Ribeiro et al. (2017) have been implemented. The major improvements are in memory management, using the explained block approach, and in complexity reduction.

Not only the adoption of multiple block types saves memory, but it also allows the implementation of distinct algorithms, optimised for the blocks and matrices being processed. This way, each LAQ operator will have multiple implementations, corresponding to all the combinations of the possible types for the two input matrices.

These multiple implementations tend to be quite repetitive, but by considering the data constrains of the matrices they receive as input, they easily outperform state of the art linear algebra kernels such as the Intel Math Kernel Library (Intel MKL) and the C language Basic Linear Algebra Subprograms (CBLAS).

3.2.1 *Hadamard-Schur Product*

As described in Section 2.4.1, the conversion process only applies this product to combine vectors. It was used by Ribeiro et al. (2017) as an optimisation of the generic Khatri-Rao product applied to vectors, since both produce the same results in this case.

However, as stated, every possible combination among all the block types will have a dedicated operation, thus the Khatri-Rao of two vectors will have the same performance of the Hadamard-Schur product, if not better. This way, the latter will not be implemented, and the Khatri-Rao will be used in its place.

3.2.2 *Khatri-Rao Product*

As explained, there are seven types of blocks. Considering that the Khatri-Rao product takes two matrices as input, it has 49 distinct implementations.

The output matrix is not counted here, because its type is derived from the input ones. This derivation is achieved by checking which arrays are in at least one of the input matrices, and then selecting the block type that contains all of them.

In the 49 distinct algorithms of the Khatri-Rao product, most just change the arguments order, and other simply combine two or more simpler variations. This way, the 49 implementations can be explained using only 6 distinct actions.

To merge values

$$[ht] \mathit{Decimal\,vector} \triangledown \mathit{Decimal\,vector} = \mathit{Decimal\,vector}$$

Although a valid operation, a Khatri-Rao where both input matrices are decimal (have the values array) leads to is non-fully optimised query. The natural path to solve a given query would be to simplify it to the point where it is composed by a single functional matrix and a couple of measures, that is, the known pairing wheel (Afonso et al., 2018).

Consider two matrices: A and B. The Khatri-Rao product can only be used to multiply them ($A \times B$); but if the desired operation is another one, for instance $A + B$, or even $A \times B + 1$, the Khatri-Rao is useless, and the lift operator should be used instead (see Section 3.2.6).

To merge rows

$$Bitmap \triangledown Bitmap = Bitmap$$

As stated in Table 6, Bitmaps are used to encode dimensions. A Khatri-Rao of two Bitmaps condenses in a single matrix the information contained in both input matrices.

Considering the Khatri-Rao product: $C = A \triangledown B$, it is known that the row index of the "1" in C is based on its original position in both A and B, being $row_C = row_A \times B_total_rows + row_B$.

The application of this formula is illustrated in (36), both in dense and CSC representations, having $0 = 0 \times 2 + 0$, $2 = 1 \times 2 + 0$, $3 = 1 \times 2 + 1$, and $4 = 2 \times 2 + 0$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \triangledown \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{matrix} values & - & & - & & - \\ rows & \begin{bmatrix} 0 & 1 & 1 & 2 \end{bmatrix} & \triangledown & \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} & = & \begin{bmatrix} 0 & 2 & 3 & 4 \end{bmatrix} \\ columns & - & & - & & - \end{matrix} \tag{36}$$

To combine values and rows

$$Decimal\ vector \triangledown Bitmap = Decimal\ map$$

If a query has no filter (`WHERE` clause), its functional part will only encode the `GROUP BY` statement, thus being a Bitmap (see Table 6). To solve a query at this state, a Khatri-Rao of the mentioned function and the needed measures (decimal vectors) should be performed.

Since neither the Decimal vector nor the Bitmap have the column pointer array, both matrices contain the same number of elements.

The Khatri-Rao is performed by simply copying the values array from the Decimal vector and the rows array from the Bitmap to the resultant Decimal map, making this a perfect example of the optimisation achieved by using only the necessary arrays. Also note that if one or both input matrices are not used again in the query, the arrays can be moved instead of copied, reducing the algorithm's complexity from linear to constant.

To select values

$$Decimal\ vector \triangledown Filtered\ bit\text{-}vector = Filtered\ decimal\ vector$$

Every algorithm multiplying filtered matrices iterates the column pointer array, comparing every pair of subsequent elements. If the two elements in the pair are distinct, that means that the current matrix column has an element, thus some action must be performed. In this case, this action is the insertion of the correspondent value in the output matrix.

As shown in (37), the only value that was not passed to the output was the "3", corresponding to the (2,2) pair in the column pointer array, and to the third position in the original Filtered bit-vector.

$$
\begin{array}{rcccl}
 & \begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} & \triangledown & \begin{bmatrix}1 & 1 & 0 & 1\end{bmatrix} & = \begin{bmatrix}1 & 2 & 0 & 4\end{bmatrix} \\
\\
\textit{values} & \begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} & & - & \begin{bmatrix}1 & 2 & 4\end{bmatrix} \\
\textit{rows} & - & \triangledown & - & = \; - \\
\textit{columns} & - & & \begin{bmatrix}0 & 1 & 2 & 2 & 3\end{bmatrix} & \begin{bmatrix}0 & 1 & 2 & 2 & 3\end{bmatrix}
\end{array}
\tag{37}
$$

To select rows

$$\textit{Bitmap} \;\triangledown\; \textit{Filtered bit-vector} \;=\; \textit{Filtered bitmap}$$

This operation is similar to the previous one, but instead of copying the values from the Decimal vector, the row indexes in the Bitmap are passed to the output matrix.

To merge columns

$$\textit{Filtered bit-vector} \;\triangledown\; \textit{Filtered bit-vector} \;=\; \textit{Filtered bit-vector}$$

As explained, if there is a column pointer array it must be iterated. However, as both matrices have this array, it is necessary to look to the number of elements in each matrix. The one with fewer elements should be iterated, because the comparison of the columns array elements will be false more often, avoiding unnecessary calculations.

So, the columns pointer array of the matrix with fewer elements is iterated and the action that should be performed is a look-up at the exact same position in the other matrix columns array: the element is only inserted in the final matrix if both comparisons are true.

The application of this algorithm is shown in (38). Note that by inserting an element in the matrix, the number of elements in the column pointer array does not change, but its value is incremented by one.

$$
\begin{array}{rcccl}
 & \begin{bmatrix}0 & 1 & 1 & 1\end{bmatrix} & \triangledown & \begin{bmatrix}1 & 1 & 0 & 1\end{bmatrix} & = \begin{bmatrix}0 & 1 & 0 & 1\end{bmatrix} \\
\\
\textit{values} & - & & - & - \\
\textit{rows} & - & \triangledown & - & = \; - \\
\textit{columns} & \begin{bmatrix}0 & 0 & 1 & 2 & 3\end{bmatrix} & & \begin{bmatrix}0 & 1 & 2 & 2 & 3\end{bmatrix} & \begin{bmatrix}0 & 0 & 1 & 1 & 2\end{bmatrix}
\end{array}
\tag{38}
$$

Generic product

$$\textit{Filtered decimal map} \;\triangledown\; \textit{Filtered decimal map} \;=\; \textit{Filtered decimal map}$$

By combining some of previous versions of the Khatri-Rao product, the other 43 can be easily defined. However, there is another version that is noteworthy, the Khatri-Rao of two CSC matrices containing all the three arrays. This operation can handle the product of two generic CSC matrices, applying all the previously explained actions.

```
1  krao (A, B, C):
2      assert A.nCols = B.nCols
3      nnz = 0
4      // Pick the matrix with fewer elements
5      if A.nnz < B.nnz:
6          // Iterate the column pointer array
7          for i = 0 to A.nCols:
8          // Store the current number of elements
9              C.cols[i] = nnz
10             // Check if an element is present in both matrices
11             if A.cols[i+1] > A.cols[i] and B.cols[i+1] > B.cols[i]:
12                 // Get the current indexes of the values and rows arrays
13                 Apos = A.cols[i]
14                 Bpos = B.cols[i]
15                 // Merge the values
16                 C.values[nnz] = A.values[Apos] * B.values[Bpos]
17                 // Merge the rows
18                 C.rows[nnz] = (A.rows[Apos] * B.nRows) + B.rows[Bpos]
19                 // Increment the number of elements
20                 nnz += 1
21     else:
22         // Same algorithm, just swap the predicates of the if condition
23     C.nnz = nnz
24     C.nRows = A.nRows * B.nRows
25     C.nCols = A.nCols
```

Listing 6: Generic Khatri-Rao of CSC matrices - detailed pseudo-code

As explained in Listing 6, the columns of the matrix with fewer elements should be iterated. After checking that a given element is present in both matrices (merge columns), it is necessary to get the values and row indexes of that element in both input matrices (select values/rows). Finally, these numbers must be multiplied and inserted in the resultant matrix (merge and combine values and rows).

An example of the application of the described algorithm, with both dense and CSC representations, is shown in (39).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \triangledown \begin{bmatrix} 0 & 2 & 0 & 3 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{matrix} values \\ rows \\ columns \end{matrix} \begin{matrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 & 2 & 3 \end{bmatrix} \end{matrix} \triangledown \begin{matrix} \begin{bmatrix} 2 & 1 & 3 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 1 & 2 & 3 \end{bmatrix} \end{matrix} = \begin{matrix} \begin{bmatrix} 4 & 9 \end{bmatrix} \\ \begin{bmatrix} 2 & 4 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 1 & 1 & 2 \end{bmatrix} \end{matrix} \tag{39}$$

3.2.3 *Dot Product*

As explained in section 2.3.1, the result of the dot product has as many rows as the leftmost argument. Therefore, the method used to deduce the block type is slightly different from the one used in the Khatri-Rao product. The values and column pointer arrays of both input matrices are combined in a similar way. However, the resulting matrix will only have the row index array if the leftmost argument also has it.

This product is one of the exceptions to the inherent parallelism of the block based approach. The adoption of the CSC format implies that this blocking strategy can only be applied column-wise. This way, the parallelization of the dot product results in:

$$\begin{aligned} A \cdot B &= \left[\ A_1 \mid A_2\ \right] \cdot \left[\ B_1 \mid B_2\ \right] \\ &= \left[\ \left[\ A_1 \mid A_2\ \right] \cdot B_1 \ \middle|\ \left[\ A_1 \mid A_2\ \right] \cdot B_2\ \right] \\ &= \left[\ A \cdot B_1 \mid A \cdot B_2\ \right] \end{aligned}$$

This means that the matrix B can be divided in multiple blocks. However, the partition of the matrix A is useless in this context, since all its blocks must be in memory during the multiplication with each block of B.

For this reason, the implementation of the dot product receives a matrix and a block, producing a single block of the output matrix. To complete the product of two matrices, it should simply receive a reference to the same matrix A, together with the multiple blocks of B, one at a time.

Typically, the dot product is used to join tables or filter dimensions. In both cases, the second argument is mostly a Bitmap and the first is often a Filtered Bit-vector or a Filtered

Bitmap. Anyway, the generic version is detailed in Listing 7, complying with the description of the Khatri-Rao product.

```
dot (A, B, C):
    assert A.nCols = B.nRows
    nnz = 0
    // Iterate the column pointer array of B
    for i = 0 to B.nCols:
    // Store the current number of elements
        C.cols[i] = nnz
        // Check if there is an element in B
        if B.cols[i+1] > B.cols[i]:
            // The column to look in A is the element's row in B
            Bpos = B.cols[i+1]
            Brow = B.rows[Bpos]
            Acol = Brow
            // Check if there is an element in A
            if A.cols[Acol+1] > A.cols[Acol]:
                Apos = A.cols[Acol+1]
                // Merge the values
                C.values[nnz] = A.values[Apos] * B.values[Bpos]
                // Save the row
                C.rows[nnz] = A.rows[Apos]
                // Increment the number of elements
                nnz += 1
    C.nnz = nnz
    C.nRows = A.nRows
    C.nCols = B.nCols
```

Listing 7: Generic dot product of CSC matrices - detailed pseudo-code

3.2.4 *Filter*

Considering the five distinct types of `WHERE` conditions presented in Section 1.1.2, the filter operator has an important role on solving most of them:

1. The direct **comparison** of two expressions, e.g. $a \geq b$, can be solved with a simple `filter(a >= b)`;
2. The boolean value that indicates if an expression is within a defined **range**, e.g. $a\ BETWEEN(x, y)$, can be converted to a more complex filter with two predicates: `filter(a >= x AND a <= y)`;
3. The boolean value that indicates if an expression equals an element in a **set**, e.g. $a\ IN(x, y, ...)$, can be divided in multiple simple predicates: `filter(a == x OR a == y OR ...)`; note that if the set is another `SELECT` statement, its elements are unknown, thus other operations must be used to solve the `IN` clause;

4. The boolean value that indicates if an expression matches a predefined **pattern**, e.g. *a LIKE "pattern"*, can be implemented similar to the SQL's `LIKE`, but receiving regular expressions instead of wildcards; too highlight this difference, it was named `MATCH`, and its application is as follows: `filter(match(a, ''pattern''));`

5. If the value is **NULL**: NULL values are not allowed in the presented system, in conformity with OLAP rules.

As this operator receives an arbitrary boolean expression, there is no way it can be compiled for any possible input, so it was implemented as a precompiled wrapper that receives a group of values and a boolean function to merge them. This function is defined in conformity with the LAQ query that is being processed and then linked against the wrapper.

Measures

To filter a measure, one only has to go through the elements of its values array, calling the boolean function with its content, which is a straightforward process.

Dimensions

In Section 2.3.4 this version of the operation is depicted in a slightly abstract way. Matrices (11) to (13) denote the multiple steps used to obtain its solution, but previous versions of the operator (Ribeiro et al., 2017) required any attribute to be loaded as a measure (array of values), thus needing two representations of each attribute to perform either filters or other operations.

In (12) is represented the result of the predicate's *calculus* over the matrix labels. By multiplying the obtained predicate with the Bitmap in (11) using the Dot product, the Filtered Bit-vector (13) is obtained.

So, the path for computing this operation would be to iteratively load the blocks of labels, calculating the predicate Filtered Bit-vector. Then do a dot product of the fully calculated predicate with each block of the Bitmap, that should be also loaded iteratively.

The operation *per se* only filters the blocks of labels, thus expecting a special block, where values are not numbers but strings.

3.2.5 *Fold*

The fold operator is quite distinct from its theoretical definition. Defining a bang vector and multiplying it with any matrix as explained in Section 2.3.6 would be a highly unoptimised process, specially since the bang is transposed, thus it is a columnar vector, and the algebraic optimizations only outperform libraries like Intel MKL and CBLAS since they take as granted that there is at most a single element per column in every matrix.

Even though some algorithms can simulate this multiplication in linear time, the storage of the results in CSC format would enforce random insertions in the output vector, making it a $\mathcal{O}(N^2)$ worst case algorithm.

To improve the worst case cost to $\mathcal{O}(N \times \log N)$, a C++ map from the standard lib was used. Each key in the map identifies a distinct row in the matrix to be aggregated, and the correspondent value its sum, count, min, and/or max.

The operator is divided in two distinct stages, the accumulation of elements inside the map, and a final conversion from the ordered map to a CSC vector or a Decimal element if the matrix being folded is already a vector.

It is imperative to use this operation as late as possible in query processing, since it is the only one that cannot be performed in linear time.

3.2.6 *Lift*

The lift operator was already partially explained in Section 3.2.2, and the algorithm used in its implementation is similar to the one used to filter measures (see Section 3.2.4). Instead of receiving a boolean function and producing a Filtered Bit-vector, the used function must return a Decimal, thus the produced result is a Decimal vector.

3.3 "STREAMING" APPROACH

The results presented by Ribeiro et al. (2017) need all data used in a given query to be loaded into memory before starting to compute the query result. This initial load phase powers the high performance of the processing phase, as in other in-memory database systems. However, if the necessary attributes do not fit in the main memory, the query will not run.

To enable querying on huge databases, a streaming approach has been designed. As explained, it consists in splitting the sparse matrices in sparse blocks of fixed size, processing them iteratively whenever possible.

Although it is not in the scope of this dissertation, it is extremely important to explore the inherent parallelism of the LAQ, and the commutative and associative algebraic properties when processing each block of the stream. The ultimate goal is to ensure that any given set of query operations is executed in optimal order (see Section 5.1.1).

3.3.1 *Dependencies in query processing*

Even though the referred scheduler would be able to dynamically reorder the query operations, an optimised version of the query should be used from the beginning. Since both

processing time and used memory are strongly tied to the data flow through the multiple operations, their data access requirements must be considered.

Weak dependencies

No matter what kind of input a LAQ operation is expecting, it will only work when data is provided. For instance, consider the Khatri-Rao product: to produce a block of the output matrix, this operator needs another two blocks; one for each argument.

This necessity of data inherent to any operation defines the weak dependencies. Although it seems elementary, the LAQ itself does not specify any load operations, which are abstracted from the end user; but these are of extreme importance for scheduling purposes, mainly due to the impact of disk accesses in the overall performance of any query.

Figure 6 contains an execution plan for the query 6 of the TPC-H. The white boxes represent the tree of operations, including the hidden load operation as the leafs of the tree, and the query result as its root. Also note the replacement of the filter on $l_{shipdate}$ by a filter on its labels, and a dot product with its Bitmap.

Strong dependencies

Sections 3.2.3 and 3.2.5 respectively provide useful information on the dot product and fold operators implementation. From there, and having in mind that these are the only two operations that create strong data dependencies, it is easy to infer that strong dependencies represent the necessity of having a given matrix totally loaded in main memory.

As expected, if a matrix has to be fully loaded before proceeding to the next operation, all the blocks of the previous ones have to be processed in advance, interrupting the stream of data. Also, the dot product and the fold operators produce slightly different interruptions: while the dot product needs its left input matrix previously computed to run, the fold can be executed anytime, just breaking the data-flow after its execution.

In Figure 6, these strong dependencies are shown as blue boxes and double arrows.

3.3.2 *Execution order*

There are multiple possibilities to go through the elements of a binary tree, which are closely related to the priority given to the processing of its left child, right child, or actual (parent) nodes.

The strategy used to go though the operations of any query is dictated by the explained data dependencies. Weak dependencies of data dictate that a parent node can never be processed before its child nodes, so queries must follow a bottom-up approach.

With strong dependencies things get a bit more complex. The dot product needs its left argument calculated, thus priority should be given to the left side of the tree. Folds can be

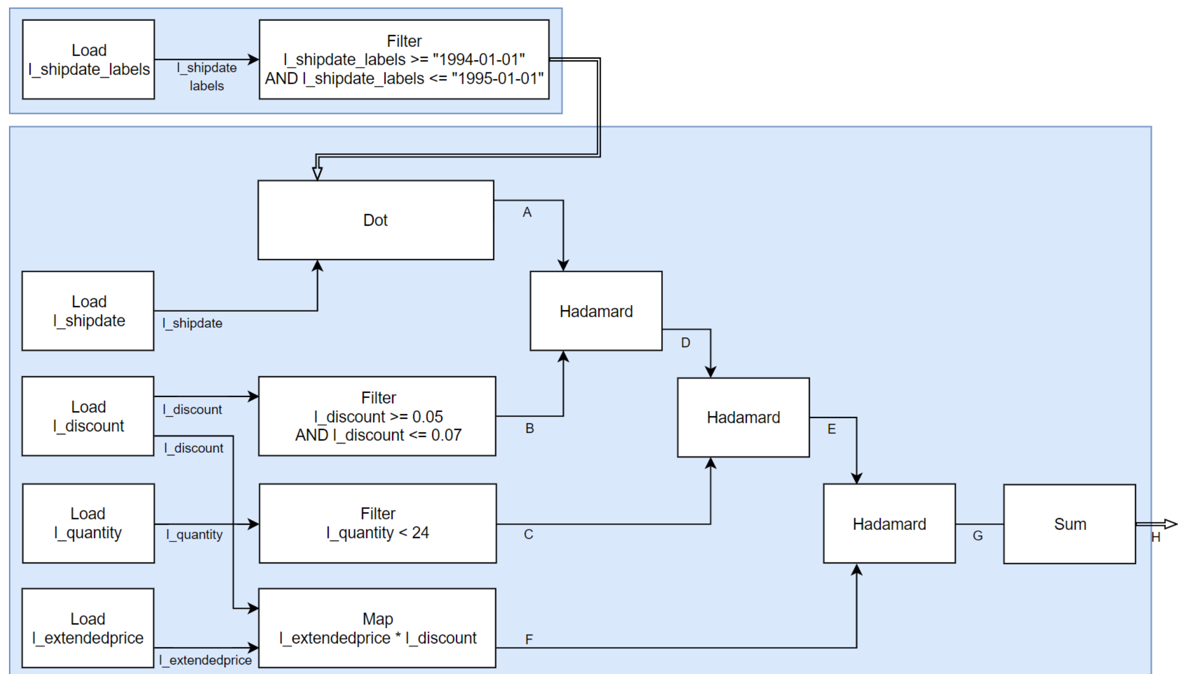


Figure 6.: TPC-H query 6 - execution plan

anywhere in a query but, unless there are subqueries, they are only used at the very end. Also, they only produce this dependency if their result is used as an argument for another operation, thus not having an important role in this initial task scheduling.

Finally, to minimise the used memory, depth-first search will be used. Otherwise, all the load operations that make the leafs of the tree would be performed at the beginning of the query, and all the matrices that have a strong dependency and are positioned at a certain level in the tree would also have to be loaded simultaneously.

3.4 SQL DRIVER

In parallel with the development of this dissertation, another project (Albuquerque and Fernandes, 2018) was run to implement this module. Their foundation was the algorithm presented by Afonso and Fernandes (2017), also described in Section 2.4 and tested against a standard analytical query in Section 2.5.

3.4.1 *SQL Parser*

SQL is a highly complex language at the syntax level. Typical commercial DBMSs take several releases to fully comply with its standards, and many others give up on it, simply picking a subset of the language or using another interface.

This way, the two main concerns with this piece of software were to make it modular, thus easily replaceable in case it does not cover any future requirement, and to define the smallest subset of the language capable of encoding typical analytical queries.

Since the LAQ language has some limitations, for instance only supporting SELECT statements, the used subset of SQL matches this limitations. It is fully depicted in Listing 8.

```
1 SELECT [DISTINCT] select_exp [, select_expr ...]
2 [FROM table_references]
3 [WHERE where_condition]
4 [GROUP BY col_name [, col_name, ...]]
5 [HAVING where_condition]
```

Listing 8: SELECT statement syntax coverage

An important step in the development of the parser was the specification and implementation of a data structure capable of encoding both SQL and LAQ syntaxes, as well as support the conversion process. The structure created by Albuquerque and Fernandes (2018) has two distinct parts, representing both the TD and the logical tree.

Having defined the context-free grammar for the selected subset of SQL, and implemented a structure to store queries, a simple parser was made with Flex and Yacc to make everything work.

3.4.2 *Query Rewriter*

Typical RDBMSs have a query rewriting phase in their optimiser. Its function is to implement rule-based modifications to the input queries, similar to the ones explained in Section 1.2.1, making them run with higher performance.

However, the goal of query rewriting in the TLA approach is not to optimise the query in such a way that it will translate in more efficient machine code, but to simplify the parsed queries, making them more suitable to be encoded in LAQ. Examples of such transformations are, among others, the elimination of some subqueries, solving of operations over constants, and the INTERVAL and BETWEEN statements.

Although shown in the system design architecture, the development of this component is beyond the scope of this dissertation (see Section 5.1.1).

3.4.3 *SQL Converter*

Once the modifications to the SQL parsing tree are complete, it can be converted to LAQ. As stated before, this conversion module was implemented by Albuquerque and Fernandes

(2018), following the algorithm described in Sections 2.4 and 2.5. This algorithm does not cover all the queries in the TPC-H benchmark, thus a further study of the conversion theories is still needed (see section 5.1.1).

Also, due to some implementation issues by Albuquerque and Fernandes (2018), the framework does not cover all the queries supported by the conversion algorithm. However, these queries can be easily translated by hand.

3.4.4 *toString*

This component is a simple piece of software that receives a structure encompassing the LAQ format of a given query, and converts it to a textual format, according to the LAQ specified standards.

There are three ideas supporting the existence of this component instead of proceeding to a direct injection of the LAQ structure in the execution module: (i) to provide the advanced user the possibility of directly writing LAQ queries, eventually more efficient than the ones translated from their SQL counterparts; (ii) the inherent modularity of this architecture, as shown in Figure 3; and (iii) deriving from the easy isolation of components, the establishment of debugging points.

3.5 LAQ ENGINE

The query executor, or LAQ processing engine, is the core of this framework; it receives LAQ queries, efficiently executes them and provides to the user the query results. A detailed description of its main modules follows.

3.5.1 *LAQ Parser*

The actual specification of the LAQ only includes consult queries (SQL `SELECT`s). This narrow usability leads to a simple syntax (Listing 9), ruled by a short CFG.

```
1 identifier = product ( identifier, identifier )
2 identifier = fold ( identifier )
3 identifier = lift ( expression )
4 identifier = filter ( expression )
```

Listing 9: Summarised LAQ syntax (possible statements)

As depicted, a LAQ query consists of a set of operations. When parsed, this set is stored as a graph that has the return statement as the root node.

The query graph is recursively processed from the root to the leafs, powering an efficient out of order execution of the expressions in the query, particularly if they are poorly ordered. However, it is imperative that when an operation is called, all the used variables are already defined (similarly to the C language), otherwise an error may occur.

Finally, it is important to mention that the extension of the language to include, at least, data insertion queries, is fairly simple (see Section 5.1.1).

3.5.2 *Query Optimiser*

Conventional queries encoded in the LAQ format (or in a LA expression) have tree-like dependency graphs, in which the root node of the tree, or any of its sub-trees, needs that all child nodes have completed to start its execution.

Moreover, the commutative and associative properties of LA operations allow the redefinition of the order in the which the operations will be processed, calling for an efficient scheduler of the operations.

Although the implementation of this component was not planned for this dissertation, some efforts have been made to integrate this framework with HEP-frame (Pereira et al., 2016), which includes a scheduler with similar requirements (see Section 5.1.1).

3.5.3 *Query Processor*

The purpose of this module is to convert the structure with the LAQ query into the correspondent C++ code. This code only includes query specific functions, namely the main function and some filters. This way, the LAQ operators and data management functions can be previously compiled, accelerating the query processing phase.

As happens with the LAQ, this intermediate representation creates the opportunity to validate and reuse queries, facilitating the identification of potential problems.

It is also known that compiled languages tend to have better performance results than interpreted ones. As in analytic queries, the time consumed in the initial stages of query processing tends to be negligible when compared to the time used to process the stored data, any small improvements in the latter will compensate the compilation time.

The process of converting the graph with the query to C++ code was divided into six steps of increasing complexity.

1. Load precompiled libraries

Since all the used functions were compiled in advance, their definition should be included at the start of each query.
Listing 10 contains three distinct groups of header files, respectively corresponding to the standard library headers, data manipulation Application Programming Interface (API), and LAQ operators.
Only the first two groups are essential to every query, the functions in the latter should be included as needed.

```
#include <chrono>
#include <iostream>
#include <string>
#include <vector>

#include "include/types.hpp"
#include "include/block.hpp"
#include "include/matrix.hpp"
#include "include/database.hpp"

#include "include/functions.hpp"
#include "include/dot.hpp"
#include "include/filter.hpp"
#include "include/fold.hpp"
#include "include/krao.hpp"
#include "include/lift.hpp"
```

Listing 10: Headers included in C++ queries

2. Define the necessary expressions

   These expressions may include conditional expressions, used on filters, or decimal expressions, used on lifts.

   As depicted in Listing 11, the key here is to replace the variable name with the position it will assume in the array of arguments. This way every conditional/decimal expression has the same definition, which is known by the filter/lift operator.

   For instance, in line 2 the same argument is used in the two predicates of the conditional expression, since it represents a BETWEEN clause.

```
inline bool filter_var_a(std::vector<engine::Literal> args) {
    return args[0] >= "1994-01-01" && args[0] < "1995-01-01";
}
inline engine::Decimal lift_var_f(std::vector<engine::Decimal> args) {
    return args[0] * args[1];
}
```

Listing 11: Example of expressions used in TPC-H query 6

3. Select the database

   After including or defining the necessary auxiliary functions, the implementation of the main function starts. The first line is just to start the timer from the *chrono* library, and its the same in every query, thus it has not been included in the conversion steps.

Then, the database metadata should be loaded, as depicted in Listing 12. The job of specifying the database that should be used is responsibility from the framework manager, to mimic the `USING DATABASE` SQL statement.

```
1 engine::Database db("data/la", "TPCH_1", false);
```

Listing 12: Loading of the database "`TPCH_1`" in read only mode

4. Load the attributes metadata

   Having selected the database that should be queried, the next step is to go through the query graph, and find all the leaf nodes. These are the attributes that will be loaded from the database. For each of them, an empty matrix with its metadata is defined.

   Listing 13 describes the application of such process for the attribute $l_{shipdate}$. Note how the variable from the LAQ is purposely prefixed with "`var_`" to avoid any possible conflict with other variables used by the framework.

```
1 engine::Bitmap *var_lineitem_shipdate =
2     new engine::Bitmap(db.data_path, db.database_name, "lineitem", "shipdate");
```

Listing 13: Loading of the attribute $l_{shipdate}$ metadata

5. Declare temporary matrices

   In a similar way, the variables defined in the LAQ script must be declared. Here the process is not straightforward, since the type of the variable depends on the operation that produces it, as well as on the types of its arguments. Also, the number of blocks of these variables is inherited from these arguments, thus the order of variable declaration is not random as it is on the leaf nodes.

   The solution is to go through the query graph in a bottom up approach, filling the information for each child node before processing their parent node.

   Listing 14 contains the declaration of the variable "a". Note how the matrix type: `FilteredBitvector` can only be inferred by looking at the operation that produces "a": a filter. Also, since the number of blocks is being extracted from the attribute $l_{shipdate}$, it is known that this is the matrix being filtered.

```
1 engine::FilteredBitVector *var_a =
2     new engine::FilteredBitVector(var_lineitem_shipdate->nBlocks);
```

Listing 14: Declaration of the temporary matrix "a"

6. Build the streaming loops

   This piece of code is responsible for loading, processing, and freeing blocks of data, as well as destroying the defined matrices when they are not needed anymore.

   Section 3.3 describes how the operations are ordered in the C++ code. Specifically, Figure 6 is important for the comprehension of this process. For example, the number of blue boxes in the diagram dictates the number of for loops in the C++ query.

   To temporarily encode these loops, a new structure was defined. It is constituted by an array of loops, each of these containing an header, a body, a footer, and two lists with the blocks/variables that should be deleted during/after its completion.

   The header corresponds to the for loops definition, that is, it includes what is between the parenthesis. The body is the code that is repeated. It can be subdivided in three subsections: data loading, executions of one or more operations, and memory release (or data deletion). The footer includes the closing bracket and the code that deletes the variables specified in the two lists.

   After this structure being completely filled, it is iterated and all the code pieces it contains are merged together to form the query result.

3.5.4 *Run-time Compiler*

Many DBMSs perform a run-time compilation of the query source code to achieve better performance results in its execution.

The implemented system will follow the trend, but using the GNU compiler to maintain the simplicity of the component.

3.5.5 *Query Execution*

The presence of this module in the diagram is merely illustrative, since it is restricted to the execution of the previously compiled query program.

3.6 THE FRAMEWORK MANAGER

The purpose of the manager is to provide a simple API to operate the database system. The idea here is that the modularity of the designed architecture implies that a query can be in multiple states, namely SQL, LAQ, C++, binary executable, and textual (its result).

The manager receives as input a query in any non-final state, and produces the desired output. To specify these parameters, the flags `-i` and `-o` should be used. Note that the framework produces all the intermediate representations between the specified ones

So, a simple way of creating, populating and querying a database would be:

1. Write simple C++ scripts to create the database and load the data; it must be in C++, as LAQ does not support yet other SQL statements than the `SELECT`; example: `ladb -i load.cpp -o check_ok.txt`.
2. Write a SQL or LAQ `select` query; for this example SQL was used and it was considered that the user wanted to check the C++ code before using it; example: `ladb -i query.sql -o query.cpp`.
3. Make the desired adjustments, and then compile the query against the framework libraries, or simply use: `ladb -i query.cpp -o query`.
4. Save the query executable anywhere, and reuse it when desired

This module could also include a cache mechanism to avoid repeating the same queries over the same data.

3.7 SUMMARY

This chapter presented an architecture for a framework that encapsulates and complements previous research on analytic data processing in LA. It detailed all framework modules, including the key libraries for matrix manipulation, the description of the SQL driver and the LAQ engine.

4

VALIDATION AND PERFORMANCE RESULTS

SQL is the standard language to operate a DBMS. However, the bulk of database systems is not full SQL compliant, thus restraining the definition of standard tools or methodologies for DBMS validation and comparison.

Moreover, considering the multiple data processing necessities and the way each system was developed to attend specific subsets of them, it is even harder to define generic performance evaluation tools. That is, even when executed under the same conditions, no benchmark can compare all database solutions, as the simple variation of the processed dataset may be enough to reverse the obtained performance results.

The main goal of the Transaction Processing Performance Council (TPC) was to define unbiased database benchmarks to evaluate the system performance. To ensure that this industry standard benchmarks are unbiased and adequate for their systems, several multinational tech companies joined the organisation.

The organisation provides two benchmarks for decision support consult queries, namely TPC-H and TPC Benchmark DS (TPC-DS). Since TPC-H is more OLAP oriented it is used to guide and validate the implemented framework, following the previous approach by Afonso and Fernandes (2017) and Ribeiro et al. (2017).

4.1 TPC BENCHMARK H

The TPC-H benchmark contains the specification of twenty two queries and the database on which they operate. Additionally, it provides the necessary tools to populate the database and metrics to measure the system performance.

Figure 7 represents the used database schema. Although it is not a typical star or snowflake schema, its relations can be divided into measures (*Lineitem*) and dimensions (the other tables).

More than specifying the table names and attributes, this figure provides valuable information about the database size. As one can see, above each table there is a formula to obtain the number of records in each table based on the used scale factor. For example, the

tables *Region* and *Nation* have fixed sizes of 5 and 25, respectively, while the table *Lineitem* has 6 million rows for a unitary scale factor, being the biggest table in the database.

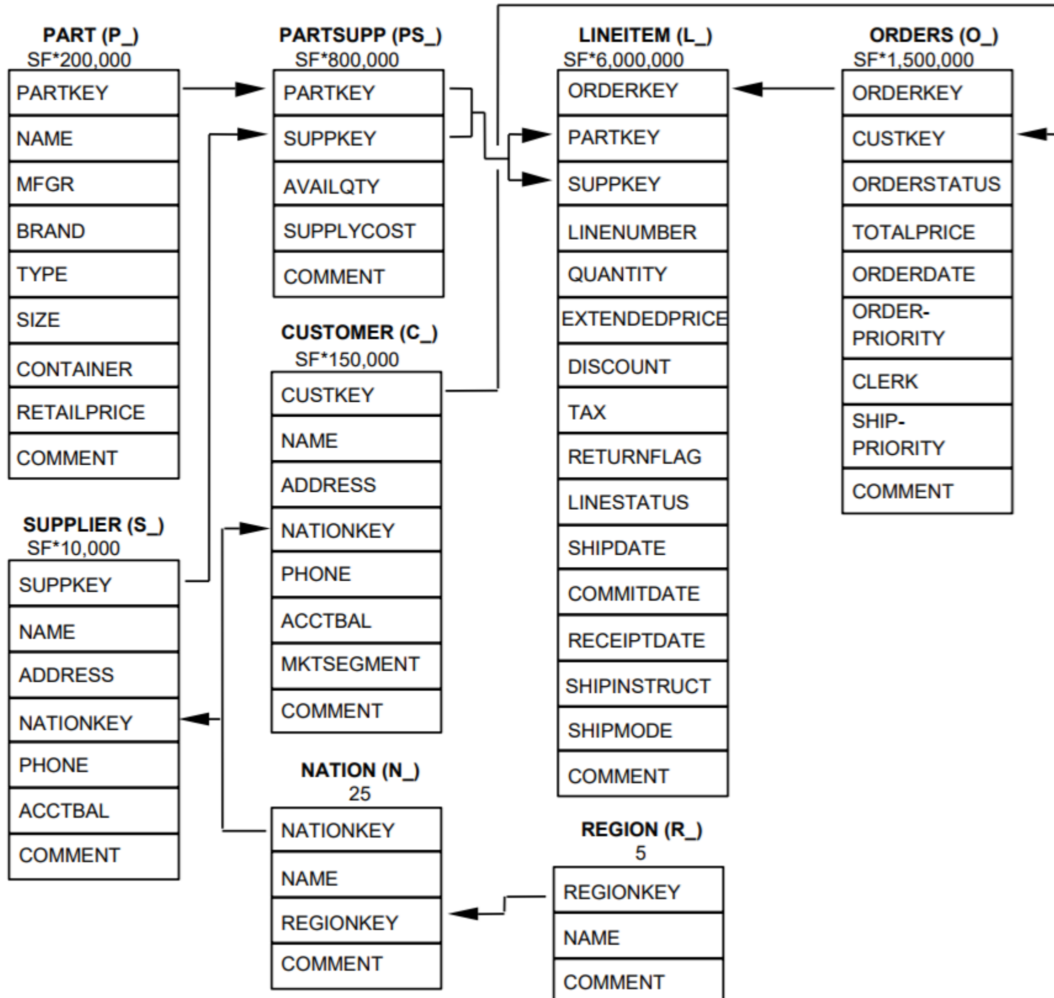


Figure 7.: TPC-H database schema

4.1.1 *Benchmark modifications*

Although TPC-H *Composite Query-per-Hour* is the standard TPC-H performance metric, this is not adequate for a fair comparative evaluation of the LA solution. Since the goal of this

evaluation is to assess the efficiency of the LA approach to process SQL queries on several dataset sizes and only after the datasets are loaded into Random Access Memory (RAM), the execution of multiple concurrent queries is not a suitable metric here.

Instead, we measured, for each query, the execution times for different dataset sizes (from 1 GiB to 64 GiB) and compare the measured figures of the LA approach with those of two open-source competitor database management systems: PostgreSQL and MySQL.

To guide and validate the LA implementation and to compare its performance with competitor solutions, several queries of this benchmark suite were selected:

- to validate the implementation of the most basic filter operations, namely equality, relational, and between - query 6;
- to explore joins and group-by clauses - query 3;
- to test more filters and logical operations, such as `CASE`, `LIKE`, `IN` and `NOT` statements - queries 12 and 14;
- to test sub-queries and filters after group-by (`HAVING`) - query 11;
- to explore semi-joins (`EXISTS`) - query 4.

The SQL and LAQ definition of these queries are in Appendix A.

4.2 TESTBED ENVIRONMENT

Trustworthy experimental results must be reproducible. When these results are code execution times, several runs give a clue on the execution stability. To minimise external unwanted interference, relevance is given to the faster times. The measurements are based on the 3-best runs out of 10, within a 5% max error interval; the best of these 3 is the recorded time.

Table 7 shows the key features of the testbed environment.

To ensure fairness, each competitor DBMS was properly configured with support from their technical staff.

MySQL has three available storage engines: MyISAM and InnoDB - the most common ones, and MEMORY (HEAP). Since the goal in this comparative evaluation is to measure in-memory performance, the MEMORY (HEAP) alternative was chosen.

PostgreSQL has no storage engine equivalent to MEMORY (HEAP) in MySQL; a fair measure of execution times requires two runs to warm up the cache (DB-cache in RAM) before the 10 runs.

The recommended size for the shared buffers is 1/4 of the RAM size, but this value was set to 3/4 to ensure that all queries data could fit in these buffers.

#Processing Unit (PU)-chips	2
Model	Intel Xeon E5-2683v4
Base clock freq	2.10 GHz (up to 3.00 GHz)
#PU cores	2 x 16 (2-way SMT support)
L1 cache	2 x 16 x 32 KiB
L2 cache	2 x 16 x 256 KiB
L3 cache	2 x 40 MiB
RAM	2 x 128 GiB (NUMA)
OS	CentOS 6.3
PostgreSQL	V. 10.2
MySQL	V. 5.7

Table 7.: Testbed environment

4.3 RESULTS AND DISCUSSION

Ribeiro et al. (2017) have already implemented and tested hard coded versions of TPC-H queries 3 and 6. Since their report contains a full coverage of the used test environment, this chapter is mainly addressed to the relevant issues, namely how this new approach managed to reach even better performance outcomes than the PostgreSQL engine (the preliminary results from the previous version are in Figure 8).

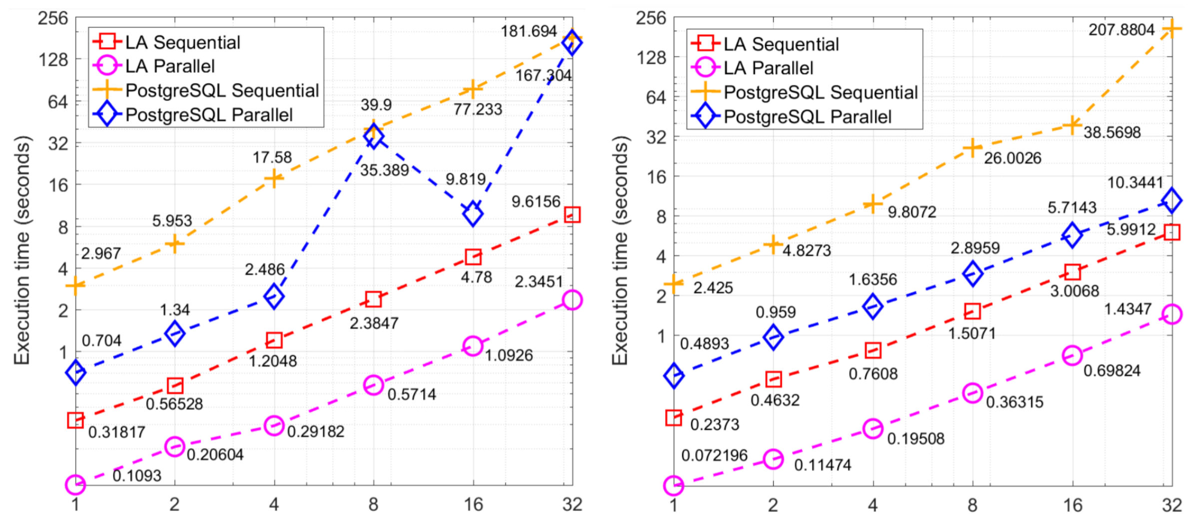


Figure 8.: TPC-H queries 3 (left) and 6 (right) – preliminary performance results for scale factors from 1 to 32 (Ribeiro et al., 2017)

The scale factors have a close relationship with the dataset size (in GiB): the highest scale factor in the preliminary tests was 32, corresponding to 192 million rows in the table *Lineitem*.

The plots in Figure 9 show the measured execution times with the upgraded sequential versions for two queries, 3 and 4, where the scale factors were extended to 64. In these larger datasets, the LA solution produces the expected results: 2 and 4 times the resources used by the scale factor 32 version. No error metric is displayed in each data-point since it is too small, staying hidden in most cases.

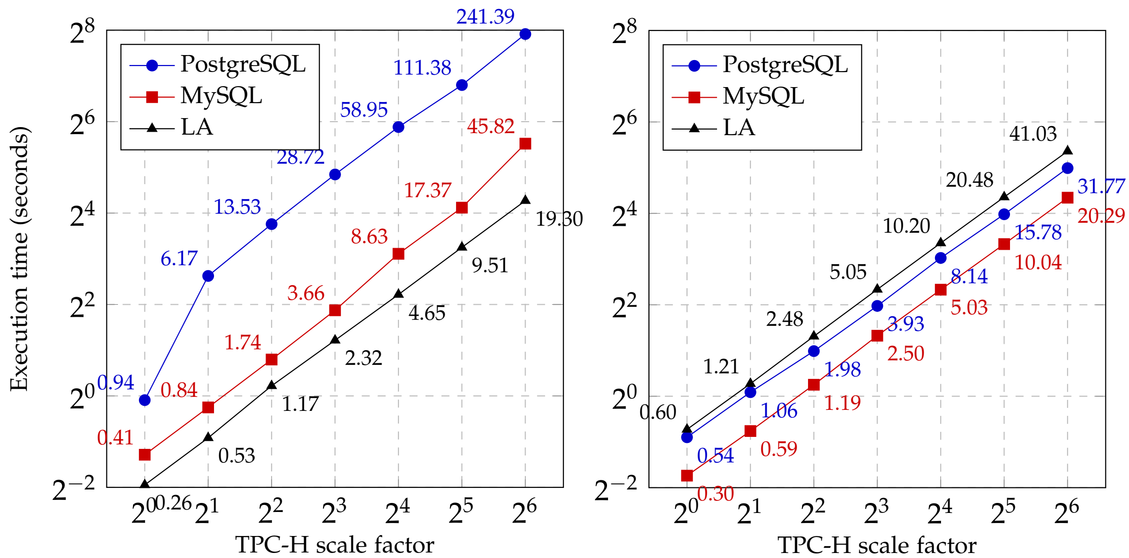


Figure 9.: TPC-H queries 3 (left) and 4 (right) performance results, sequential versions

Comparing these results with the ones obtained by Ribeiro et al. (2017) a slightly performance increase can be observed. Despite the changes in the implementation of some LAQ operators, this difference can be justified with the utilisation of distinct hardware to run the benchmarks. Also, in the previous tests, the PostgreSQL took some poor decisions in the selection of the query plans for the parallel tests. As shown in Figure 9, for the scale factors 8 and 32, the parallel execution time is almost the same as the sequential. This issue was solved by updating to the latest version of the DBMS.

The LA approach is clearly the fastest in query 3 for all dataset sizes, while the redundant filtering operations in query 4 take over 60% of the overall execution time, considerably degrading its performance (the same happens in query 12).

Figure 10 plots the execution times and memory usage for the sequential version (single-threaded) of six selected queries from the TPC-H benchmark suite, with scale factor 32, and for each of the three competing environments, PostgreSQL, MySQL and the LA approach. Since the 6 tested queries display almost the same linear behaviour across all scale factor values, there is no need to display their plots.

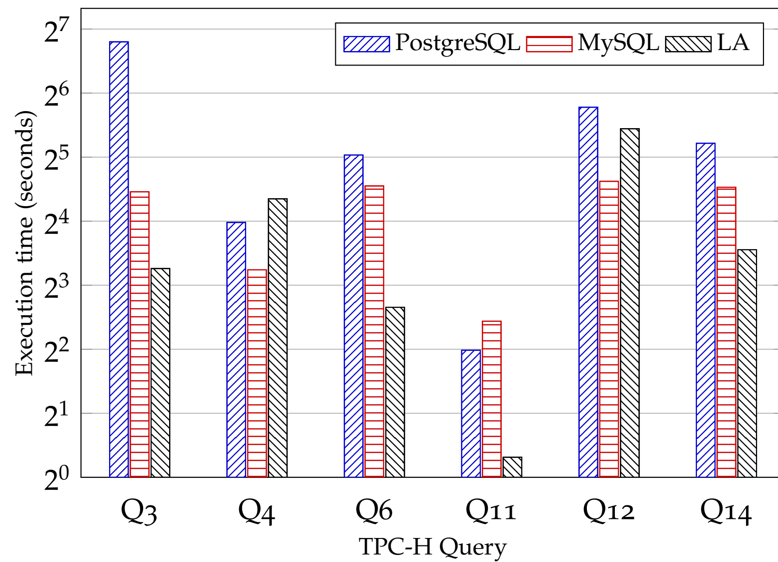


Figure 10.: Execution times (scale factor: 2^5)

The LA approach has proved to be faster than its competitors, in its current prototype version; it only has a lower performance in queries 4 and 12 due to redundant filtering (this can be solved, see Section 5.1.1).

The use of column-oriented tables in the LA approach (attribute oriented) gives advantages over the competitors, namely because:

- the columnar approach loads less data;
- it avoids operations that implicitly require a row orientation (namely, converse and matrix composition); thus the overall emphasis on the Khatri-Rao product;
- measurements are incorporated as late as possible, taking advantage of using boolean matrices as long as possible;
- it saves one matrix composition by equi-join, due to the smart encoding of primary keys.

Performance can be further improved if the available cores in the PU-chips are adequately used: while MySQL explores parallelism by concurrently processing multiple queries, PostgreSQL can use in parallel up to all available cores to process any query, and each kernel operation in LA approach can also use up to all available cores.

Figure 11 compares the single and multi-threaded versions of query 6 in the LA approach with the PostgreSQL versions, using up to all available cores in the server. MySQL was excluded from this comparison since it has no parallel version of a single query.

As expected, both parallel versions run consistently faster than the corresponding sequential ones, and the gain is lower when the dataset size is small.

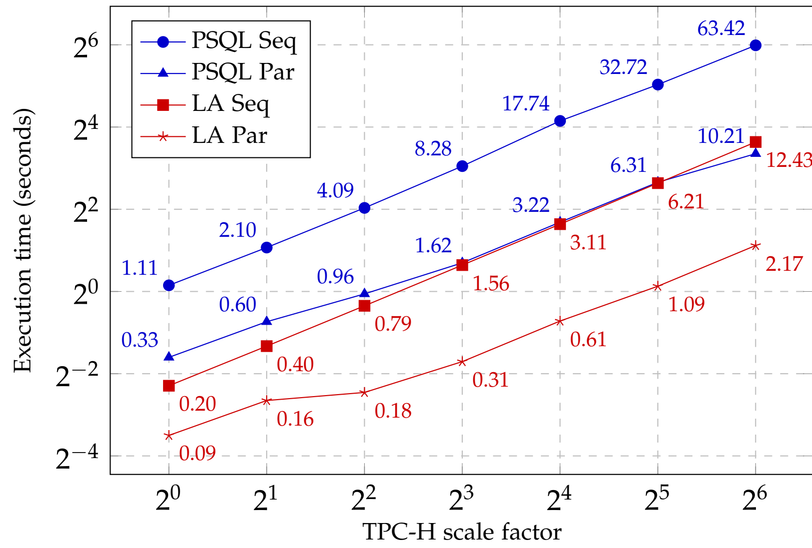


Figure 11.: Sequential and Parallel Query 6

However, the parallel efficiency in both systems is quite low: using 32 cores, only the larger scale factor managed to reach 6x speedup.

In some queries the parallel version of PostgreSQL has unstable behaviour for larger dataset sizes; for instance, in query 3 its query planner fails in such a way that the parallel execution times in datasets larger than 16 are longer than the sequential version.

Overall, the LA approach shows again its parallel superiority against PostgreSQL and both versions (sequential and parallel) have a consistent behaviour through all queries.

Figure 12 shows the maximum RAM space required for each of the sequential versions of the six queries in each DBMS (using the 3-worst case out of 10), for the scale factor 32, and measured using `dstat`.

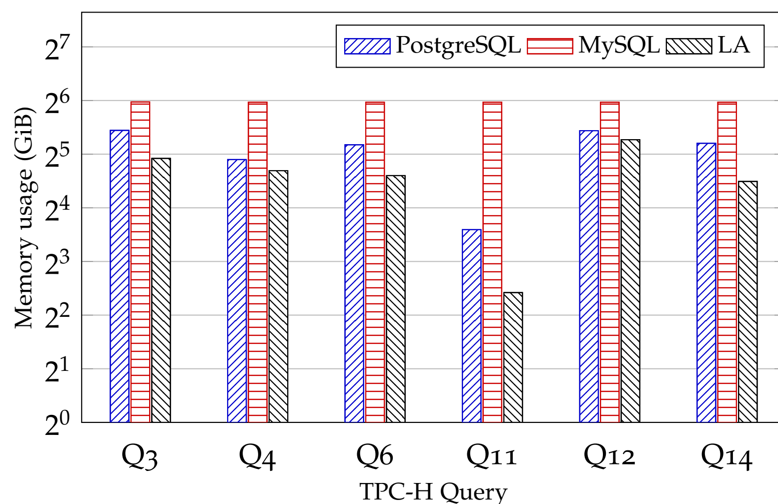


Figure 12.: Memory usage (scale factor: 2^5)

Code efficiency is also related to the required memory to run each query. The three systems follow different approaches: while PostgreSQL loads blocks of data in RAM, keeping those that it may need (disk cache), all data in MySQL is directly inserted in RAM (in these tests). The LA approach only places in RAM the attributes it will need.

This plot clearly shows that the LA approach is very efficient in managing the used RAM and these figures can be further improved.

4.4 SUMMARY

This chapter described the process conducted to measure the performance of the framework. Starting with the selection of processing time and used memory in TPC-H queries as the standard measure, it then moved to a complete description of the computing environment in which the tests were run. Finally, the results of the comparison with PostgreSQL and MySQL were presented and discussed.

5

CONCLUSIONS

The promising results of recent attempts (Afonso and Fernandes, 2017; Ribeiro et al., 2017) to prove the better efficiency of LA queries in comparison to its relational counterparts have opened an interesting possibility for the development of a consolidated database system. This dissertation was set up to complete the first steps on the development of such system.

After working in Afonso and Fernandes (2017), the author was already up-to-date with the state of the art on LA querying, only requiring a brief research on the architecture of some database systems (MySQL, PostgreSQL, Apache Drill, Apache Kylin, ...), essential to sketch the framework described in this document.

The first step to implement this architecture was the definition of an efficient format for data encoding. Since the database dimensions are stored in bitmaps (matrices containing either zeros or ones), and the number of zeros is orders of magnitude higher than the number of ones, dense representations were immediately excluded. Three formats for sparse matrix encoding have been analysed: LIL, COO, and CSR/CSC.

LIL needs an array with so many elements as the number of rows in the matrix it encodes. Considering that in some cases the number of rows is larger than the range of a long integer, this format wastes too much memory after some Khatri-Rao operations. COO is the most memory efficient of the studied formats. However, it does not support direct accesses to the data it encodes, increasing the complexity of some operations from $\mathcal{O}(NNZ)$ to $\mathcal{O}(NNZ^2)$. The difference between CSR and CSC is the presence of an array with the cumulative number of elements in each row/column. This way, CSR suffers from the same problems as LIL, thus CSC was the chosen format for data encoding.

A new DSL was defined, LAQ, to operate these matrices towards the query result. It contains three LA operators: the Hadamard, Khatri-Rao, and dot products, as well as three derived ones, filter, fold, and lift operators.

To implement the first ones, the data properties were analysed: the used matrices are always functional, that is, they have at most one element per column. By implementing the algorithms with this property in consideration, these operations perform much faster than the generic implementations of general purpose LA libraries, such as CBLAS and Intel MKL.

The second group of operators derive from the need to express the complex logical expressions from the SQL `WHERE` clauses, as well as the arithmetic expressions and aggregations in the `SELECT` statement. Although most efforts were on the optimisation of these operators, they still are the ones with more room for performance improvements.

While developing the LAQ engine, more specifically the LAQ to C++ converter, the author worked as an advisor for the SQL to LAQ converter project (Albuquerque and Fernandes, 2018) and also in the writing of the paper "Typed Linear Algebra for Efficient Analytical Querying" (Afonso et al., 2018).

Regarding the LAQ engine, a simple parser for the language was implemented, together with a module that converts the parsed queries in C++ scripts. This generated pieces of code are then linked to the developed kernel, and executed to produce the query results.

To validate the results of the developed framework, some research was done in the identification of a widely accepted OLAP benchmark. As a result, TPC-H was chosen. This benchmark is composed by twenty two queries with distinct complexities. Among them, some are not yet supported due to exceptions in the multiple modules of the framework. The supported queries are 1, 3, 6, 12, 14, and 19; the unsupported ones are:

- query 2: the query does not follow the OLAP rules - any attribute in the `SELECT` statement must be in the `GROUP BY` or under an aggregation function;
- queries 4, 7, 8, 9, 11, 13, 15, 16, 17, 20, 21, and 22: the subqueries are not yet supported by the SQL to LAQ conversion algorithm; however, it is possible to write scripts in LAQ to evaluate the query result and performance, as happened in queries 4 and 11;
- query 5: the TD produced by the query has a circular pattern unsolvable by the conversion algorithm;
- queries 10 and 18: the number of subsequent Khatri-Rao products creates matrices with so many rows that its indexes cannot be represented within the range of a long integer; the use of multi-precision libraries as GNU Multi Precision (GMP) (Granlund and Team, 2015) was considered, but not explored.

After ensuring the correct results in the executed queries, the performance of the LAQ engine was compared with two conventional row-oriented DBMSs: PostgreSQL and MySQL. The LA solution outperformed both systems in most queries. The causes for the lower performance in the other ones can be overcome, as explained in Section 5.1.1.

Beyond the original planned work was a comparative performance evaluation with a column-oriented systems. MonetDB (Idreos et al., 2012) was selected as a representative column-oriented DBMSs. The obtained performance results are presented and discussed in Section 5.1.1.

Being the first prototype, current version of the LAQ engine is still far from a robust solution to be deployed. Even though, the proposed modular architecture allows an easy isolation of the framework components, making it easier to solve its main issues. Moreover, detailed suggestions to improve most modules, and the framework as a whole, are presented in the following section.

5.1 FUTURE WORK

As stated in the title of this dissertation, its main goal is to make considerable advances in the functionality and efficiency of a LA based DBMS. This section includes an analysis of some required steps to further improve and complete the framework, as well as a discussion of other relevant features the framework may include.

5.1.1 *Framework extensions*

Query Rewriter

The high complexity of the SQL language and the lack of similarities with the LAQ, result in the difficult conversion between both languages. However, some parts of a SQL query can be simplified before the conversion, thus creating a new SQL query that produces the same result.

A further study on what can be simplified is definitely required, but some aspects can already be pointed. They are:

- The computation of constant arithmetic expressions

  Consider the conditional statement `WHERE var < 3+5`. The simple identification of the sum of the two constants and its resolution ($3 + 5 = 8$) means that the conversion algorithm is not required to have an arithmetic calculator. Also, if the sum was kept until the execution phase, it would have been redundantly calculated for each comparison.

- The `interval` statement

  The `interval` is a special case of a constant expression, as it is used to operate on dates. It can be solved in a similar way, having `'1995-03-15' + interval '1' year = '1996-03-15'`.

- The `between` statement

  Mostly used to shorten the spelling of two logical expressions defining an interval, the `between` can be converted to those expressions: `var between 5 and 6` is equivalent to `var >= 5 and var <= 6`.

- Sub-queries

  A more extensive study of these queries is still required. Sub-queries that match the typology `SELECT FROM SELECT` can usually be independently calculated and then used as any normal table. However, sub-queries with the typology `SELECT WHERE SELECT` usually cannot be isolated from the main `SELECT`.

Query Optimiser - Operation Scheduler

It is known that some operations reduce more data than others, speeding up the subsequent ones, since fewer elements have to be processed.

DBMSs like PostgreSQL take into consideration the impact of the data in the final query plan, taking a sample of that data, and optimising the plan accordingly to the statistical analysis of the sample. This initial analysis is condemned to be in an eternal threshold between its quality and the time it consumes.

Here is where a dynamic query plan can be beneficial. A powerful scheduler like the one in HEP-Frame (Pereira et al., 2016) can be used to reorder operations in real-time, ensuring that the ones that filter more data are executed first.

Moreover, data loads may or not be performed in parallel with data processing. The HEP-Frame scheduler can dynamically give some control to these decisions that are currently left to the Operating System (OS).

Query Optimiser - Filter Redundancy

Currently, filters are one of the most time consuming operations in the LA approach, and that can be easily explained. When in the relational approach a tuple does not satisfy a predicate filtering any of its attributes, the entire tuple is removed and is not processed by the subsequent filtering operations. However, using the standard LA methodology, all the filters would be independently calculated and then merged: all table records would be iterated as many times as the number of filters in a given query.

There is a simple way to overcome this design problem: iterate any previously calculated filter, and for each element it has, validate the specified condition in the unfiltered attribute at the position dictated by that element.

This optimisation would have perfect results if matrices were encoded in COO. However, as CSC was chosen, the column pointer array still has to be iterated and compared. The overall complexity remains, except the time to process a single unit, which is reduced, since function calls are avoided. Regarding dimensions, numbers are compared instead of strings.

A key point in this issue is the extensive study of columnar DBMSs like MonetDB. As they also separate the attributes of each table, the algorithms they use to avoid making redundant filtering can be extrapolated to the presented framework.

LAQ specification

Currently the LAQ only covers consult queries. The inclusion of already implemented methods like the SQL COPY FROM and INSERT INTO is not complex. New data structures can be designed with the specific purpose of covering these new types of queries, without interfering with the already implemented ones.

Also, some new operations have been added in the most recent benchmarks. One example is the unvec used to improve the performance of query 4 (see Appendix A.2)

Regarding the CFG, it is necessary to add the new projections in the root node, leading to these types of queries instead of the consulting ones, and then include as many projections as needed below them.

SQL converter

The implemented algorithm was the one proposed by Afonso and Fernandes (2017), but since it does not cover all the queries in the TPC-H benchmark, it needs to be upgraded.

As mentioned by the authors in their future work section, the two major obstacles to be overtaken are the specification of a format for the output matrix, and the conversion of queries with multiple SELECT statements (sub-queries) to the equivalent LAQ scripts.

Starting with the results matrix, some progress has been done as the numerical aggregations can already be shown. However, it is still necessary to implement the reconstruction of the dimension attributes, which needs the inclusion, in each matrix, of a variable containing its type.

Consider the queries that match the SELECT ... WHERE ... SELECT format. In these, the sub-query cannot be replaced by a single value for a given state of the database. They instead must be calculated for each element in the other argument of the conditional predicate it belongs to.

Although the description of this process calls for a row-wise approach, it can be achieved by calculating a new vector with the results for all the possible calls. The complexity inherent to its implementation is the retrieval of data from the inner SELECT statement and its usage as an argument of a predicate in the specific conditional clause.

Moreover, query 2 does not comply with the OLAP rules, since it has attributes in the SELECT statement that are not specified in the GROUP BY, neither involved by an aggregation function. Since this is one algorithm requirement, this query was not translated yet.

Finally, the TD of query 5 has an unusual cyclical pattern in the TD, due to the way the joins are organised when building the TD. This cycle created an infinite loop in the used algorithm, making the query nonconvertible.

Overall performance

For a comparative evaluation of the performance of the implemented framework we selected the two most popular open-source RDBMSs: PostgreSQL and MySQL. However, columnar database engines tend to perform at least an order of magnitude faster. For a fairer comparative evaluation of the selected benchmarks we also included the MonetDB, following the methodology used with PostgreSQL, going beyond the goals defined for this dissertation.

Figures 13 and 14 show execution times and memory usage of the LA framework and MonetDB, which shows that our framework is currently outperformed by MonetDB.

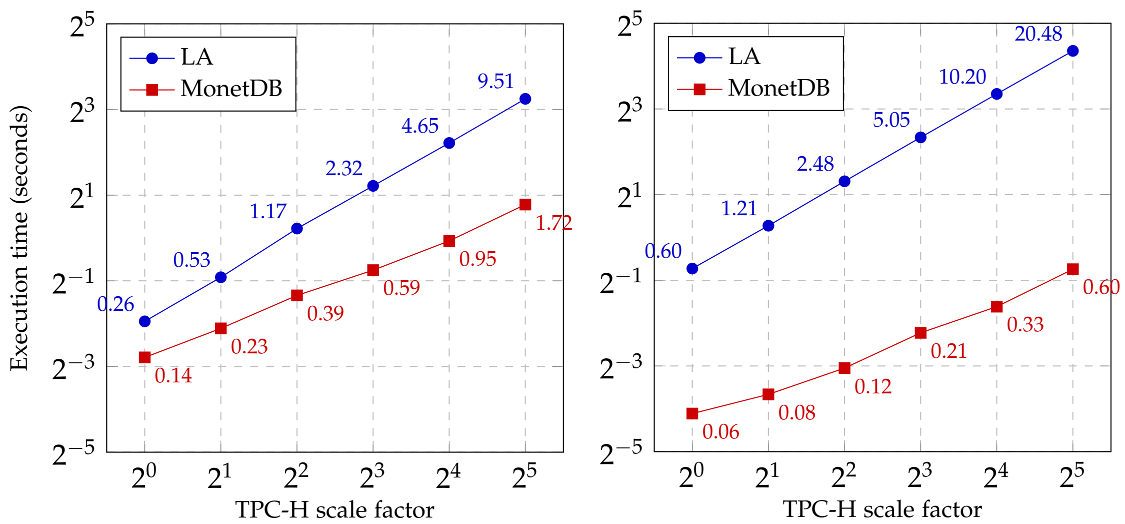


Figure 13.: TPC-H queries 3 (left) and 4 (right) performance results, sequential versions

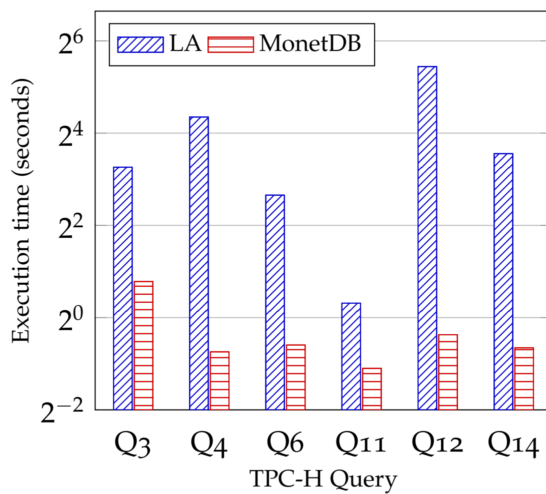


Figure 14.: Execution times (scale factor: 2^5)

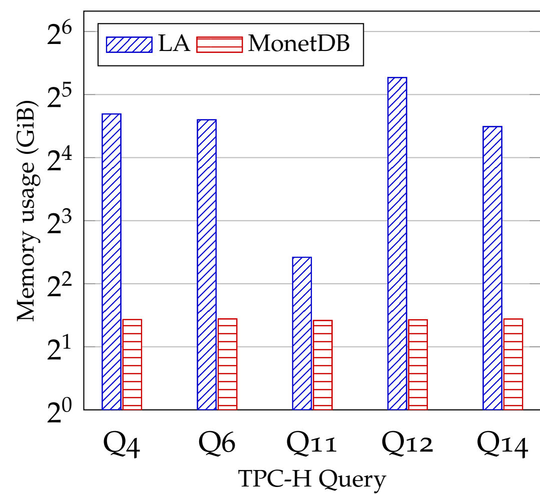


Figure 15.: Memory usage (scale factor: 2^5)

We are aware that MonetDB had several years of performance tuning and optimisation and we expect that with further performance enhancements the LAQ operations can perform faster. However, the better than linear scalability of MonetDB raises some concerns on the validity of the obtained times. For instance, in query 4, an increase 32x on the dataset size represents only an increase of 10x in the computation time.

Another issue is the similarity in the memory usage of all tested queries when compared to the LA solution (Figure 15). MonetDB internals must also be studied to ensure that no intermediate computations are being stored and used in subsequent tests.

5.1.2 *Horizontal scalability*

The implemented LAQ operations were optimised to minimise the impact of data dependencies. This way, the migration from thread level to process level parallelism is simpler.

Even though, when implementing the streaming approach across multiple computing nodes, the data dependency problems will be intensified by communication overheads, calling for specific algorithms. These algorithms have not been implemented yet, although some possible approaches have already been analysed.

Embarrassingly parallel operators

Operations like the Khatri-Rao, filter or lift can be executed block-wise, and this aspect remains when following the distributed memory paradigm. Each process is able to load a block of each argument, producing the respective block of the output matrix, without communicating with any other process. This is another clear opportunity to improve the overall performance in OLAP transactions.

Dot product

In Section 3.2.3, the dot product $C = A.B$ needs the full matrix A to compute each C block. Figure 16 shows a way to overcome this limitation, by replicating matrix A across all nodes: these nodes could then independently process a group of blocks.

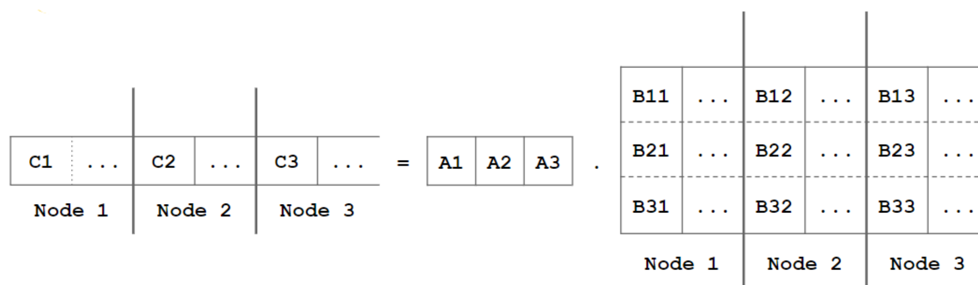


Figure 16.: Distributed dot product - replicate A

Another alternative is to also divide the matrix A among all computing nodes (as in Figure 17), rotating it across all nodes in multiple iterations: the used memory is reduced and each node has access to all data in matrix A.

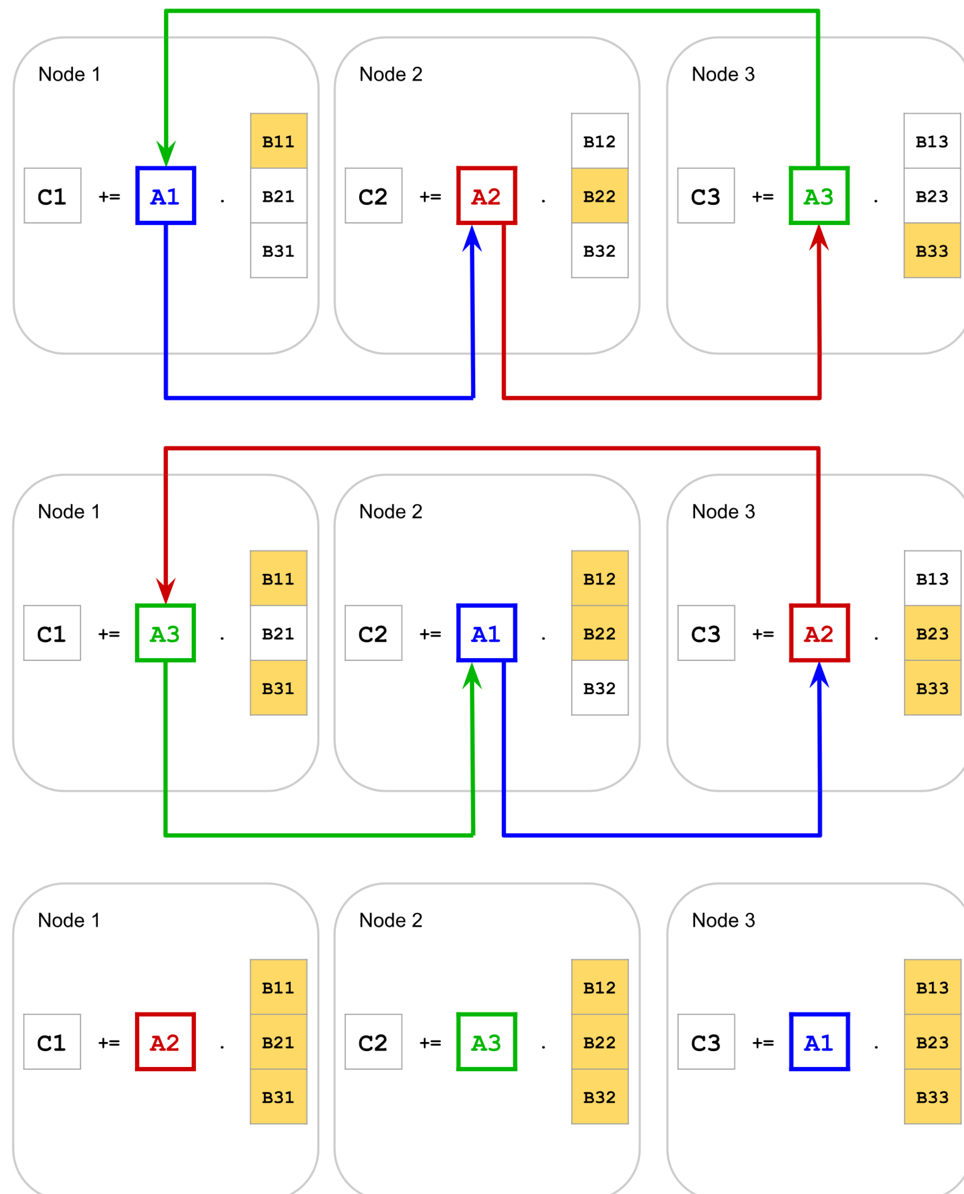


Figure 17.: Distributed dot product - rotate A

The problem with this approach is that there will be as much data rotations among all nodes as the number of blocks in the matrix B, possibly causing communication overheads.

An alternative would be to keep matrix A steady and make all nodes compute the same block of C. This solution follows the map-reduce pattern since after processing the blocks (map) it requires a reduction (matrix sum) of the block parts obtained in each node.

Fold

A possible distributed memory implementation of the fold operator has two distinct phases. First, each node should aggregate its group of blocks, storing the results as done in shared memory. Then, the information obtained should be condensed in a single structure.

As the fold operator converts the accumulator structure to a matrix, the aggregation process can be made over any of these formats. The goal should be to break the matrix (vertical vector) in the same positions across every node, distribute its content accordingly, and make each node complete the aggregation for a set of rows.

5.1.3 *Incremental querying*

With the rise of big data, ever-growing, unbounded datasets are more and more common.

The querying of such data can be processed much more efficiently by using intermediate tables, where only the new data (inserted after the last query) is processed, rather than processing the whole datasets every time.

During this dissertation work, some research was done to understand the complexity of implementing such concepts on top of the developed framework.

The research also followed a specific syntax, where $[A|\delta A]$ represents the matrix A after some data have been inserted, that is, $A + \delta A$. The remaining syntax is extracted from the LAQ, with the exception that the result of the function may not be assigned to a variable, but used as an argument to another function. Following these guidelines, Listing 15 represents the incremental version of the TPC-H query 3.

```
1  δA = dot( filter( [c_mktseg_labels | δc_mktseg_labels] ), δc_mktseg )
2  δB = dot( [A | δA], δo_custkey )
3  δC = dot( filter( [o_orddate_labels | δo_orddate_labels] ), δo_orddate)
4  δD = krao( δB, δC )
5  δE = krao( δD, δo_orderdate )
6  δF = krao( δE, δo_shippriority )
7  δG = dot( [F | δF], δl_orderkey )
8  δH = dot( filter( [l_shipdate_labels | δl_shipdate_labels] ), δl_shipdate )
9  δI = krao( δG, δH )
10 δJ = krao( δl_orderkey, δI )
11 δK = lift( δl_extendedprice * (1 - δl_discount) )
12 δL = krao( δJ, δK )
13 δM = sum( δL )
14 new_M = add(M, δM)
15 return( new_M )
```

Listing 15: Incremental TPC-H query 3 - update after data insertion

Comparing this query with the streaming approach used in the framework, it is unequivocal that the deltas can be seen as blocks, thus ruled by the weak data dependencies. Also,

the necessity of processing the full matrix occurs in the same conditions as the strong dependencies in normal queries.

Note that if data is only inserted in the fact table, thus avoiding dot products, the query can be resolved without consulting old data.

This way, it is clean that a future incremental engine can be implemented with few modifications to the proposed system.

Summing up, from the improvement of the developed framework to its extension to other fields, it is clear that the possibilities for future work are very broad, some of them being already on-going.

BIBLIOGRAPHY

João Afonso and João Fernandes. Towards an efficient Linear Algebra encoding for OLAP. Project report, DI, University of Minho, July 2017.

João Afonso, Gabriel Fernandes, João Fernandes, Filipe Oliveira, Bruno Ribeiro, Rogério Pontes, José Oliveira, and Alberto Proença. Typed Linear Algebra for Efficient Analytical Querying. *ArXiv e-prints*, September 2018.

Luís Albuquerque and Rafael Fernandes. Converting SQL into a Linear Algebra DSL. Project report, DI, University of Minho, July 2018.

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

Edgar Codd. A database sublanguage founded on the relational calculus. In *SIGFIDET Workshop*, pages 35–68, San Diego, California, November 1971. ACM.

Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management: Global Edition*. Always Learning. Pearson Education Limited, Harlow, Essex, England, 6th edition, September 2014.

John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, December 2012.

Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, United Kingdom, 2015.

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.

William Inmon, Claudia Imhoff, and Ryan Sousa. *Corporate information factory*. John Wiley & Sons, 2002.

Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

Peter Lake and Paul Crowther. *Concise Guide to Databases: A Practical Introduction*. Undergraduate Topics in Computer Science. Springer, 1st edition, 2013.

Hugo Macedo and José Oliveira. A linear algebra approach to olap. *Formal Aspects of Computing*, 27(2):283–307, Mar 2015.

Hector Molina, Jeffrey Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2nd edition, 2008.

Filipe Oliveira and Sérgio Caldas. Optimisation of a Linear Algebra approach to OLAP. 2016.

José Oliveira and Hugo Macedo. The data cube as a typed linear algebra operator. In *Proceedings of The 16th International Symposium on Database Programming Languages*, DBPL '17, pages 6:1–6:11, New York, NY, USA, 2017. ACM.

André Pereira, António Onofre, and Alberto Proença. Tuning pipelined scientific data analyses for efficient multicore execution. In *Proc. Int. Conf. High Performance Computing and Simulation, HPCS 2016*, pages 751–758, 2016.

Rogério Pontes. Benchmarking a Linear Algebra Approach to OLAP. Master's thesis, University of Minho, December 2015.

Bruno Ribeiro, Fernanda Alves, and Gabriel Fernandes. HPC OLAP queries powered by a linear algebra representation. Project report, DI, University of Minho, July 2017.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

Michael Stonebreaker, Joseph M. Hellerstein, and Peter Bailis. *Readings in Database Systems*. www.redbook.io, 5th edition, 2015.

TPC-H QUERIES - SQL AND LAQ VERSIONS

A.1 QUERY 3

```
1  select
2      l_orderkey,
3      sum(l_extendedprice * (1 - l_discount)) as revenue,
4      o_orderdate,
5      o_shippriority
6  from
7      customer,
8      orders,
9      lineitem
10 where
11     c_mktsegment = ':1'
12     and c_custkey = o_custkey
13     and l_orderkey = o_orderkey
14     and o_orderdate < date ':2'
15     and l_shipdate > date ':2'
16 group by
17     l_orderkey,
18     o_orderdate,
19     o_shippriority
20 order by
21     revenue desc,
22     o_orderdate;
```

```
1  A = filter( customer.mktsegment == "MACHINERY" )
2  B = dot( A, orders.custkey )
3  C = filter( orders.orderdate < "1995-03-10" )
4  D = hadamard( B, C )
5  E = krao( D, orders.orderdate )
6  F = krao( E, orders.shippriority )
7  G = dot( F, lineitem.orderkey )
8  H = filter( lineitem.shipdate > "1995-03-10" )
9  I = krao( G, H )
10 J = krao( lineitem.orderkey, I )
11 K = lift( lineitem.extendedprice * (1 - lineitem.discount) )
12 L = krao( J, K )
```

```
13 M = sum( L )
14 return( l_orderkey, M, o_orderdate, o_shippriority )
```

A.2 QUERY 4

```
1  select
2      o_orderpriority,
3      count(*) as order_count
4  from
5      orders
6  where
7      o_orderdate >= date ':1'
8      and o_orderdate < date ':1' + interval '3' month
9      and exists (
10         select
11             *
12         from
13             lineitem
14         where
15             l_orderkey = o_orderkey
16             and l_commitdate < l_receiptdate
17     )
18 group by
19     o_orderpriority
20 order by
21     o_orderpriority;
```

```
1  A = filter( lineitem.commitdate < lineitem.receiptdate )
2  B = krao( lineitem.orderkey, A )
3  C = filter( orders.orderdate < "1993-07-01" )
4  D = filter( orders.orderdate >= "1993-07-01" )
5  E = hadamard( C, D )
6  F = krao( orders.orderpriority, E )
7  G = dot( F, B )
8  H = krao( B, G )
9  I = avg( H )
10 J = unvec( I )
11 K = count( J )
12 return( orders.orderpriority, K )
```

A.3 QUERY 6

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date ':1'
    and l_shipdate < date ':1' + interval '1' year
    and l_discount between :2 - 0.01 and :2 + 0.01
    and l_quantity < :3;
```

```
A = filter( lineitem.shipdate >= "1994-01-01" AND lineitem.shipdate < "1995-01-01" )
B = filter( lineitem.discount >= 0.05 AND lineitem.discount <= 0.07 )
C = hadamard( A, B )
D = filter( lineitem.quantity < 24 )
E = hadamard( C, D )
F = lift( lineitem.extendedprice * lineitem.discount )
G = hadamard( E, F )
H = sum( G )
return ( H )
```

A.4 QUERY 11

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'GERMANY'
group by
    ps_partkey having
        sum(ps_supplycost * ps_availqty) > (
            select
                sum(ps_supplycost * ps_availqty) * 0.0001
            from
                partsupp,
                supplier,
                nation
            where
```

```
22                  ps_suppkey = s_suppkey
23                  and s_nationkey = n_nationkey
24                  and n_name = 'GERMANY'
25          )
26  order by
27      value desc;
```

```
1   A = filter( nation.name = "GERMANY" )
2   B = dot( A, supplier.nationkey )
3   C = dot ( B, partsupp.suppkey )
4   D = lift( partsupp.supplycost * partsupp.availqty )
5   E = hadamard( C, D )
6   F = sum( E )
7   G = map( F * 0.0001 )
8   H = krao( partsupp.partkey, C )
9   I = krao( H, D )
10  J = sum( I )
11  K = filter( J > G )
12  L = hadamard( J, K )
13  return( partsupp.partkey, L )
```

A.5 QUERY 12

```
1   select
2       l_shipmode,
3       sum(case
4           when o_orderpriority = '1-URGENT'
5               or o_orderpriority = '2-HIGH'
6               then 1
7           else 0
8       end) as high_line_count,
9       sum(case
10          when o_orderpriority <> '1-URGENT'
11              and o_orderpriority <> '2-HIGH'
12              then 1
13          else 0
14      end) as low_line_count
15  from
16      orders,
17      lineitem
18  where
19      o_orderkey = l_orderkey
20      and l_shipmode in (':1', ':2')
21      and l_commitdate < l_receiptdate
22      and l_shipdate < l_commitdate
23      and l_receiptdate >= date ':3'
24      and l_receiptdate < date ':3' + interval '1' year
```

```
25 group by
26     l_shipmode
27 order by
28     l_shipmode;
```

```
1  A = filter( lineitem.shipmode IN ("MAIL", "SHIP") )
2  B = filter( l_commitdate < l_receiptdate )
3  C = filter( l_shipdate < l_commitdate )
4  D = filter( l_receiptdate >= "1994-01-01" AND l_receiptdate < "1995-01-01" )
5  E = hadamard( C, D )
6  F = hadamard( B, E )
7  G = hadamard( A, F )
8  H = krao( l_shipmode, G )
9  I = dot( orders.orderpriority, lineitem.orderkey )
10 J = filter( I = "1-URGENT" )
11 K = filter( I = "2-HIGH" )
12 L = or( J, K )
13 M = lift( NOT L )
14 N = krao( H, L )
15 O = krao( H, M )
16 P = sum( N )
17 Q = sum( O )
18 return( P, Q )
```

A.6 QUERY 14

```
1  select
2      100.00 * sum(case
3          when p_type like 'PROMO%'
4              then l_extendedprice * (1 - l_discount)
5          else 0
6      end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
7  from
8      lineitem,
9      part
10 where
11     l_partkey = p_partkey
12     and l_shipdate >= date ':1'
13     and l_shipdate < date ':1' + interval '1' month;
```

```
1  A = filter( lineitem.shipdate >= "1995-09-01" AND lineitem.shipdate < "1995-10-01")
2  B = lift( lineitem.extendedprice * (1 - lineitem.discount) )
3  C = hadamard( A, B )
```

```
4  D = filter( match( part.type , "PROMO.*" ) )
5  E = dot( D, lineitem.partkey )
6  F = hadamard( C, E )
7  G = sum( F )
8  H = sum( C )
9  I = lift( 100.00 * G / H )
10 return ( I )
```