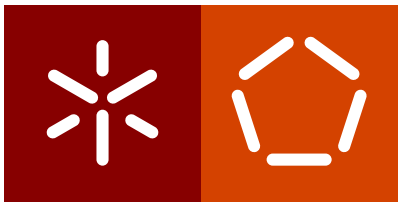
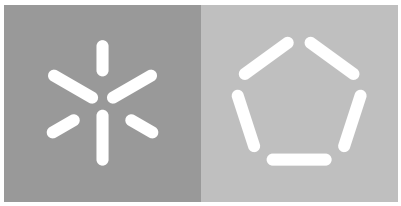


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Rodrigues

Adapting HEP-Frame for a Streaming Approach to a Linear Algebra OLAP Engine

December 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel Rodrigues

Adapting HEP-Frame for a Streaming Approach to a Linear Algebra OLAP Engine

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Alberto José Proença
André Pereira

December 2019

ACKNOWLEDGEMENTS

I am deeply grateful to my research supervisors, Professor Alberto Proença and Professor André Pereira, for their valuable suggestions, patient guidance and the opportunities they provided.

I also want to thank my family for all the support and dedication. Especially my parents, who always strived to give me the best.

ABSTRACT

Analysing and storing large amounts of information is a crucial component of corporate business. Aiming at the strong interest of these organisations for more intuitive and high-performance solutions, multidimensional data analysis techniques emerge.

Among the different techniques of multidimensional analysis stands out the *Online Analytical Processing (OLAP)*, due to the indexed and compact data organisation that allows the development of more efficient solutions. Recently, a new OLAP engine based on *Typed Linear Algebra (TLA)* has been developed; this new proposal replace *Relational Algebra (RA)* with TLA, to create more efficient and robust queries. The TLA approach already produced interesting results that outperformed MySQL and PostgreSQL databases; however, it did not yet manage to overcome the performance of columnar-oriented database systems, such as MonetDB.

The main goal of this dissertation is to improve the performance of *Linear Algebra (LA)* queries in a multicore environment, taking advantage of the efficiency of LA operations and their inherent parallelism. In this sense, two new stream versions were developed and evaluated, based respectively on the *Data Stream Processing (DSP)* approach and the *Highly Efficient Pipeline Framework (HEP-Frame)*. Additionally, through performance analysis, the significant limitations were identified and removed from both the new versions and the previously stream version.

To evaluate and compare the performance of the developed versions, were performed different analyses on the benchmark *Transaction Processing Performance Council Benchmark H (TPC-H)*. These analyses compared run times, miss caches, memory utilisation, and computational performance on different machines; enabling the identification of advantages and limitations of the approaches studied. Finally, the results obtained were compared with the MonetDB database.

RESUMO

A Análise e armazenamento de grandes quantidades de informações é um componente crucial das corporações empresariais. Visando o forte interesse destas organizações por soluções intuitivas e de alto desempenho, surgem as técnicas de análise multidimensional de dados. Dentre estas técnicas, destaca-se o OLAP, devido à sua organização de dados indexada e compacta que proporciona o desenvolvimento de soluções mais eficientes. Recentemente, foi desenvolvido um novo motor de base de dados OLAP baseado em álgebra linear tipada; que substitui a convencional álgebra relacional por álgebra linear a fim de criar queries mais eficientes e robustas. Essa abordagem já produziu resultados notáveis que superaram as bases de dados MySQL e PostgreSQL. No entanto, ainda não superar o desempenho de sistemas de bases que seguem uma orientação colunar, como a base de dados MonetDB.

O principal objectivo desta dissertação é melhorar o desempenho das query de álgebra linear em um ambiente multicore, aproveitando a eficiência das operações de álgebra linear e seu paralelismo inerente. Nesse sentido, duas novas versões stream foram desenvolvidas e avaliadas, baseadas respectivamente na abordagem DSP e no HEP-Frame. Além disso, através de testes de desempenho, foram identificadas e removidas as principais limitações destas novas versões e da versão stream anteriormente desenvolvida.

Para avaliar e comparar o desempenho das versões desenvolvidas, foram realizadas diferentes análises no benchmark TPC-H. Essas análises compararam tempos de execução, caches misses, utilização de memória e desempenho computacional em diferentes máquinas; permitir a identificação de vantagens e limitações das abordagens estudadas. Finalmente, os resultados obtidos foram comparados com a base de dados MonetDB.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

AVX Advanced Vector Extensions.

C

CSC Compressed Sparse Column.

D

DP Data Processing.

DS Data Setup.

DSP Data Stream Processing.

DW Data Warehouse.

H

HEP-FRAME Highly Efficient Pipeline Framework.

I

ILP Instruction Level Parallelism.

L

LA Linear Algebra.

LAQ Linear Algebra Query Language.

M

MC Multiple-Consumers.

MP Multiple-Producers.

MPMC Multiple-Producers-Multiple-Consumers.

MPSC Multiple-Producers-Single-Consumer.

N

NUMA Non-uniform memory access.

O

OLAP Online Analytical Processing.

P

PAPI Performance API.

PU Processing Unit.

R

RA Relational Algebra.

S

SIMD Single instruction multiple data.

SPMC Single-Producer-Multiple-Consumers.

SPSC Single-Producer-Single-Consumer.

SQL Structured Query Language.

SSE Streaming SIMD Extensions.

T

TLA Typed Linear Algebra.

TLP Thread Level Parallelism.

TPC Transaction Processing Performance Council.

TPC-H Transaction Processing Performance Council Benchmark H.

1

INTRODUCTION

1.1 CONTEXT

With the growing amount of data produced worldwide, the need to store, organise, and analyse this information has become indispensable for any business. Many companies opted for multidimensional analysis techniques to maximise the value of the stored data by extracting as much useful information as possible. Among the various options for multidimensional analysis techniques, the OLAP technology stands out, providing a simple and efficient way of analysing raw data (IBM [2011]).

The OLAP technology is based on a multidimensional view of data, providing a more efficient and robust way of organising information, thereby enabling the development of more efficient queries. These queries are optimised to operate on large amounts of data, organised into multiple relations, often involving sophisticated data selection and manipulation operations. This technology is commonly associated with *Data Warehouse (DW)* database systems that are designed to support nonvolatile data storage and analysis. As expected from these systems, despite their dimensions and complexity, they value response time, which is one of the main focuses of OLAP technology.

Aiming at the strong interest of organisations for this type of databases, and the advantages in performance and usability of this technology, the TLA (Macedo and Oliveira [2010]) proposal comes up. This proposal highlights the advantages of combining the two technologies, the computational efficiency present in LA operations and the potential present in the OLAP approach, to optimise the performance of query execution.

The TLA approach is based on the formalisation of analytical querying through LA operations with a strongly typed notation and a representation of queries in Diagrams (Typed Diagrams); promoting a new way of approaching the conception of aggregate functions, through the combination of LA operations. It has been proven by Macedo and Oliveira [2010], Macedo and Oliveira [2015], and Macedo and Oliveira [2017] that data aggregation can be expressed with this approach, and therefore, it is conceivable to develop an OLAP database with these concepts.

The development of an OLAP database engine based on LA by Afonso [2018] proposes a new approach to RA on which OLAP approaches are based, replacing RA with LA; made possible by the concepts derived from the TLA approach and its data aggregation formulation. This approach has proven competitive compared to PostgreSQL (PostgreSQL Dev. Group [2019]) and MySQL (Oracle Corporation [2019]) databases based on traditional RA, outperforming the performance of these databases. However, it has not yet outperformed databases that follow a columnar approach, such as MonetDB (Idreos et al. [2012]). Thus, it is intended to optimise the current kernel of LA operations in order to approximate the performance outcomes of the MonetDB database. To achieve this goal, it is necessary to explore in greater detail the parallelism of LA operations, as well as the efficient distribution of tasks derived from them across the computational resources.Taking into account these requirements and characteristics of LA queries, it is essential to use a suitable framework, namely HEP-Frame.

The HEP-Frame (Pereira et al. [2016] and Pereira et al. [2015]) was developed to improve the execution throughput of pipelined data stream applications through the ordering of the operations performed on the data. Data stream pipeline applications consist of data processing tasks that have inter-dependencies, forming one or multiple graphs similar to the DSP architecture. The diagrams derived from the LA queries fall into place as they can be considered as a pipeline of LA operations. Thus enabling the integration of this queries with the HEP-Frame framework.

1.2 CHALLENGES AND GOALS

The goal of this dissertation is to optimise the performance of the current LA queries, to compete with database systems that follows a columnar orientation, such as MonetDB (Idreos et al. [2012]).

To this end, new versions of the LA query were developed, including the adaptation of the HEP-Frame and the implementation of queries in this framework. Throughout the development of these versions, performance analyses were performed in order to identify the limitations presented by the developed solutions and proceed with their removal. Finally, the developed versions were compared with the MonetDB.

The following tasks were addressed to achieve these goals:

- Analysis of *Linear Algebra Query Language (LAQ)* engine and HEP-Frame;
- Implementation of a new LA stream approach;
- HEP-Frame adaptation to support the execution of LA queries;
- Analysis and removal of limitations of the developed solutions;

- Comparison of the solutions developed with other databases;

1.3 CONTRIBUTION

The contribution of this work is essentially aimed at the development of more efficient LA queries. In which were optimised the various parts that form these queries, namely the memory management, its libraries and some changes in the definition of specific methods/functions present in this queries.

Additionally, two new stream approaches were developed and studied, including new points of view for future developments.

These contributions can be summarised as follows:

- Library optimisation of LA operations;
- Optimisation of query memory management;
- Development of two new LA stream approaches;

1.4 DISSERTATION OUTLINE

Chapter 2 summarises the state of the art about the LAQ engine, the HEP-Frame, the DSP approach, the target platforms of this work, and additionally, the libraries and tools used in the developed work. Chapter 3 describes the development made on the three stream approaches, OMP, DSP and HEP-Frame in conjunction with the applied optimisations. Chapter 4 presents the tests and analyses performed on the developed solutions, as well as their comparison with the MonetDB database. Finally, Chapter 5 presents the author's considerations about the work developed, as well as some relevant ideas to be explored in future works.

2

STATE OF THE ART

The basis for the development of this work is the OLAP engine based on the TLA approach, which was developed and tested by Afonso [2018]. With the results obtained by Afonso [2018] and Ribeiro et al. [2017], it has already been proved that this approach has the potential to surpass databases based on the RA approach, such as PostgreSQL (PostgreSQL Dev. Group [2019]) and MySQL (Oracle Corporation [2019]). However, there is still room for improvement, and it is in this sense that it is intended to optimise the system in order to compete directly with databases that follow a columnar approach such as the MonetDB database (Idreos et al. [2012]).

2.1 LINEAR ALGEBRA OLAP ENGINE

In order to understand the operations present in the LAQ engine, it is essential to understand how the data is represented and how the operations manipulate this data. Therefore, this section will present the processes that allow the generation of queries based on the TLA approach, including the different optimisations applied to LA operations and the bases that support these optimisations.

2.1.1 *Data encode*

The LAQ engine data model is primarily based on Kimball's methods (Connolly and Begg [2014] p. 1261). In this approach, the attributes of a table are divided into two categories, dimensions and measures; dimensions correspond to the attributes that have a specific range of values, for example, dates or names. Measures correspond to attributes that can be counted, for example, the number of orders or product prices.

As a practical example, table 1 shows the attributes *Status*, *Quantity*, and *ExtendedPrice*; where the first is a dimension, while the last two attributes are measures.

Lineitem				
#	OrderKey	Status	Quantity	ExtendedPrice
1	591	″O″	10	20
2	210	″O″	3	95
3	21	″F″	7	76
4	101	″O″	2	85

Table 1.: Relational representation of Lineitem table TPC-H

Order			
#	OrderKey	Priority	Date
1	101	″HIGH″	″1996-07-31″
2	210	″MEDIUM″	″1996-07-31″
3	591	″MEDIUM″	″1996-07-31″
4	21	″HIGH″	″1997-01-09″

Table 2.: Relational representation of Order table TPC-H

Representation of Dimensions and Measures

Since the TLA approach operates on matrices, it is natural that the data present in the database tables are represented in such structures. Therefore, each attribute of the tables is represented by a single matrix, where the dimensions are represented by Boolean matrices, while measures are represented by Decimal vectors. The differences in the type of representation between dimensions and measures are easily deducible, as the data type associated with each category is different; it requires an appropriate representation.

In the case of measures, the data represented is always in the same context so that the values can be directly associated with the corresponding tuple. That is, the data is represented in the cells of a row matrix, or vector, where the value of each tuple is represented in its column.

In the case of dimensions, the representation of the attribute value range requires the association between these and the tuples. For this, a Boolean matrix is used, where each row corresponds to a specific value from the value range of the respective attribute. While the columns correspond to the tuples present in the database, the association of the tuple (column) and the attribute value (row) is made by the values of the cells; where the positive value confirms the association, while the negative value negates the association.

For example, table 3 shows the attributes *Priority* and *Quantity* where the attribute values are represented in the rows of the matrix and the respective tuples are represented in the columns. It is important to note that there is only one match per column, as each tuple can only be associated with one value from the attribute's value range.

Date		Priority		Quantity
$\begin{matrix}\#\\ \text{"1996}-07-31\text{"}\\ \text{"1997}-01-09\text{"}\end{matrix}$	$\begin{matrix}1&2&3&4\\ \begin{bmatrix}1&1&1&0\\0&0&0&1\end{bmatrix}\end{matrix}$	$\begin{matrix}\#\\ \text{"}MEDIUM\text{"}\\ \text{"}HIGH\text{"}\end{matrix}$	$\begin{matrix}1&2&3&4\\ \begin{bmatrix}0&1&1&0\\1&0&0&0\end{bmatrix}\end{matrix}$	$\begin{matrix}1&2&3&4\\ \begin{bmatrix}10&3&7&2\end{bmatrix}\end{matrix}$

Table 3.: Representation of attributes in matrices

This type of representation originates from the projection functions, such as:

$$f : A \to B, \forall_{a \in A},\ \forall_{b \in B} \tag{1}$$

This function matches the tuple (columns B) to the attribute value (row where the bit corresponds to 1). Thus, if $[[f]]$ represents the matrix with A rows and B columns, it translates into the following formula:

$$f[a][b] = \begin{cases} 1, & \text{if } a = f(b) \\ 0, & \text{otherwise} \end{cases}, \forall_{a \in A},\ \forall_{b \in B} \tag{2}$$

This type of representation is presented by Macedo and Oliveira [2017] and Afonso et al. [2018] to introduce the representation of attributes in matrices and to demonstrate the potential of the TLA approach in the representation of these data and their relations.

It is from the matrix representation of the attributes and the conjugation of linear algebra operations that the LA queries are generated. Therefore, it is the basis for the development of the operations implemented in the LAQ engine.

Data consolidation

There are multiple ways to combine tables and attributes, such as aggregation, group-by, cross-tabulation and cube. These representations are supported by TLA approach, as demonstrated by Macedo and Oliveira [2010], Macedo and Oliveira [2015] and Macedo and Oliveira [2017].

To illustrate the formation of these data consolidations will be briefly presented the LA operations that correspond to the respective selection, equi-join, group-by, and Cartesian product operations; since these operations present well the concept of data manipulation in this approach.

The selection consists of determining the attributes to be consulted, so it is possible to identify the matrices that are used in the LA operations.

The join operation between two tables can be represented by a Boolean matrix, or projection function, that relates the keys between the two tables, as follows:

$$o\_orderkey^{o} \,.\, l\_orderkey \tag{3}$$

$$\begin{array}{c} \# \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 21 & 101 & 210 & 591 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ (o\_orderkey^{o}) \end{array} \cdot \begin{array}{c} \begin{array}{cccc} \# & 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \\ (l\_orderkey) \end{array} = \begin{array}{c} \begin{array}{cccc} \# & 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\ (o\_orderkey^{o}.l\_orderkey) \end{array} \begin{array}{c} \# \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

The matrix multiplication of the transpose matrix of the primary key (*Order* table) and the foreign key matrix (*Lineitem* table) produces a matrix that relates the tuples of both tables. Thus, the rows of the result matrix represent the *Order* tuples and the *Lineitem* tuples are represented by the columns.

Due to the Boolean representation of the associations, the multiplication of the matrices results in the value-by-value comparison between the matrix rows of $o\_OrderKey^{o}$ (representing the value of its primary key) and the column of matrix $l\_OrderKey$ (representing the respective foreign key value). In this way, the columns of the $o\_OrderKey$ matrix (or rows of the matrix $o\_OrderKey^{o}$) and the matching $l\_OrderKey$ columns confirm the relation between the two tuples.

On the other hand, the group-by operation is obtained from the composition of the chosen dimensions and the selected measure, as follows:

$$(o\_orderdate \bigtriangledown o\_orderpriority) \,.\, (o\_orderkey^{o} \,.\, l\_orderkey) \,.\, l\_extendedprice^{o} \tag{4}$$

$$\begin{array}{c} \begin{array}{cccc} \# & 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \\ (o\_orderdate) \end{array} \bigtriangledown \begin{array}{c} \begin{array}{cccc} \# & 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ (o\_orderpriority) \end{array} = \begin{array}{c} \begin{array}{cccc} \# & 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ (o\_orderdate \bigtriangledown o\_orderpriority) \end{array} \begin{array}{l} \text{"1996-07-31" \& "HIGH"} \\ \text{"1996-07-31" \& "MEDIUM"} \\ \text{"1997-09-01" \& "HIGH"} \\ \text{"1997-09-01" \& "MEDIUM"} \end{array}$$

$$
\underset{(o\_orderdate \,\triangledown\, o\_orderpriority)}{\begin{array}{c}\# \;\; 0 \;\; 1 \;\; 2 \;\; 3 \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}\end{array}} \cdot \underset{(o\_orderkey^{o} \,.\, l\_orderkey)}{\begin{array}{c}\# \;\; 0 \;\; 1 \;\; 2 \;\; 3 \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\end{array}} \cdot \underset{(l\_extendedprice^{o})}{\begin{bmatrix} 111.3 \\ 2019.7 \\ 1212.0 \\ 14.0 \end{bmatrix}} = \underset{(result\ of\ (4))}{\begin{bmatrix} 14.0 \\ 2131.0 \\ 1212.0 \\ 0.0 \end{bmatrix}}
$$

In this example is shown the combination of the operations corresponding to the Cartesian product, equi-join and group-by operations.

Firstly, the Khatri-Rao operation $(\triangledown)$ (corresponding to the Cartesian product) between *Date* and *Priority* dimensions is applied, producing a projection matrix that combines both dimensions into a single matrix, as explained in section 2.1.2. Secondly, there is the multiplication of the matrix resulting from the parity operation, Khatri-Rao, and the projection matrix resulting from the equi-join operation, previously calculated. Finally, the *Dot* product obtained with the $l\_extendedprice^{o}$ allows the aggregation of the values (corresponding to the group-by operation) according to the attribute values represented by the lines of the projection matrix previously calculated, as shown.

These concepts can be extended to more complex queries that meet the requirements of the OLAP standard. These operations are explained in more detail in Afonso et al. [2018],Macedo and Oliveira [2015] and Macedo and Oliveira [2017].

2.1.2 *Linear Algebra Operations*

For the execution of LA queries, the following operations are essential: Dot product, Hadamard-Schur product and Khatri-Rao product, together with the derived operations: Attribute filter, Fold and Lift operator. These operations are responsible for manipulating data within query execution, similar to the examples shown in the previous section. However, in their implementation, these operations have undergone significant changes respectively to their original formulas. Thus, in order to better understand the operations of LA queries, it is essential to analyse in detail how these operations were implemented in the LAQ engine.

Before explaining the algorithms of the LA operations implemented, will be presented the two main optimisations applied to the system, which focus respectively on the programming paradigm and the representation of sparse matrices used.

Pipelining of operation and Block Matrices

The programming paradigm used takes advantage of two crucial aspects present in the LA queries.

- The first aspect is the possibility of decomposing the queries into base operations, enabling the formulation of execution pipelines based on the different operations that constitute the original formula (query).
- The second aspect concerns the possibility of breaking down the calculation of intermediate matrices by dividing the matrices into multiple submatrices (or *blocks*) and applying the LA operation on them.

Since both the division of the matrix into *blocks* and pipelining of operations are supported by the commutative and associative properties of these operations (with some exceptions). This allows the independent processing of different *blocks* of data simultaneously, provided that the limitations presented by LA operations are respected.

These assumptions lead to the definition of two types of dependencies between operations as not all operations allow division into *blocks*; which is discussed, in greater detail, in section 2.1.3.

It is important to note that *block* division in this particular case only applies to the dimension of the matrix representing the tuples of interest. That is, the data for a given tuple is never divided; if the tuples are represented in the matrix columns, the division of the matrix is made by columns, if the tuples are represented in the lines the division is made by lines. Therefore, each *block* contains the information of a specific number of tuples in a given matrix.

Briefly, the programming paradigm used is based on the LA operations pipeline, along with problem division into minor problems, similar to the division and conquer paradigm, with some constraints.

Sparse Matrices

The second optimisation concerns the representation of the matrices used in the queries since most of these are sparse, their representation in their entirety is inefficient, both in the amount of memory used and in the performance of the operations applied on them. For this, in the work of Afonso [2018], the *Compressed Sparse Column (CSC)* compression format was selected given that this format presents the best characteristics in terms of memory usage and asymptotic complexity.

As the name suggests, the data organisation is done column by column, and its access also follows this orientation. Therefore, each cell of the matrix is decomposed into the three parameters that define it: the value of the cell, the row and the column. For each

matrix, the respective cell values are stored in three distinct arrays where two of them save, respectively, the value of the cell and its row, for all non-null values. While the third array retains the column pointers, that relates the column index to its row and value in the respective matrices. Thus, the column array always has the size of the number of elements in the *block* plus one, so it stores all indexes and the final position (which corresponds to the size of the arrays).

As shown in the following example, the matrix representation of the *Shipdate* attribute in the CSC format:

Shipdate ("Sparse Matrix")		**CSC format**	
	# 1 2 3 4	*values*	$\begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix}$
$"1996-07-31"$	$\begin{bmatrix}1 & 1 & 1 & 0\end{bmatrix}$	*row_indices*	$\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$
$"1997-01-09"$	$\begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}$	*column_pointers*	$\begin{bmatrix}0 & 1 & 2 & 3 & 4\end{bmatrix}$

This format significantly reduces the space required to maintain and operate the matrices, while also providing a significant reduction in the amount of data manipulated. Both *block* division and CSC representation have been successfully implemented and tested by Afonso [2018].

Optimised data blocks

Thanks to the advantages of CSC compression and the characteristics of the matrices used in the representation of attributes and other intermediate representations, it was possible to develop eight structures that take advantage of these specific cases, as shown in the table 4.

In short, depending on the type of data to be represented, it is possible to discard the redundant components of the matrix elements. For example, in the representation of vectors, the representation of the lines is not required as these have only one line. In the representation of Boolean vectors or matrices, the representation of the values is not required since their representation does not add any information that is not verified with other fields.

Finally, in cases where there is one and only one element per column (dimensions and measures), the representation of the columns is redundant.

As a result of combining both optimisations, especially the CSC representation, LA operations have undergone drastic changes in the algorithms that implement them. The following analysis will present in more detail the changes that have been applied to LA operations to incorporate the optimisations presented in the previous section. For each operation will be presented the associated matrix operation, the operation that supports the

Block Type	Application example	Values	Rows	Columns
Decimal Vector	Store measures	✓	–	–
Bitmap	Store dimensions	–	✓	–
Filtered BitVector	Result of filter operation	–	–	✓
Decimal Map	Query with a GROUP BY clauses before aggregation, and after doing the Khatri–Rao operation	✓	✓	–
Filtered Decimal Vector	Query with WHERE clauses before aggregation, 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 aggregation	✓	✓	✓

Table 4.: LA CSC *Block* types (from Afonso [2018])

division into *blocks* and finally the implementation that aggregates the division paradigm and the CSC format.

Dot Product

In the LA approach, the *Dot* product corresponds to the well-known matrix product, following the same principles and requirements. The multiplication of two matrices A and B is denoted by $A.B$ or $dot(A, B)$ in LAQ. As shown in the following algebraic equation:

$$C_{ij} = \sum_{n=0}^{p-1} A_{ik} \ * \ B_{kj}, \ A \in M_{m*p}, B \in M_{n*p} \tag{5}$$

Due to the division into *blocks*, this multiplication is translated in the following equation:

$$A.B = \Big[A.B_1 | A.B_2\Big] = \Big[C_1 | C_2\Big] = C, \ B = [B_1 | B_2] \tag{6}$$

Note that, to calculate C, it is necessary to access the elements of B_1 and to access the whole matrix A. It is from this kind of restrictions that strong data dependencies are originated, as is presented in more detail in section 2.1.3. It is also important to regard that the number of *blocks* extends to the number of tuples in the matrix; thus, matrix B can be divided as many times as needed. As long as the number of tuples is respected, and the data is correctly aggregated at the end of the operation or query.

The *Dot* operation translates into the CSC format in an algorithm that computes the values of C through the values of A and B, discarding all null elements of A and B, indirectly, due to the CSC representation. The result of multiplying the element of B by the element of A is stored in C.

```
dot (A, B, C):
   nnz = 0 // Number of elements in C
   // 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 and save row
            C.values[nnz] = A.values[Apos] * B.values[Bpos]
            C.rows[nnz] = A.rows[Apos]
            // Increment the number of elements in C
            nnz += 1
   C.nnz = nnz
   C.nRows = A.nRows * B.nRows
   C.nCols = A.nCols
```

Listing 2.1: Dot pseudo-code (from Afonso [2018])

Since, by definition, there is at most one element per column, there is no need to traverse all rows of matrix A nor the rows of matrix B; it is only necessary to iterate through the columns of matrix B and calculate the product of the elements of matrix B with the corresponding element of matrix A. Lastly, if the elements of the matrices are booleans or bits (BitMap), there is no need to multiply the two elements, the validation of the elements is sufficient.

Khatri-Rao Product

The *Khatri-Rao* product corresponds to the line-by-line multiplication of a matrix A across matrix B, represented by $A \triangledown B$ or by $krao(A, B)$ in LAQ. The output from $A \triangledown B$ shows all row-by-row combinations of matrix A with matrix B, so it is possible to combine different dimensions in the same matrix; forming new matrices that indicate for each entry,

which of the new combinations of attributes is satisfied. In this case, the number of lines corresponds to all possible combinations of matrices A and B.

The following equation corresponds to the calculus of the cell value of row i and column j, where $A \in M_{m*p}$ and $B \in M_{n*p}$.

$$C_{ij} = A_{iy} \ * \ B_{kj}, \ y = i/m, \ k = i\%m \tag{7}$$

In this equation, y is the result of the integer division of row i by the number of rows in matrix A and k is the remainder of that same division.

For example:

$$A \triangledown B = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \triangledown \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} a*1 & b*2 \\ a*3 & b*4 \\ a*5 & b*6 \\ c*1 & d*1 \\ c*3 & d*4 \\ c*5 & d*6 \\ e*1 & f*2 \\ e*3 & f*4 \\ e*5 & f*6 \end{bmatrix} \tag{8}$$

With the division into *blocks*, this operation presents the following equation:

$$C \ = \ A \triangledown B = \Big[A1|A2\Big] \triangledown \Big[B1|B2\Big] = \Big[A1 \triangledown B1|A2 \triangledown B2\Big] \tag{9}$$

In this way, it is possible to calculate C_1 and C_2 entirely and independently. For the calculation of one of the *blocks* of C, it is only necessary to access the data of the respective *blocks* of A and B, without the need to load the entire matrix. In contrast to the definition of strong dependencies, in this case, these dependencies are classified as weak, which do not need to wait the full processing of other operations.

Due to the compression of the matrices in CSC format and due to the characteristics of the various data types that are processed by this operation, it is possible to optimise each case in a specific way, substantially improving the efficiency of the algorithms. These optimisations are similar to the optimisations presented in the *Dot* operation and are related to the data structures shown in figure 4. The following pseudo-code represents the generic algorithm of the *Khatri-Rao* operation.

```
1 krao (A, B, C):
2     nnz = 0 // Number of elements in C
3     // Pick the matrix with fewer elements
```

```
if A.nnz < B.nnz
   // Iterate the column pointer array until the las element (<A.nnz)
   for i = 0 to A.nCols AND A.cols[i] < A.nnz:
      // Store the current number of elements
      C.cols[i] = nnz
      // Chech if an element is present in both matrices
      if A.cols[i+1] > A.cols[i] AND B.cols[i+1] > B.cols[i]:
         // Get the current indexes of the values and rows arrays
         Apos = A.cols[i]
         Bpos = B.cols[i]
         // Merge the values and rows
         C.values[nnz] = A.values[Apos] * B.values[Bpos]
         C.rows[nnz] = (A.rows[Apos] * B.nRows) + B.rows[Bpos]
         // Increment the number of elements in C
         nnz += 1
else:
   // Same algorithm, just swap the predicates of the if condition
C.nnz = nnz
C.nRows = A.nRows * B.nRows
C.nCols = A.nCols
```

Listing 2.2: Khatri-Rao pseudo-code (from Afonso [2018])

Hadamard-Schur Product

This operation is similar to the *Khatri-Rao* operation except that it is applied to vectors. The following equation shows the calculation of the elements resulting from the operation:

$$C_i = A_i * B_i \tag{10}$$

Both the equation corresponding to *block* division and the implementation are similar to the *Khatri-Rao* operation, always considering that they are applied to vectors.

Attribute Filter

As its name implies, this operation is used to filter out tuples that do not satisfy a particular predicate. This operation is applied to both dimensions and measures, showing only differences in the type of data that are compared.

Measure filtering consists of applying the Boolean function to the elements of the value field, indicating those that pass the test.

In the case of dimensions, the data filtered by the operation are the elements of the label corresponding to its dimension with the filtered data. The filtering algorithm is similar to

the case of the measures above, except for the change of element type (from Decimal to Literal).

In both cases, what is obtained from the attribute filter operation is a Boolean vector that indicates which tuples have verified the condition and which not. In order to obtain the values corresponding to the respective tuples, it is necessary to combine the result obtained with the tuple data using a dot operation for dimensions or the *Hadamard-Schur* operation for measures.

The following equation calculates the vector elements resulting from the application of operation *AttributeFilter*.

$$C_i = f(A_i) \tag{11}$$

Similar to the *Khatri-Rao* operation, this operation allows the independent calculation of matrix *blocks*, as shown in the following equation:

$$C = F(A) = [F(A_1)|F(A_2)],\ A = [A_1|A_2] \tag{12}$$

The algorithm of the operation, taking into account data format changes is as follows:

```
filter (A, C):
   nnz = 0 // Number of elements in C
   // Iterate the elements of A
   for i = 0 to A.nnz:
      // Store the current number of elements
      C.cols[i] = nnz
      // Checks whether the element satisfies the condition
      if F(A.values[i]):
         // Increment the number of elements in C
         nnz += 1
   C.cols[i] = i
   C.nnz = nnz
```

Listing 2.3: Attribute Filter pseudo-code

Lift

The $Lift$ operation is used to perform mathematical operations, element by element, on a vector (normally derived from measures). It is even possible to aggregate multiple vectors into the expression in order to obtain the desired results.

The following equation calculates the elements of the result vector C:

$$C_i = f(A_i, B_i, ...) \tag{13}$$

With the *block* division of matrices, the calculation of matrix C can be represented by the following equation:

$$C = F(A, B, ...) = [F(A_1, B_1, ...)|F(A_2, B_2, ...)] \tag{14}$$

Finally, the algorithm that translates the calculation of elements of vector C, taking into account the CSC format, is as follows:

```
lift (A[], C):
   // Iterate all tuples of element A[o]
   for i = o to A[o].nnz:
      // Iterate all elements of A
      for j = o to A.size()
         // Store all values of index i
         v[j] = A[j].values[i]
      // Stores the value of the function F applied to the tuples i
      C.values[i] = F(v)
   C.nnz = A[o].nnz
```

Listing 2.4: Lift pseudo-code

Fold

The *Fold* operation serves to aggregate the data, usually results of the *Lift* or simple measures, allowing the calculation of the sum, average, major or minor of its values. This operation is often the last operation to be applied, as it is commonly the query result.

As an example of a *Fold* operation function, consider the summation of the elements of A:

$$fold(A) = \sum_{i=0}^{n} Ai \tag{15}$$

The equation that represents the *block* division of this operation is divided into two phases. First, the function is applied to each *block* of the matrix; then the same function is

applied again to the data obtained from each *block*. The result of the last operation is the final result, as shown in the following equation:

$$fold(A) = \sum_{k=0}^{K}(\sum_{i=0}^{N/K} Ak_i) \tag{16}$$

In the equation the value N is the total number of elements of A, K is the total number of *blocks* and Ak_i represents the i element of *block* k of matrix A. For the remaining calculations it is only necessary to replace the function of summation with another function, the principles applied are the same.

```
fold (A, C):
   C = 0
   // Iterate all tuples of element A
   for i = 0 to A.nnz:
      // Adds the value of tuple i
      C += A.values[i]
```

Listing 2.5: Fold pseudo-code

2.1.3 *Streaming approach*

Block division not only dramatically decreases the amount of memory required in query execution, but also provides for parallel calculation of multiple *blocks* simultaneously. Based on these properties, a first query LA stream approach was developed by Afonso [2018].

In this streaming approach, query calculation is performed through a pipeline of LA operations, taking advantage of the commutative and associative properties of linear algebra together with *block* division. Thus it is possible to define a pipeline of LA operations that process multiple *blocks* of data in parallel, as shown in figure 1. Hence the adoption of the term streaming, since data is processed as it is produced, element by element, forming a data stream that extends through the pipeline.

However, the division of matrices into *blocks* and the parallel and chain execution of LA operations implies some requirements, and dependencies, that must be met. One such requirement is the identification of *blocks* processed by the pipeline, which in this case is indirect since related *blocks* are processed separately from the others. Moreover, in the constitution of the query pipeline and its execution, it is necessary to satisfy the different data dependencies between operations.

These dependencies are classified into two categories: strong dependencies and weak dependencies; indicating what kind of influence these dependencies have on the chain of operations.

Strong Dependencies

Strong dependencies, as already mentioned in 2.1.2, require the complete *block* of data flow until the dependency is satisfied. There are two operations that can enforce this kind of dependencies, the *Dot* operation and the *Fold* operation.

In the case of the *Dot* operation, the matrix A must be fully loaded before starting any mathematical operation on the *blocks* of matrix B; as shown in formula 6.

While in the case of *Fold* operation, there is a different type of strong dependency, the operation itself does not require strong dependency; however, this imposes a strong dependency on operations that depend on it. Since the result of *Fold* operation is unique and indivisible, it can only be used when all instances of the respective *Fold* operation are completed.

Weak Dependencies

Weak dependencies identify dependencies between pipeline operations that operate on data *blocks*. Preventing only the advancement of the data flow dependent on the *blocks* in question, such as $Lift$, *AttributeFilter*, *Khatri-Rao* and *Hadamard-Schur* Product.

The implementation of this system is based on the shared memory approach and makes use of the OpenMP model where each pipeline delimited by strong dependencies constitutes a potential parallel cycle. As shown in figure 1, each blue box represents a loop, and the blue arrow represents its strong dependency.

For ease of identification, this version will be called the OMP version throughout the work.

2.1.4 *Query Example*

In order to consolidate these concepts, a practical example will be analysed, in which these processes are applied. The most appropriate query for an in-depth analysis of the operation of this approach is the query 6 from the TPC-H benchmark. Since it presents a greater variety of operations compared to other queries implemented in the LAQ engine.

Query 6 has the following structure in *Structured Query Language (SQL)*.

```
select
   sum(l_extendedprice * l_discount) as revenue
from
   lineitem_1
where
   l_shipdate >= '1994-01-01'
```

```
    and l_shipdate < '1994-01-01' + interval '1' year
    and l_discount between 0.06 - 0.01 and 0.06 + 0.01
    and l_quantity < 24;
```

Listing 2.6: TPC-H query 6 - SQL code

The code in SQL is converted to the LAQ language as shown below.

```
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)
```

Listing 2.7: TPC-H query 6 - LAQ code

This LAQ code translates to the pipeline schema of figure 1, where is shown the data flow that occurs between operations. By observing this structure can be noted that each box represents an independent operation and that it can be executed in parallel with the others. Nevertheless, there is an order that has to be preserved in data processing.

This approach allows for better performance and better use of memory compared to previous versions since it is not necessary to load the entire matrices into main memory, which may even be impossible for significantly large databases.

2.1.5 *Considerations to highlight*

The current version takes advantage of the parallelism present in each loop, taking advantage of a high degree of parallelism, which brings significantly good results compared to the MySQL and PostgreSQL databases. However, this approach has the disadvantage of imposing the sequential execution of these pipelines, imposed by strong dependencies; when it is possible to advance the execution of other operations that are not directly affected by this dependency, especially load operations. Therefore, exploring a new way of performing operations can bring significant performance gains.

2.2 HEP-Frame

HEP-Frame is a framework designed to optimise the execution of pipeline data stream applications. These applications consist of a set of inter-dependent tasks that coerces the

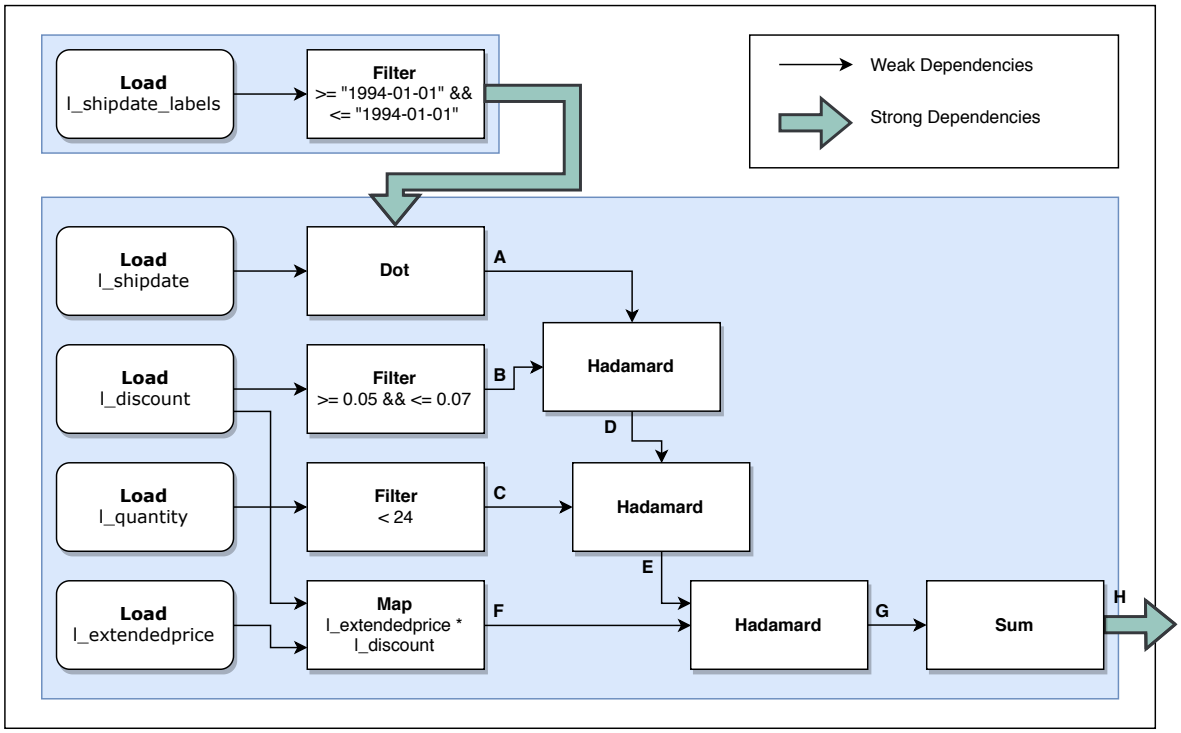


Figure 1.: TPC-H Query 6 - LAQ representation (from Afonso [2018])

order in which these are executed, forming a processing pipeline. This pipeline is responsible for processing multiple independent datasets to obtain the desired results. It takes advantage of both the parallelism present in the data and the parallelism present between pipeline tasks; through parallel execution of multiple pipeline instances, processing multiple datasets simultaneously, and parallel execution of multiple independent tasks on the same pipeline instance. Since datasets are independent, and so are the instances of the pipeline, this problem falls into the category of embarrassingly parallel problems.

However, being in this category does not guarantee that the direct exploitation of their parallelism is the most efficient solution for these applications. Once, there are other factors to take into account, such as the possible limitations of its characteristics, namely computer-bound, memory-bound or even I/O-bound.

The application execution process typically involves loading datasets of varying size and processing them simultaneously, by the pipeline tasks to obtain the desired results. The load and preprocessing process include loading data into datasets and allocating them to appropriate data structures. Once the data is allocated, it is ready to be processed. Each dataset is then processed by the proposition pipeline, where each proposition consists of a task that will operate on the dataset and a filtering criterion that determines whether or not the dataset should proceed in the pipeline flow. Data filtering prevents the processing of irrelevant, or even unwanted, data in the computation of the final result. Thus, these

propositions and their dependencies compose the pipeline of propositions, as shown in figure 2.

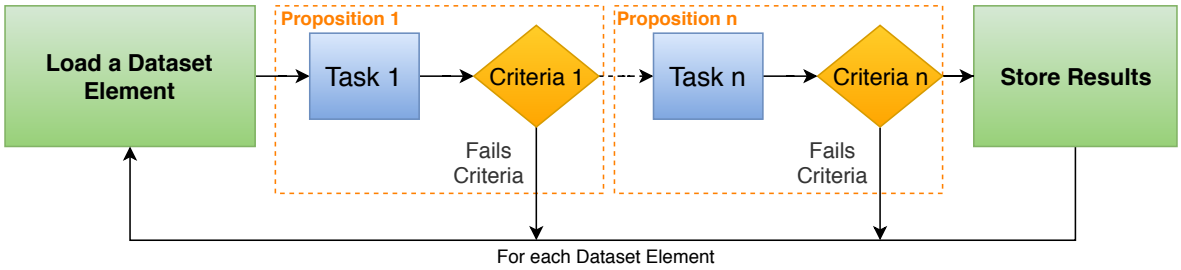


Figure 2.: Structure of a typical pipelined (from Pereira et al. [2016])

At run time, the framework manages the resources allocated to the *Data Setup (DS)* and *Data Processing (DP)* tasks favouring the computationally heaviest. At the same time, the task sequence is reordered to find the most efficient solution, where tasks that discard more data and/or are computationally lighter are executed first; while tasks that discard less data and/or are computationally more demanding are performed at the end of the pipeline. This framework has already been tested and verified and has shown impressive results in both computer-bound and memory-bound applications (Pereira et al. [2016]).

In short, the most important features of this framework for the present work are:

- Distribution of work by available resources;
- Discard of data that is not relevant;
- Dynamic reordering of the pipeline;

2.2.1 *Filtering*

Filtering consists of validating the data obtained in the execution of a task, if the result is invalid the data is discarded from the system otherwise the data is saved, and the data continues to be processed by the following tasks. This mechanism is not imposed by the framework, since not all analysis contemplates verification phases.

However, the addition of this mechanism should always be considered, since its addition contributes significantly to the reduction of the execution time.

2.2.2 *Reordering Pipeline*

The dependencies between the tasks impose an order that is not changed by the framework; however, it is possible to change the order between tasks that are not directly dependent. The purpose of changing the order of the tasks is to find a solution that maximises resource utilisation, for this are considered two characteristics of each task to perform, its computational weight and the number of events that discards. If a task is significantly heavy compared to the others, it is advantageous to process it after processing the lighter tasks, since the data filtering can discard the dataset before reaching this task, to save time and resources. If a task discards many datasets, it is advantageous to place the task in the first levels of the pipeline, maximising the useful work of each task.

Pipeline reordering takes place at runtime dynamically, making the framework extremely flexible and portable as it adapts to the system. Additionally, it is possible to save the pipeline configuration at the end of the execution for reuse in the future. Thus in the next executions, the application starts with an optimal pipeline order.

2.2.3 *Task Scheduler*

The scheduling of tasks is one of the most important features of this framework, allowing the assignment of machine resources to the tasks that benefit most. Task scheduling involves the distribution of resources by running tasks; the goal is to minimise the execution time by balancing the load tasks with the computational tasks since load tasks can take longer to load data than tasks can process them leading to starvation. The initial configuration of the Scheduler allocates more resources to the load processes than to the computational processes; then at runtime, the Scheduler tries to find the balance between the two processing types.

2.2.4 *Considerations to highlight*

The features previously mentioned make this framework ideal for the execution of LA queries, allowing the exploration of the parallelism present in the queries and the balance between load and process operations. Since the queries fit into the class of pipeline data stream applications that HEP-Frame supports; as queries can be decomposed into tasks and the pipelines can be formed with their dependencies.

For more details on how the framework works, it is advisable to read the following papers (Pereira et al. [2015] and Pereira et al. [2016]).

2.3 DATA STREAM PROCESSING

The DSP approach (Zhang et al. [2017]) is very popular in systems that demand a rapid response. With the growth of this type of systems have appeared several proposals with different architectures, such as Apache Storm (Apache Group [2019b]) and Flink (Apache Group [2019a]). These approaches usually have a processing pipeline with message passing and/or parallel data on-demand, both of which occur in the LAQ engine. As the name implies, these approaches are based on stream processing which is described in more detail in the following section.

2.3.1 *Stream approaches*

In data processing, there are two fundamental approaches that have profound implications on system architecture and performance, streaming and batch processing.

Batch processing consists of processing multiple data elements at a time, which is done after a specific period of time or when a given condition is satisfied.

Stream processing consists of processing each data element as soon as it is received, and the time between receiving the element and its processing is expected to be minimal. The main advantage of stream processing is its responsiveness, as this paradigm adjusts to a fluid processing of the elements, inducing a faster response by the system. However, in order for the data to be processed continuously, constant monitoring of the received data is necessary, which introduces an overhead in the control of the fluidity of the system.

Given that, the stream approach has the potential to increase the throughput, and thus increase system performance, despite the potential overhead of data control.

2.3.2 *Architecture*

Pipeline processing with message passing can be abstracted to a graph architecture, where nodes in the graph represent either data source operators or data processing operators (tasks), and the edges represent the data flow between operators. As shown in figure 3.

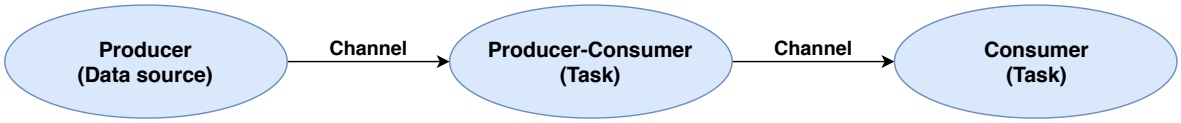


Figure 3.: Example of a DSP architecture

In this topology, there are two fundamental operators, the data sources and the processors. The data sources generate data (for example, loading data from disk) which will be consumed by the tasks.

On the other hand, from the perspective of message-passing process, other differences between operations can be identified, subdividing these into two subsets, the set of Producers and the set of Consumers. Producers are responsible for providing the data to one or multiple Consumers, while Consumers undertake to perform specific operations on the data received. With this definition, it is easy to conclude that data sources are the only elements that can not be Consumers. However, the intermediate pipeline operations also produce data derived from the data sources.

The message passing between a Producer and a Consumer is made through one or several communication channels, with the possibility of several Producers feeding the same channel (*Multiple-Producers (MP)*), and at the same time, several Consumers reading data from that channel (*Multiple-Consumers (MC)*). With these combinations, there are four types of relation between Producers and Consumers, which are *Single-Producer-Single-Consumer (SPSC)*, *Multiple-Producers-Single-Consumer (MPSC)*, *Single-Producer-Multiple-Consumers (SPMC)* and *Multiple-Producers-Multiple-Consumers (MPMC)*, as shown in figure 4.

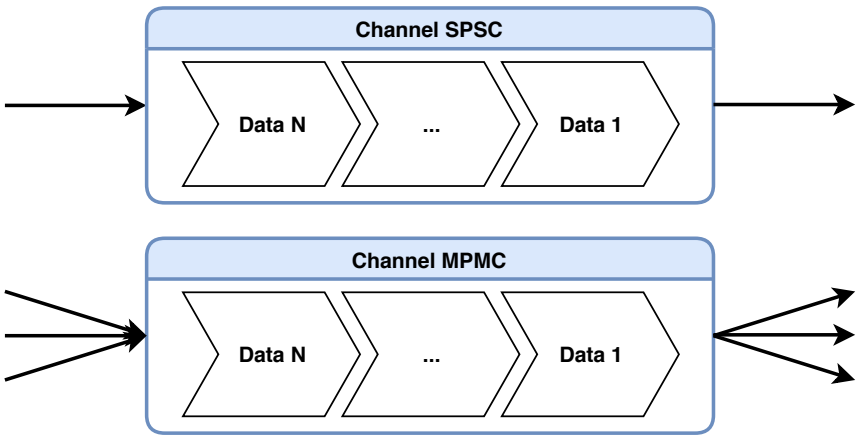


Figure 4.: Types of Communication Channels

Since, as data flow control increases, there is an increase in design complexity, SPSC communication channels tend to perform better than the rest, whereas MPMC communication channels tend to perform poorly.

2.3.3 *Considerations to highlight*

The DSP approach was used for the development of an intermediate version, before proceeding with the development of queries in the HEP-Frame. The focus of this approach

is to explore the parallelism between multiple operations, enabling a more flexible and dynamic query execution.

2.4 TARGET PLATFORMS

Computer systems, over the years, have dramatically increased their computing power, storage capacity, energy efficiency and consequently their complexity. These changes are driven by the growing demand for higher capacity systems to execute heavier applications, decrease response time, and increase throughput. These developments include improvements and specifications in the design and architecture of these systems and their processing units, memory hierarchies, and components that enable its communication; from massive supercomputers, which have already reached hundreds of thousands of TFlops/s, to small devices, which despite their size and power limitations deliver impressive performance results.

However, nowadays, the hardware optimisations do not always guarantee performance improvements, to get the most out of these optimisations, they must be explicitly exploited. In particular, the exploitation of multiple processing units through parallel programming models, which since 2005, due to the inability to increase clock frequencies caused by thermal dissipation, has become the most efficient method to increase processing capacity in these systems. For this, it is crucial to understand the mechanisms behind these improvements and how to leverage them to develop more efficient solutions.

2.4.1 *Homogeneous systems*

The target platform of this work is the homogeneous systems. These systems consist of one or multiple CPU chips in conjunction with their memory banks. When there are multiple CPU chips in the same socket, they are interconnected by their point-to-point interconnect interface allowing data to be shared. Thus each CPU can access the memory bank of other chips; however, access time will be limited by the CPU communication interface.

These systems have *Non-uniform memory access (NUMA)* pattern, which is another important aspect to keep in mind when developing applications for these systems. This aspect is crucial and tricky to evaluate without the necessary knowledge, as the allocation and access to data are not explicit; which can lead to possible performance losses that may go unnoticed if no proper assessment is made.

Despite the growing appeal of heterogeneous computing systems and their performance results, they will not be addressed in this work. However, the possibility of its investigation in future works is not discarded.

CPU Chip

Microprocessors have undergone multiple changes in their capabilities and architecture over the years. Starting with single-core microprocessor models with the ability to execute thousands of instructions per second to today's models with dozens of cores and capable of executing billions of instructions per second on each core. Relying on the multiple optimisation mechanisms that enabled this growth.

This innovation process consisted of exploring the limitations of each microprocessor generation and the characteristics of the applications executed by them, developing new methods to fill these gaps. Coming to the design of today's microarchitectures, which have assimilated into a single chip multiple core processors, currently two levels of private cache for each core and a third level of shared cache; not to mention the various mechanisms that aim to maximise the performance of instructions executed in each core. Among these mechanisms will be briefly discussed the memory hierarchy (cache), prefetch, *Instruction Level Parallelism (ILP)*, vectorisation and *Thread Level Parallelism (TLP)*.

Memory Hierarchy

In order to hide the high latency access to global memory, memory hierarchy levels take advantage of the principles of temporal and spatial locality of data. The memory hierarchy consists of multiple layers of memory where higher levels, closer to the core, have smaller sizes and lower access latencies. While the levels furthest from the core have higher storage capacities and in turn, higher latencies.

Of this hierarchy stands out the cache memories, which are located between the cores and the main memory. Their design seeks to optimise data access times by taking advantage of consecutive data access and data reuse, placing data at higher levels in the hierarchy.

However, this technology alone cannot take full advantage of these features; in most cases, it is necessary to design the data structures and operations that compute on this data.

2.4.2 *Prefetch*

The purpose of the prefetch is to predict what data/instructions the program will access and load the data/instruction to higher memory levels in order to feed the processor faster.

Prefetching instructions is usually easy to predict, not requiring complex speculation methods since the execution of the instructions is mostly sequential; thus, most of the times it is only necessary to load the adjacent instructions in advance.

In the case of data, the prediction of the access pattern is more complex. In this sense, multiple forecasting and data loading mechanisms have been developed, among which

stream prefetching and spatial prefetching are present in the *Processing Unit (PU)* used in this work.

The stream prefetching takes care of loading streams of data that are repeated throughout the execution based on the pattern of accesses previously verified, making it possible to load sparse data. While spatial prefetching focuses on consecutive data loading. Data prefetching can also be done at the software level through specific functions or by the compiler.

These mechanisms allow the exploration of memory hierarchy beyond the spatial and temporal location of the data, even allowing the reduction of cache misses due to cold cache, that would be impossible otherwise. However, some restrictions must be respected, as these mechanisms require available memory space and bandwidth.

2.4.3 *Instruction Level Parallelism*

ILP, as its name suggests, consists of overlapping instructions when they can be executed in parallel. There are multiple techniques that exploit instruction-level parallelism, both hardware-level (dynamically) and software-level (statically). Among these methods stand out pipeline, multiple-issue and speculative execution.

Pipeline

The pipeline is a method that consists in the division and execution of instructions in multiple stages, taking advantage of the parallelism present between the different steps of the executing instructions. Instruction stages include, or are part of, the instruction fetch, instruction decoding, execution, memory access, and write-back operations that are part of the instruction. Enabling to execute up to five instructions simultaneously, assuming that the previous operations constitute the pipeline stages, each at a different stage of the pipeline. Note that the depth of the pipeline is determined by the processor architecture.

Multiple issue (Superscalar)

Multiple issue is the replication of internal core components allowing the launch of multiple operations at each stage of the pipeline. These approaches increase peak instruction throughput and exploit parallelism at the level of the instructions. However, control and data dependencies greatly limit performance and gains, as the core has to wait for dependencies to be resolved before continuing with the other dependent statements.

Speculative execution

To circumvent the control dependencies and to fill in the gaps present in stages and units that have no instructions to perform, the PU is allowed to speculate on the results of the branches. Through prediction mechanisms, the result of the branches is stipulated, and the instructions of this path are executed. Subsequently, if the stipulated solution is found to be incorrect, the instructions executed in that context are discarded, and the correct path is taken. Otherwise, the instructions are confirmed, and the execution continues.

2.4.4 *Vectorisation*

Another form of parallelism is the single instruction multiple data stream operated on data vectors. This mechanism consists of applying a single vector instruction to a set of elements exploring data-level parallelism. These mechanisms are usually applied in cycles where algebraic operations are repeated over a contiguous data structure. However, there are dependencies that limit or prevent the application of these methods on the code in question. Some of these dependencies are presented below.

Loop dependencies

Cycle instances should be as independent as possible; The best case is when the cycles do not have any dependency between them, allowing unrestricted parallel processing. Loop dependencies occur whenever there are dependencies between loops, where the result of one iteration influences the result of the next iteration.

Indirect memory access

Access to data from a data structure, such as an array or matrix, must be direct. That is, the element to be accessed must be determined directly by a counter or variable without the involvement of complex computations. The indirect memory access occurs when data access is determined by a function or other method that does not resort to direct assignment of index values.

Branches

The presence of conditions or jumps implies the consideration of different paths or set of operations to be performed, which prevents the direct application of vector operations. To solve this problem, jumps and conditions should be avoided; if the conditions cannot be avoided, it is possible to replace them with masked assignments.

Function calls

There should be no functions in the loop unless these functions are inlined or if they are vector functions. The way functions are declared inline has no implications on the application of the vectorisation, the declaration can be done automatically, by the compiler, or manually. Functions that have a vectorial version in their libraries are also allowed.

Pointer aliasing

Due to the flexibility of pointers in C/C++, it is possible that the memory regions identified by the different pointers may overlap. The compiler may not be able to determine if the applied operations are safe. In these cases, it is possible to indicate to the compiler that the pointers are not overlapping by adding special keywords.

Unknown loop count

It is essential for the compiler to identify the total number of loop iterations to form their vector instructions, as instructions may attempt to alter unwanted elements. For this, the number of iterations must be invariant, and there can be no breaks in the loop.

Other loops

Vectorisation of multiple nested loops is possible; however, the target loop of vectorisation is always the innermost.

2.4.5 *Thread level parallelism*

Thread level parallelism is the parallelism between multiple threads, excerpts from the main program that have their instructions and data, allowing them to be executed in parallel. This kind of parallelism, compared to the previous ones, is considered of high degree because it is the division into multiple tasks of the work involved in a program.

There are many programs that present this type of parallelism intuitively, as embarrassingly parallel cases, in which the work is easily divided into multiple tasks presented few dependencies. However, it is not always easy or intuitive to explore these concepts, let alone analyse their behaviour in a system; this is why it is essential to understand how these mechanisms work.

Multithreading technology allows the sharing of core resources across different threads, contributing to a more efficient use of them. By overlapping thread execution, it is possible to occupy functional units that ILP cannot fill, such as dependencies derived from I/O operations that block the execution sequence.

This technology features some variants that more or less aggressively exploit the processing unit resource, where the more aggressive variant, simultaneous multithreading, ensures simultaneous execution of multiple threads on a single core. While other approaches consist of sharing resources over time, assigning resources to a single thread. In the case of the simultaneous multithreading variant, it is necessary to replicate specific resources and provide the processing unit with the ability to launch multi-threaded instructions in a single clock cycle, such as Intel Hyperthreading.

2.5 LIBRARIES AND PROFILING TOOLS

Throughout this work, different software tools were used for the development and evaluation of the different components. These tools include development tools: OpenMP (Board [2015]), MPI (Edgar Gabriel et al. [2004]), HEP-Frame by Pereira et al. [2016] (already presented 2.2); as well as the performance analysis and profiling tools: PAPI (S. Browne et al. [1999]), Perf (Weaver [2013]), VTune (Reinders [2017]), Dstat (Wieers).

OpenMP

The OpenMP library provides an API that makes it possible to exploit thread-level parallelism in a shared memory environment. This directive extends the C, C ++, and Fortran programming languages with a series of mechanisms that allow thread execution and control, providing the necessary mechanisms to manipulate their behaviour.

Message Passing Interface

MPI is an interface library used for problems that exploit parallelism in distributed memory environments. It supports the C, C ++ and Fortran programming languages and provides the mechanisms that allow the creation, manipulation and communication between multiple processes. This library can be used in conjunction with the OMP library to explore both shared and distributed memory environment.

Intel VTune

VTune is a profiling and tracing tool focused on performance analysis of sequential and parallel programs. In the multiple analysis profiles, multiple performance analysis metrics are combined to expose the application behaviour in the computer system. In this work stands out the use the following profiles: Hotspots, Memory Consumption, and Threading.

The Hotspots profile analyses in detail the path of function calls, allowing the identification of functions that have the most significant impact on application execution. These

analyses are of great importance in identifying the sections of code that are worth considering in advance, as these are where the most considerable limitations lie.

Additionally, the Memory consumption profile, which is more focused on memory allocation and release operations, allows the analysis of memory management throughout the execution.

Finally, the Threading Profile focuses on thread performance analysis, monitoring the consumption of computational resources, lock and wait functions (lock mechanisms) and the execution time of each thread.

PAPI

The *Performance API (PAPI)* project specifies an API that allows access to hardware performance counters. These counters are associated with specific events that occurred during program execution, such as cache-related events, number of instructions, number of cycles, among others.

Perf - Performance analysis tools for Linux

The Perf tool is a profiling and tracing tool capable of accessing hardware event counters allowing the counting of executed instructions, cycles, misses, and others. It has the advantage of not requesting changes to the code of programs to be exploited.

Dstat

Dstat is a system resource monitoring tool that allows the recording of the use and behaviour of computational resources, such as the amount of memory used, I/O operations, network traffic, and more.

3

DEVELOPMENT AND OPTIMISATION OF STREAM APPROACHES

As already mentioned in the state of the art, the LAQ engine already has a parallel version that presents good performance results. However, there is room for improvement; the approach used was developed with OpenMP and consists of parallelisation of the entire pipeline. This version considers each pipeline bounded by strong dependencies as a possible parallelable loop. In other words, it is taking advantage of the parallelism of the data since different datasets are independent of each other, as are the calculations applied to them. However, it is not taking advantage of the parallelism between the operations present in the pipeline. From another point of view, the parallelism explored is at the pipeline level and not at the operations level.

Thus, moving from the pipeline level to the operations level (decreasing the granularity of parallelism) can bring significant performance gains. The decrease in granularity cannot be excessive, as this extreme can lead to loss of performance, so betting on parallelisation at the operations level is a promising solution; making it possible to perform operations within the pipeline in parallel. However, for the correct execution of these operations, it is expected that the control mechanisms present a higher overhead since the shift in parallelism granularity brings new requirements and limitations.

Therefore, for this solution to be viable, it is necessary to make the most of the characteristics of the operations and the pipeline itself, taking into account the available computational resources by maintaining a minimum overhead of the control mechanisms. It is at this point that the advantages of the HEP-Frame structure are emphasised, considering that the execution of the LA query can be divided into tasks/operations forming a pipeline. By taking advantage of the mechanisms and structure from the HEP-Frame, such as pipeline reordering, distribution of resources and data filtering. As already presented in 2.2.

To this end, the HEP-Frame must be adapted to the execution of LA queries. Since the query structure already fits the framework requirements, but the HEP-Frame does not have the mechanisms that support the execution of certain operations, specifically load operations. Nevertheless, before proceeding with the integration, an intermediate solution will be implemented to assess the performance gains of the approach while at the same time, reassessing operations and query execution.

3.1 DSP APPROACH TO LA QUERIES

In order to develop LA queries in this approach, it is essential to consider some requirements, identified from the analysis of query 6, which can be generalised to the rest; these requirements are:

- Support for SPSC, SPMC, MPSC communication;
- Identification of stream elements;
- Memory management.

3.1.1 *Support for SPSC, SPMC, MPMC communication*

All types of communications are formed by one or multiple communication channels, as already mentioned in sectionf 2.3. That is, the SPMC and MPSC communication channels are formed from SPSC communication channels, as shown in figure 5, where SPMC channels are labelled with green arrows while the MPSC channels are identified by operations classified as Join. In order to facilitate both the implementation and the understanding of the system, it was defined that Consumers are the basis for the creation of dependencies (communication channels) between Producers and Consumers in the pipeline. As such, Consumers own the communication channels, while Producers only use the available communication channels to communicate with Consumers.

SPMC communications are formed by multiple single channels between a Producer and multiple Consumers, so the Producer is responsible for sending the data produced to their channels. There is no limit to the number of Consumers present on an SPMC communication channel, as long as the channels are made available to the Producer.

In the case of MPSC communication, certain restrictions arise in the handling of datasets received by the Consumer. Since datasets must be processed in pairs, as each pair is unique and identifiable, they must be joined before proceeding with their processing. The explanation of this structure is presented below.

Join

Operations classified as Join are those that receive multiple arguments as input, requiring at least two input channels (i.e. MPSC communication channels). As the datasets of these inputs must be correctly matched, and since the synchronisation and order of receipt of datasets are not guaranteed, it is necessary to control their reception. Therefore, data sets that do not have a pair are stored until their pair is received, the process only continues when the pair is complete. For this, a shared data structure was used to store the datasets

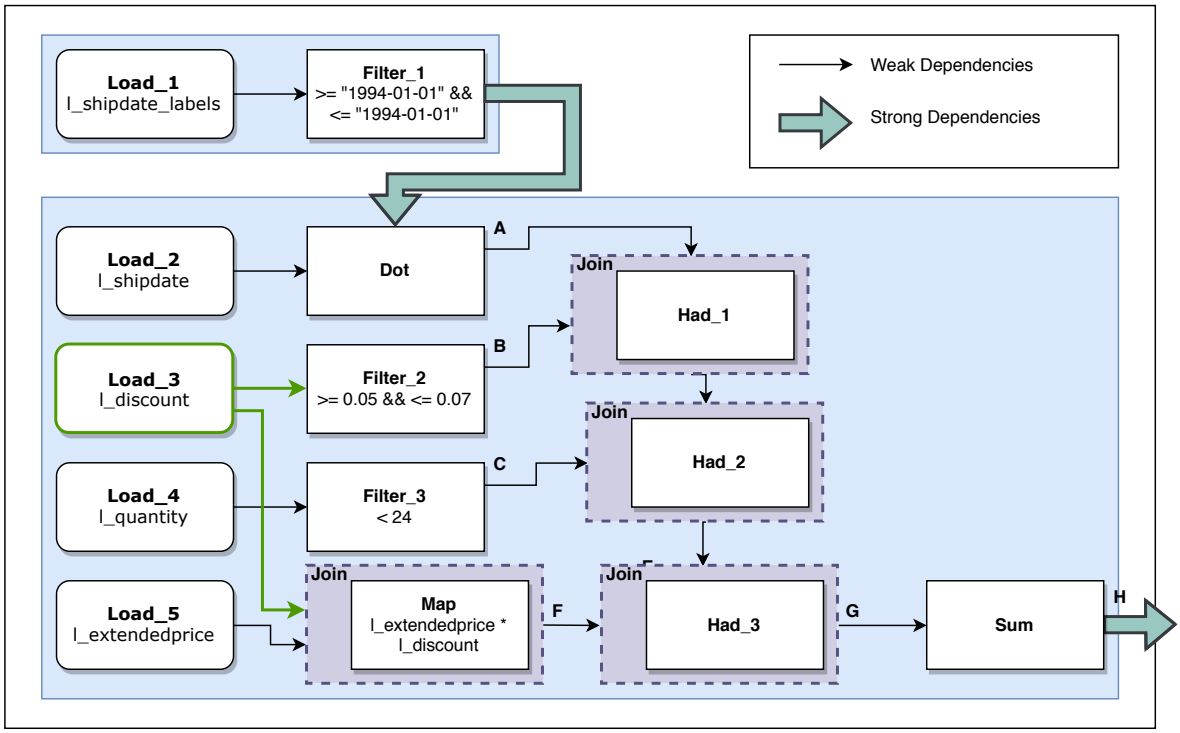


Figure 5.: TPC-H Query 6 - DSP design

with their identifier, controlling access to them by lock mechanisms (mutual exclusion). In the current state of the system, Join operations only support the reception of data from two Producers; however, it is possible to scale the current implementation to a more significant number of Producers.

It is important to note that the operation/task is incorporated into the Join structure, allowing direct access to the data.

In an initial implementation, it was considered a more generic solution that involved the use of three communication channels. Where the function of the Join operation was to aggregate the pairs of arguments, that are related, and send the new structure to the respective LA operation. However, VTune performance analysis showed that the solution brought excessive overhead. Therefore this solution has been replaced by the solution previously presented

Communication channels

Each communication channel is made up of at least two dataset queues; these queues are divided into two sets, the queues held by the Producers and the queues held by the Consumers. Each Producer has its queue of reusable datasets while Consumers, depending

on the number of inputs they receive as arguments, have one or multiple queues of datasets ready to be processed; as shown in figure 6.

Thus, whenever it is necessary to produce a new dataset, the Producer requests a new structure available in the reusable dataset queue or allocates a new structure. That is, if there are reusable data structures, the Producer gives preference to reuse them; however, in case there are no reusable structures available the Producer proceeds with the allocation of a new data structure.

Once the dataset is delivered to the Consumer queue, it forces one of the threads to wake up, the threads then retrieve the dataset from the queue and process it. When dataset processing finishes, the Consumer checks to see if it has already been consumed by all Consumers who share it, if this condition is met, it is reinitialised and queued for reusable dataset. Datasets are only considered "consumed" when all Consumers who share the dataset have finished processing it.

All queues are shared between multiple instances of their Producer and/or Consumer, and their access is restricted by mutual exclusion. For Producers, there is no restriction on the creation of new datasets, as mentioned above; thus, they are not blocked. However, Consumers are blocked from accessing their data as well as access to their Producer's reusable dataset queue.

As mentioned above, Consumers do not have active waits, as in this case, this solution performs poorly. Thus, passive waiting mechanisms have been developed; these mechanisms are modulated according to the number of elements present in the queue. Consequently forcing threads to sleep when a queue is empty, reducing resource usage while waiting for new datasets.

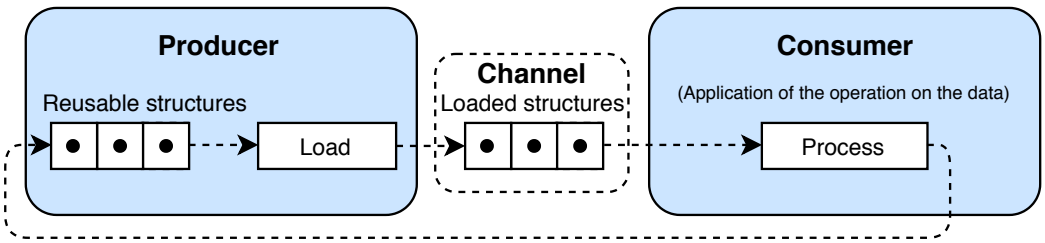


Figure 6.: Communication channel queues

It is important to note that the size of the dataset stream is undetermined, i.e., both Producer and Consumers are unaware of its size. Therefore, mechanisms are required to indicate if the channel's data stream is active or has already ended. For this purpose, non-blocking (or lockfree) variables were used to indicate the current state of the stream, being visible to all Consumers and Producers that share the channels.

3.1.2 *Identification of stream elements*

In addition to the representation of the communication channels, it is essential that the data in the stream is identifiable by the operations. Once this approach does not guarantee that the order of the datasets will be maintained through the pipeline and since this order is crucial in the execution of operations; an identifier is added to each dataset and transmitted along with the dataset throughout the pipeline.

3.1.3 *Memory management*

In addition to the reuse of the datasets explained above, there is another aspect to consider in memory management, namely the sharing of datasets that occurs in SPMC communications. At first glance it may seem that the reuse or remove of data from memory can be entirely performed by any Consumer; however, there are cases where elements are shared by several Consumers, and it is impossible to assign the task of reuse or remove the elements to a random Consumer, since data may still be required.

For this problem, two approaches were considered; the first was the duplication of the elements that are shared. In this way, the data is no longer shared, and each consumer is responsible for releasing their own. However, this approach was discarded because it implies a significant increase in memory consumption.

The other approach consists in the use of a counter, which starts with the number of Consumers that share the element. And then, every time a Consumer finishes processing the element, the counter is decremented; when the counter reaches zero, the dataset is added to the Producer queue for reuse or is removed.

Lockfree implementation

A lockfree solution has been tested; however, despite good performance results, queue sharing across multiple threads does not guarantee exclusive access to the elements. This fact leads to the degradation of the results obtained by the system. For lack of solutions to solve this problem, the use of lockfree data structures was discarded.

3.2 LA OPTIMISATIONS

After analysing the performance of LA query execution, it was observed that the creation of new *blocks* is responsible for a significant amount of execution time, so three optimisations were implemented to the OMP and DSP versions, and later extend to the HEP-Frame version:

- Removal of initialisation;
- Reuse of *blocks*;
- Avoid data structure copy operations;

3.2.1 *Removal of initialisation*

The first applied optimisation is the removal of the initialisation of the vectors used in the *block* structures since this is not required for the correct execution of the operations. The values of the vectors are always rewritten above the previous ones, which allows to further optimise the use and creation of *blocks*, as presented in the next point.

3.2.2 *Reuse of blocks*

The second optimisation is the reuse of *blocks* since their allocation and release has a significant weight at execution time. This optimisation is made possible by the systematic use of this data structure in LA operations and the ease with which these structures are prepared to be reused. Since its size is fixed throughout the query execution and as data is rewritten whenever an operation is performed.

In the OMP version, this optimisation can be carried out to the extreme since the number of instances of operations is constant throughout the execution, it is possible to define a constant number of *blocks* needed for the execution of the query. Each thread will allocate the set of *blocks* needed to execute its pipeline and will reuse these *blocks* for each iteration. Thus the amount of memory requested by the program remains constant throughout the execution, avoiding unnecessary memory allocations and releases.

In the stream approach, due to the flow of data between threads, it is not appropriate to define a constant number of *blocks* to perform the operations. Since by limiting the number of *blocks* that are used, the dynamic characteristics of the approach would be limited as well. That is, given that Producers and Consumers proceed independently, unnecessary waiting time may be required due to the lack of available *blocks*, which in some cases may even lead to deadlocks.Instead, *blocks* are created as needed, and all *blocks* that have already been consumed are immediately made available for reuse.

This process significantly decreases memory usage and runtime. However, it requires some overhead for maintaining the *blocks* and for communicating them between the Producer and Consumer entities. This is the same system presented in section 3.1, since the first implementation, like the first OMP version, proceeded with the allocation and release of the *blocks* whenever necessary.

3.2.3 *Avoid data structure copy operations*

In addition to these optimisations, some improvements were also made to LA operations.

Filter

In the filtering operations, it is verified that whenever the filtering function is called the vector, passed as an argument, is copied. This occurs because of the vector structure is passed directly as an argument, which, by definition of the C++ Standard Library, implies a copy of the vector. For further details, it is advisable to read the move semantics of C++ Standard Library (Josuttis [2012]).

This behaviour is not needed in this case and is even unwanted as it adds a substantial overhead for each filtering operation executed. Thus instead of passing vectors directly as an argument will be used its reference (or pointer) to avoid copy instructions.

Query Structure

In this case, there is a problem similar to the previous one; however, this problem occurs in the code that constitutes the LA queries. In *Filter* and *Lift* operations, the *blocks* are passed through vectors; which implies the copy of the *blocks* if they are not passed by reference. Similar to the previous problem, the references to the *blocks* were passed to the vector.

3.2.4 *Vectorisation*

Another evaluation performed in this work is the potential of LA operations for vectorisation. Since the LA approach is based on matrix and vector operations, its study is a crucial point for the development of more efficient operations and consequently, an improvement in query performance. For this study, the vectorisation reports generated by the compiler were used.

Dot

The *Dot* operation has a data dependency between cycles, being the variable *Nnz* responsible for this dependency. Since *Nnz* is updated in each cycle if a particular condition is confirmed, it implies a dependency between cycles that cannot be avoided. In addition, there are indirect and non-consecutive accesses in the remaining variables, as shown in the pseudo-code 2.1.

Khatri-Rao/Hadamard-Schur

Similar to the *Dot* operation, the data dependence of the variable Nnz is verified in both operations, making loop vectorisation impossible.

Filter

In this operation a data dependence between cycles is verified again, being the variable Nnz again responsible for this dependence. There is also a function call within the loop; however, in this case, it is not an obstacle to vectorisation as it is generally defined as inline. On the other hand, the inner cycle can be optimised through unrolling since it calls for a value assignment.

Lift

In the case of the $Lift$ operation, it is possible to apply the vectorisation, since there are no data dependencies between cycles, the inner cycle can be removed or unrolled, and the function F is defined as inline. However, the estimated gains are not very significant.

Fold

This operation already supports vectorisation and does not require any changes, since the operation consists only of the sum of the vector elements. In this case, the vectorisation report estimates a significant gain of $1,9$.

3.2.5 *Results of Optimisation*

From the results, it can be observed that both optimisations show a significant improvement in queue 6 performance. Especially in the reuse of *block*, both in the OMP and in the DSP version, as shown in figure 7.

These results highlight the impact that the allocation and release of objects have on the performance of this type of application. The gains are most evident in the DSP version, although the OMP version would be expected to take more advantage of *block* reuse due to its design. This difference is mainly due to the flexibility of the DSP approach, which, in addition to taking advantage of the *block* reuse, reducing execution time, also takes advantage of its dynamic characteristics; since *blocks* are always created as needed without additional wait times.

Only by comparing the execution times of the sequential tests is visible a significant difference in the performance of the various versions. In addition, it is also noted that both

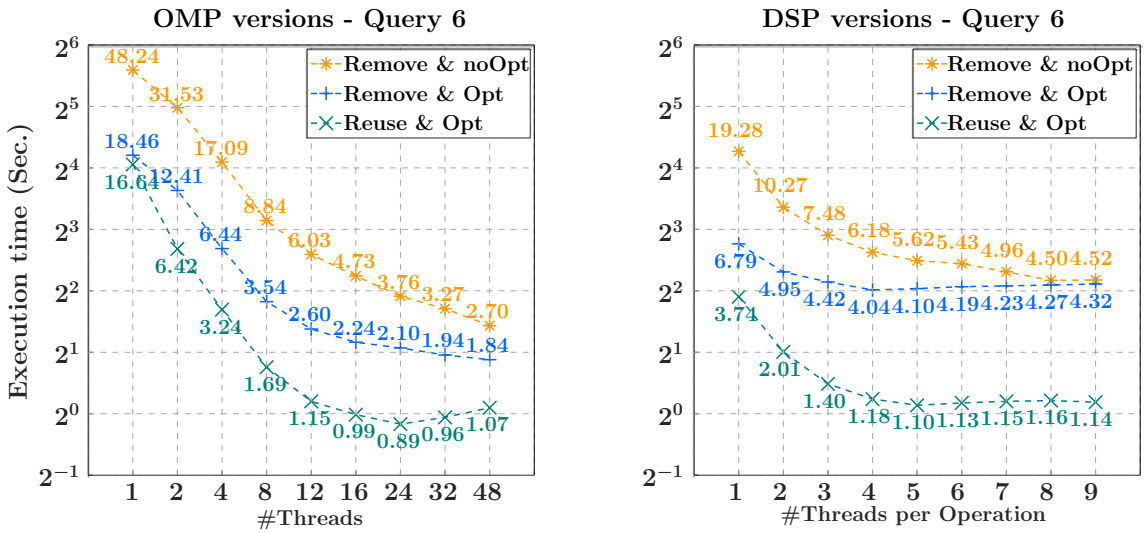


Figure 7.: Comparison between optimisations (scale $2^5 GB$) - Node 662

approaches, with optimisations, scale significantly better. Especially the OMP version that features a super-linear speedup up to 12 threads.

3.3 HEP-FRAME IMPLEMENTATION

At last, a prototype structure of execution was developed in the HEP-Frame. Since the queries structure falls into the HEP-Frame supported applications category; it was only necessary to adapt the current framework system to allow the execution of LA queries.

This adaptation includes changes to the dataset load and process since the structure relied solely on the ROOT library (Rademakers et al. [2012]) to load and create data structures for the datasets. Considering that the use of this library is specific and does not allow the load of data in the file format in which tables are handled in the LAQ engine, it was necessary to implement new methods that allow it. Thus, new load methods had been developed based on LA query load processes, keeping the same design as the mechanisms based on the ROOT library.

Additionally, memory management mechanisms such as memory reuse, developed in the DSP approach, and methods to control the amount of memory used by the program were added. The points at which changes take place can be organised as follows:

- Data reading;
- Data Processing;
- Memory management;

However, to accommodate the execution of LA queries, some features of HEP-Frame have been set aside, namely the Scheduler. Since thread management has undergone some changes, it has been opted to keep the number of threads constant throughout execution to facilitate the development of this new version.

3.3.1 *Data Reading Methods*

For the correct execution of load operations and to facilitate the implementation of the reuse of data structures, some changes have been made in the execution of read threads.

In general, the events are pre-distributed by read threads, and they are responsible for their load, reuse and/or removal. The events considered correspond to the *block* set that is loaded in a pipeline iteration, as in the OMP version, the same applies to the dataset. Both events and dataset are viewed as a single element to be processed by the pipeline.

The assignment of datasets is done in an interleaved manner to facilitate control of boundaries between the domain of the read threads and the process threads. Thus, the read threads proceed pairwise without the need for large control mechanisms. Event loading continues to be in parallel; however, Scheduler's thread management throughout the execution is not supported in this version.

The functions responsible for loading data are inserted into the framework interface just as they are used in OMP version queries. Moreover, the HEP-Frame complex management mechanisms remain hidden from the programmer, just as the methods that use the ROOT library.

3.3.2 *Data Processing Methods*

The most significant changes to the data processing methods is the limitation of the number of threads per event and the data access mechanisms. Preliminary tests show that running multiple threads on the same event impairs application performance. Since all prepositions have a low computational weight and due to the locks implicit in the assignment of propositions (access to the preposition table) the overhead associated with the competition between threads is too high. Thus, a maximum number of threads per event is pre-defined, forcing parallelism exploration between multiple events rather than parallelism between pipeline operations.

Finally, to limit data access for both read and process threads, a status indicator is used for each event. The value indicates whether the event is ready to be processed or if it has already been processed. In this way, the process threads know when they can start processing the event or if they should expect it to be fully loaded. On the other hand, load

threads also use the pointer to reuse or remove events that have already been processed. This requires that at the end of processing an event, the respective process thread changes the value of the indicator.

3.3.3 *Memory management*

Memory management includes limiting the amount of memory used by the application and the mechanisms available for data reuse, similar to the DSP approach.

Control over the amount of memory consists of restricting the number of datasets allocated by each thread throughout the execution. The maximum number of datasets is determined by the maximum amount of memory required for processing an event and the maximum amount of memory available to the application. After determining the number of datasets, each read thread is assigned with the number of datasets that it can keep in memory. As soon as the read threads are released, they are responsible for maintaining their datasets, both in the case of memory allocation and release, as in memory reuse variant.

For memory reuse, the principles used in the DSP approach were followed, with some ease and new limitations. Firstly, the data structure that allocates the datasets is divided by the read threads, where each thread is responsible for its dataset, and these are always accessible. That is, there are no complex communication methods used in the HEP-Frame approach to return the datasets already processed. Thus the read threads are responsible for all dataset maintenance, allocation, reuse, and memory release. Secondly, as the DSP approach, whenever a read thread needs a new data structure, it requests a structure already allocated (reusable), if no structure is available, it proceeds to allocate a new one. Unless the memory limit has already been reached, in this case, the thread is prevented from progressing, until it receives a reusable dataset, and the allocation option is blocked for the remaining requests for new datasets. Therefore, the thread can only rely on the reuse of datasets, and if they are not available, it is put to sleep for a predetermined period of time.

3.3.4 *Results*

The tests performed on this version are not extensive; only the execution times of the two memory usage variants previously presented were compared. Additionally, the amount of memory and the number of threads per event were limited to 2GiB and 2, respectively. It is also important to note that the data presented correspond to the execution of query 6 with *block* of 64Ki size since this option gives the best results.

As shown in figure 8, the removal and allocation of memory has significant overhead; reusing events is twice as fast as removing and allocating memory.

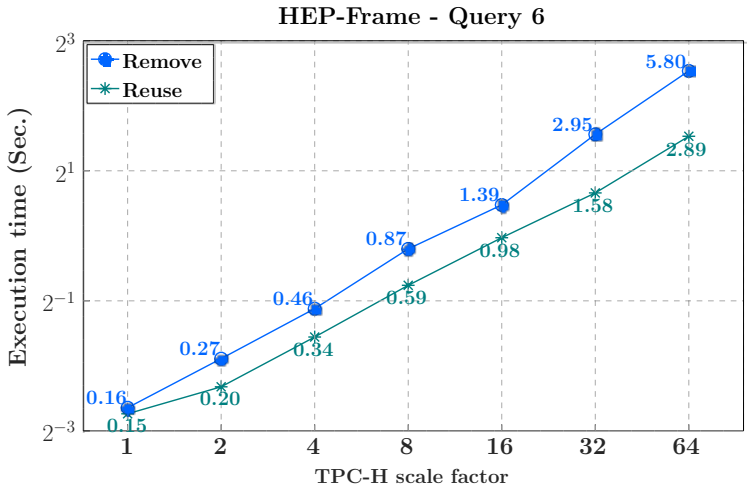


Figure 8.: Execution time HEP-Frame - Node 662

Nevertheless, the results obtained do not perform as well as the DSP and OMP versions; on the other hand, it has slightly better performance than the non-stream version of query 6. This release requires further study in order to find possible limitations.

4

VALIDATION AND PERFORMANCE RESULTS

This chapter will present the performed tests and analysis of the different versions developed to evaluate their performance. Additionally, the test environment, the benchmark used to compare the different approaches and the compilation options used are specified.

4.1 TEST ENVIRONMENT

For the evaluation and comparison of the results obtained, they must be reproducible under the same or similar conditions. This section covers the characteristics and conditions used to obtain the results presented throughout the paper.

TPC-H benchmark

The *Transaction Processing Performance Council (TPC)* benchmark aims to evaluate the performance of query execution on a database management system. There are multiple versions of the benchmark in order to evaluate different metrics and different database models. The version used for this work is TPC-H, being the version that best fits the OLAP model. Similar to previous work on the linear algebra database, this version of the benchmark was used.

The tool provides mechanisms to populate and test the database in addition to a standard query set. This queries differ in both the complexity of data manipulation and the amount of data in which they operate. All queries presented in this work are taken from the set of queries provided by the benchmark.

Heuristic

Among the different heuristics for the determination of results, the K best heuristic was chosen. Since the results obtained may vary significantly due to factors external to the application. The K best heuristic minimises these variations by determining a consistent sample of results; this choice varies both in the number of sample elements (K) and the maximum range of variation between the elements. That is, the heuristic determines the

best set of K elements within the stipulated range, taking into account a maximum number of tests. In this work, the heuristic is set to determine the best three results with a 5% range and with a total of 15 samples of a maximum of 25. If the results obtained after 15 tests do not yield at least three results within the 5% range, more tests are conducted, to achieve the set of 3 elements or reach the maximum number of tests.

Compile options

The compiler used was the Intel ICC (version 19.0.1) with the -O3 optimisation option, enabling all optimisations offered by the -O2 option in addition to more aggressive optimisation such as cycle fusion and collapse of IF statement. The application of vector operations (*Advanced Vector Extensions (AVX)*/*Streaming SIMD Extensions (SSE)* instruction set) is also included in the optimisations. The following libraries were also used: Boost (version 1.69.0), OpenMPI (version 1.8.2), OpenMP (version 4.5).

The following table shows the characteristics of the machines used in performance tests.

Table 5.: Testing platform: Computational nodes 662 and 781

Node	662	781
Model	Xeon E5-2695 v2	Xeon E5-2683 v4
Architecture	Ivy Bridge	Broadwell
#PU Chips	2	2
#Cores per PU	12 (24 Threads)	16 (32 Threads)
Base clock Freq.	2.4GHz	2.1GHz
L1 Cache	384KB (2 x 12 x 32 KB)	1MB 2x (16 x 32KB)
L2 Cache	4MB 2x (12 x 256KB)	8MB (2 x 16 x 256KB)
L3 Cache	2 x 20MB	2 x 40MB
Bandwidth (Máx.)	59.7GB/s	76.8GB/s
SIMD (instruction set)	AVX	AVX2
Network supported	Eth, Myrinet 10Gbps	Eth, Myrinet 10Gbps

Roofline Model

The roofline is a performance analysis model based on bounds and bottlenecks, allowing the evaluation of these limitations in a given system for different kernels. By considering the different hardware characteristics of the system, it is possible to identify in which way the kernels are limited, or whether the limiting factor is the hardware.

This association is presented in a graph, where the vertical axis represents the peak achievable performance in operations per second and the horizontal axis represents the operational intensity in bytes accessed per operation. Thus the association allows to relate the

characteristics of the hardware with the characteristics of the kernels. The two components that make up the roof are the peak performance of the processors and the peak performance of the memory used. These components represent the maximum peak performance that a kernel can achieve in the system. Thus, if the kernel is under one of these roofs, it is classified as computer bound, or memory bound, respectively. Furthermore, multiple ceilings can be added to the graph to highlight the different hardware optimisations and/or characteristics that must be exploited to achieve their performance limits. These ceilings serve as guides for evaluating kernel performance and getting a sense of the impact that kernels have or may have on kernel performance when exploited.

The construction of the roofline requires the calculation and selection of the most relevant system characteristics for the performance analysis of the kernels. Starting with the performance measure, the operational intensity, in this work, does not refer exclusively to the number of float point operations performed per byte loaded from memory. Since the amount of float point operations is relatively low in the studied kernels, as shown in section 4.2.1. Instead of considering float point operations only, it makes more sense to consider the other operations as well, excluding memory access operations. As for the ceilings, the following sequence was considered, ILP, TLP, *Single instruction multiple data (SIMD)*, and exploitation of the distributed memory system. The exploitation of the second processor has been placed at last because this aspect is not being explored properly in these versions.

For the node 662, the values for the ceilings were calculated considering the processor clock, the number of cores, the number of operations that are launched per clock (in single float point) and the number of processors used. As shown in the following formula 17.

$$PeakPerformance = \#PUs * \#CoresPerPU * ClockFreq * \#InstPerClock \tag{17}$$

$$PeakPerformance = 2 * 12 * 2.4E9 * 8 = 460.8GOps$$

For the number of operations launched per clock, the AVX instruction set is considered, which allows the execution of 8 single float operations per clock cycle. Although this value does not represent the total number of instructions performed per clock, since all arithmetic operations are considered, it is a reasonably close value to peak performance. As for the bandwidth, to obtain the maximum attainable performance values, the STREAM measuring tool was used.

4.2 COMPARISON BETWEEN DSP AND OMP VERSIONS

In search of answers for the difference in the performance of the DSP approach compared to the OMP version, multiple comparisons were made between the approaches. Among which the following aspects were analysed:

- Instruction Mix;
- Miss Rate;
- Execution time and Miss Rate per Operation;
- Roofline;

4.2.1 *Instruction Mix*

By analysing the instruction mix of the two approaches, it can be concluded that there are no significant differences between the approaches, as noted in 9. Both the relative and the absolute number of instructions.

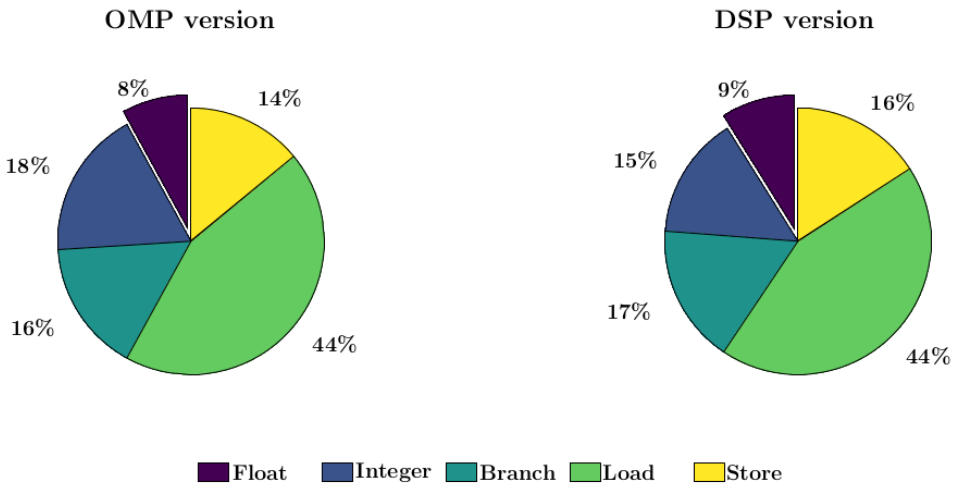


Figure 9.: Instruction Mix (Query6)

It can also be noticed that the amount of float point instructions is relatively low, as is the amount of instruction on integers. This already indicates a low computational intensity compared to the amount of memory access instructions. Especially load operations, which almost represent half of all instructions executed.

4.2.2 *Miss Rate*

When comparing the Miss Rate, there are no significant differences either, both approaches have the same behaviour, even with the *block* size variation.

The comparison of different *block* sizes is relevant because of their impact on query execution; as shown in figure 10, *block* size has a direct impact on both data read time and

processing. While this is not the only factor influencing runtime, it is a factor to be aware of.

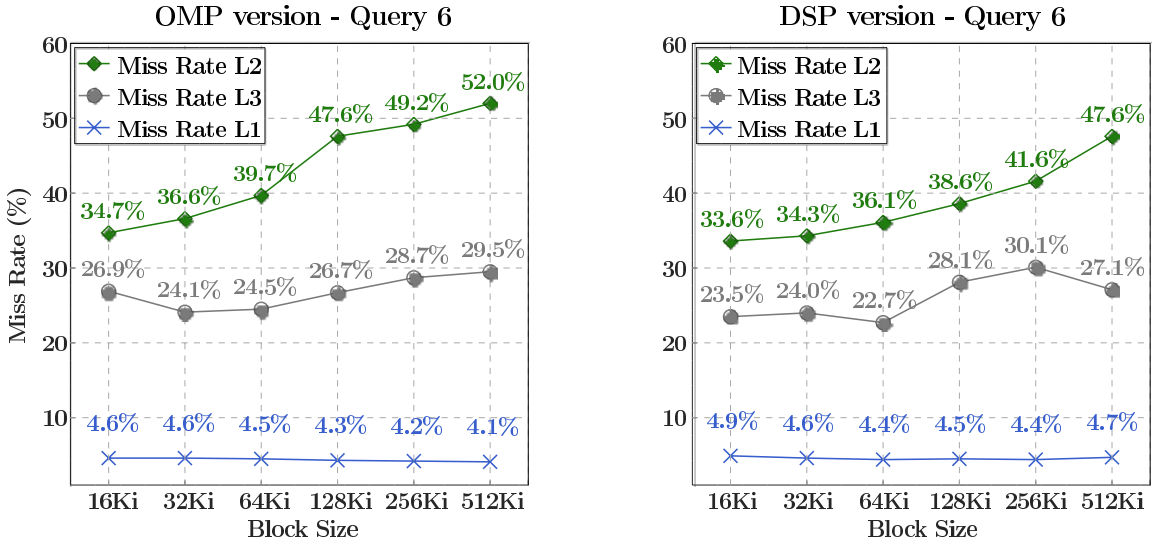


Figure 10.: Miss Rate - Node 662

The L1 cache misses shows a slight decrease in the OMP version as the number of elements in each *block* increases. For the DSP version, there is also a slight decrease to *blocks* of 256Ki elements. As for L2 and L3 caches, there is a noticeable increase, especially in the L2 cache. These results reflect the occupancy of the L2 cache, since, on average, each *block* of 32Ki elements occupies about 256KiBytes, which already exceeds the size of the L2 cache. This implies higher utilisation of the L3 cache, so there is an increase in L2 cache misses. The L1 miss cache is not affected due to data prefetching, since access to L1 cache is mostly sequential, as shown in the description of operations in section 2.1.2. The same does not apply to the L2 cache because of the relatively small and non-constant amount of access to it, which does not allow the application to prefetch data consistently. These factors favour, in this system, the *blocks* of size 64Ki, being this the size that presents the best results in terms of performance, in the OMP version, as in the DSP version.

It is important to note that both approaches make good use of the cache, with a maximum of 5% of misses in the L1 cache. Despite having relatively high miss rate values for L2 and L3 cache.

4.2.3 *Execution time and Miss Rate per Operation*

Regarding the analysis by operations, there are apparent differences between the approaches, as shown in figure 11. Firstly, in the DSP approach, there are noticeably better

load times, while the processing operations, without exception, have a significant loss of performance.

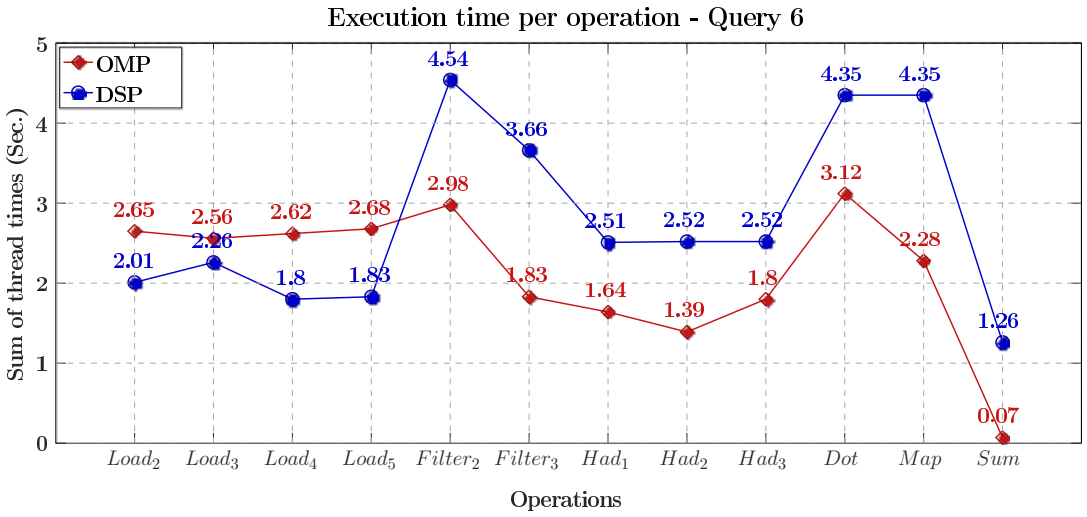


Figure 11.: Execution time per operation - Node 662

These results can be partially explained by the differences between cache miss operations, as shown in the graphs in figure 12. They show a slight decrease in the L1 cache miss rate when switching from the OMP approach to the DSP approach, which supports improved read times. Additionally, there is also a decrease in the L3 cache miss rate in these cases.

However, the differences mentioned earlier for load operations do not apply to the processing operations. Given that there are no significant or consistent differences with the performance loss observed in these operations.

4.2.4 *RoofLine*

For the analysis of the Roofline shown in figure 13, it can be concluded that both approaches are memory bound. Given that both versions are under the roof which delimits the peak performance of bandwidth.

Additionally, the performance shown on the dual-socket system has similar values to those shown on the same system with only one processor. This indicates the lack of memory balance between the two processors, especially in the load operations. There are other optimisations that are not yet explored, and that can bring significant performance gains, but not as impactful as exploiting the distributed memory environment.

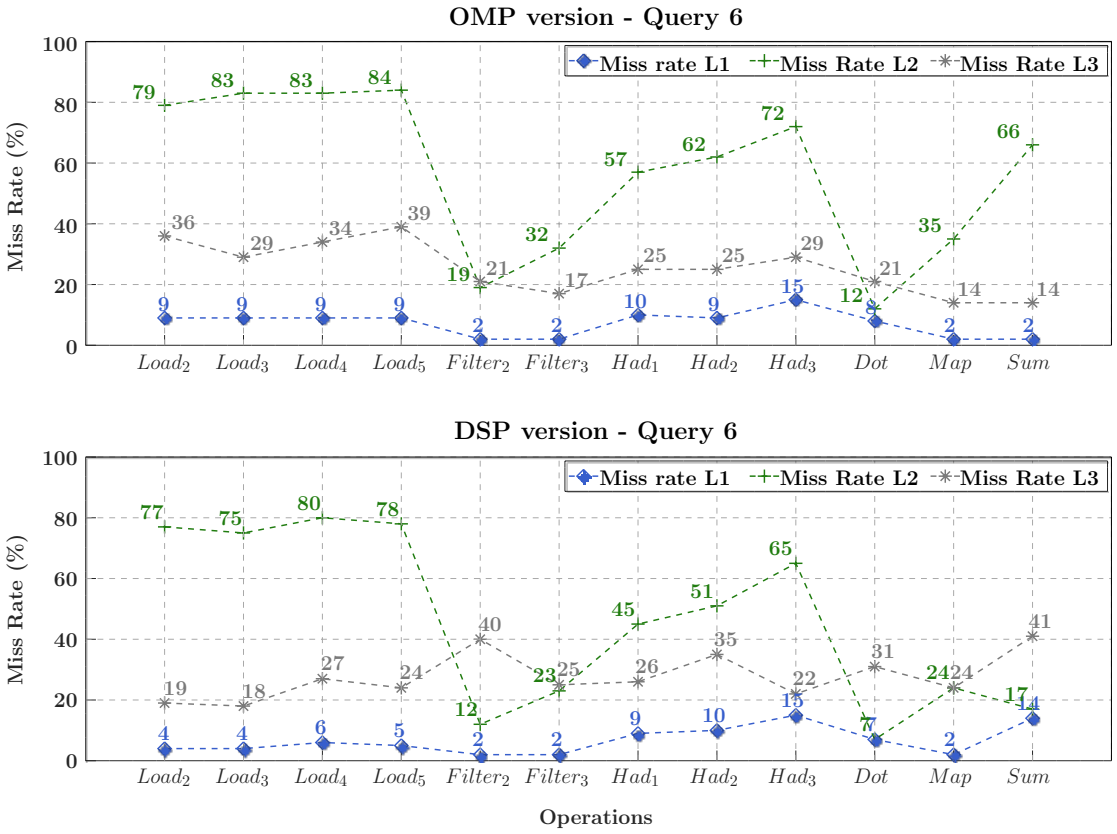


Figure 12.: Miss Rate per operation - Node 662

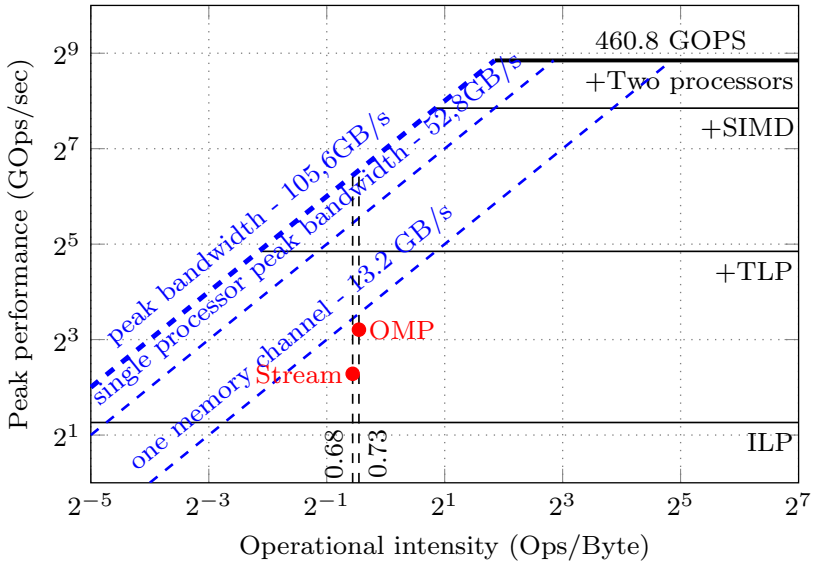


Figure 13.: Roofline - Node 662

As for the comparison between the two approaches, there is a noticeable difference in performance between the two, although the two approaches do not differ much in operational intensity. By assuming that both approaches take advantage of ILP, vector instructions (SIMD) (by instruction mix analysis) and since neither approach has significant use of both processors, what differentiates both approaches is the exploitation of TLP.

In conclusion, from the analyses performed and the VTune analysis, the cause for the performance difference between the two versions is the overhead caused by the management of the communication channels of the DSP version.

4.3 Final Results

The non-stream version, as already mentioned, has already presented results that demonstrate that the LA approach has the potential to surpass, in performance, the MySQL and PostgreSQL databases. In this work, the objective is to prove that the LA approach can go even further and for that, the performance of the OMP and DSP versions will be compared with the results obtained by the MonetDB database.

Execution time

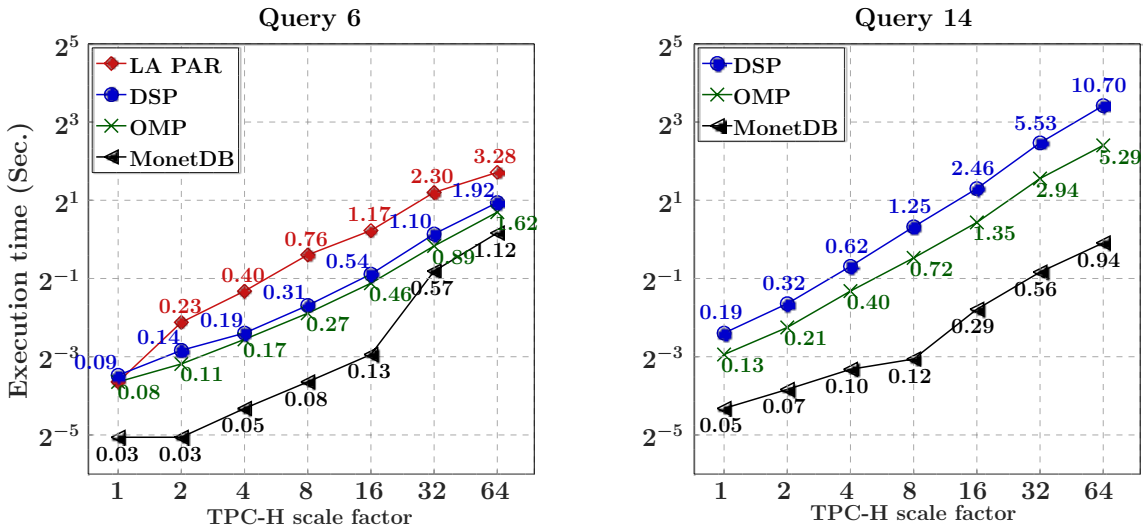


Figure 14.: Execution time query 6 and 14 - Node 662

In order to compare the various approaches, the tests were performed on the two implemented queries (query 6 and 14) on two different machines, node 662 and node 781. In addition, the performance of query 6 without *block* division (non-stream version) was

compared in order to verify the progress made so far. The same cannot be done with query 14 since no parallel version of this has been developed. It is also important to note that queries from the MonetDB database are executed sequentially, so the comparison between approaches is not completely fair. However, no parallel versions were found in this approach for proper comparisons.

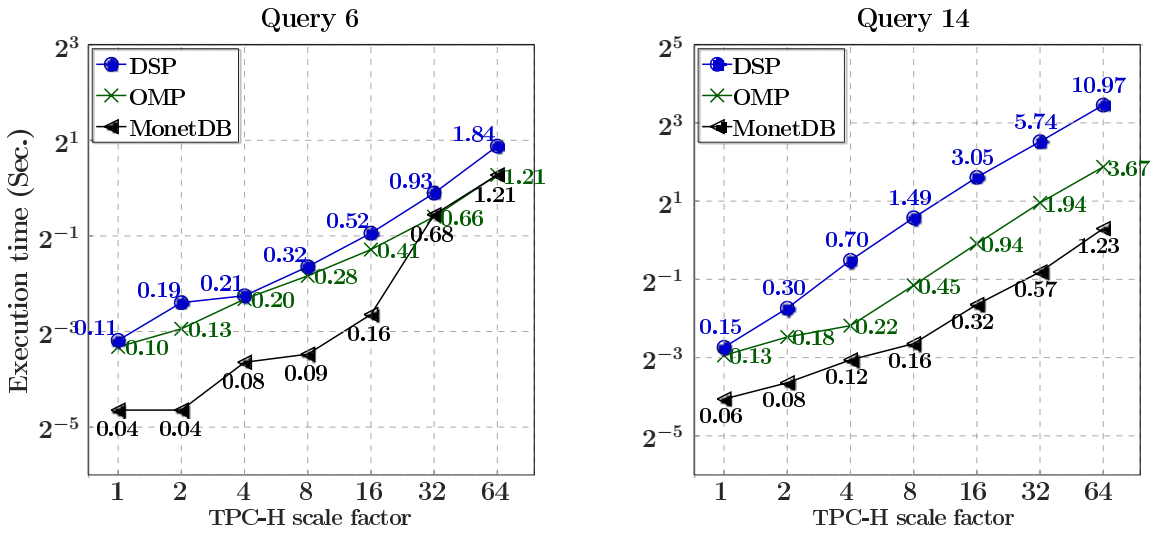


Figure 15.: Execution time query 6 and 14 - Node 781

By comparing the times obtained from query 6 on node 662, it can be seen that both the OMP version and the DSP version show significant improvements over the previous non-stream version. At node 781, the OMP version continues to show significant improvement, taking advantage of the increased bandwidth and processing power. However, the same is not true for the DSP version, which, despite the increase in hardware characteristics, does not show any performance improvement compared to the results obtained on node 662. Additionally, the non-stream version presents results similar to the DSP version so these have not been added to the chart.

As for query 14, the results obtained by the OMP and DSP versions on both nodes are far from the MonetDB database, presenting very significant differences. Especially the DSP version which has execution times up to ten times higher than MonetDB and up to three times higher than the OMP version. Moreover, unlike the OMP version, the DSP version does not show any performance improvements between the results obtained on nodes 781 and 662. Part of the apparent loss of performance between query 14 and query 6 is due to the characteristics of query 14, which has a more complex design, involving multiple strong dependencies, and a significantly larger load component than query 6.

In conclusion, the performance of the developed queries is not yet able to surpass the performance of the MonetDB database. Despite the noticeable improvements, especially of

the OMP version, memory limitations are a barrier that significantly restricts query performance.

Memory usage

Regarding memory usage, due to the optimisations and reuse of the *blocks*, the maximum amount of memory used throughout the execution as decreases drastically. As shown in figure 16, the OMP version only uses 1GB in both queries, while the DSP version uses up to 3,5GB. Despite the dynamic characteristics of the DSP version, the amount of memory used is acceptable for the size of the problem at hands.

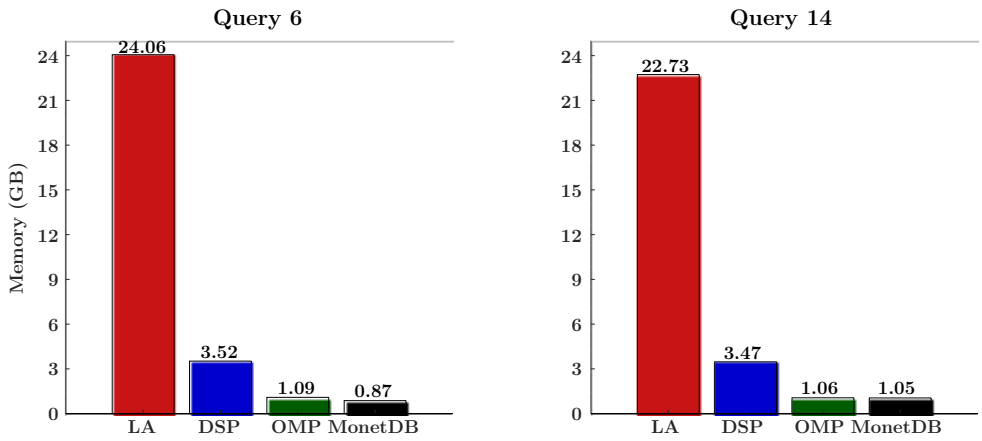


Figure 16.: Memory utilisation (scale $2^5 GB$)

It is important to note that the amount of memory used in OMP version does not depend on the size of the problem since this version has a constant amount of memory used during the execution. The amount of memory depends only on the query configuration and the number of threads used, as explained in section 2.1.2.

For the amount of memory used in the non-stream version of query 14, its sequential version was used, since the amount of memory used in a parallel version of the same approach would yield similar results.

5

CONCLUSION

This dissertation follows the developments made to compose the LAQ engine and the respective studies on the relationship between TLA and OLAP technology. These works have proved that the TLA approach is capable of outperforming databases based on RA that follow row-oriented data approach; however, it was unable to outperform databases based on RA that follows a column orientation. It is through this analysis and study of the LA queries that the untapped potential of this new approach and the limitations that the developed implementations have are highlighted. In particular, the deeper exploration of the parallelism present in LA operations and the possibility of circumventing, to some extent, the limitations imposed by strong dependencies.

In order to address these limitations and to take full advantage of the parallelism of LA operations, it is essential to use a framework that addresses these types of applications. For this purpose, the HEP-Frame was used, as it fits in this type of applications and since it presents impressive results in their exploration in parallel environments. However, although LA queries fit into the stream pipeline application category, HEP-Frame was not prepared to fully support the execution of its algebraic operations, requiring a change in its design, as shown in section 3.3. In addition, in the middle of the integration with HEP-Frame, a new stream version based on the DSP approach was developed to explore in greater detail the parallelism between LA operations and query behaviour in this approach, as shown in section 3.1.

Through the analysis of LA operations and the execution of the different versions, it was possible to identify and remove the main limitations of the three versions studied, both the previously developed OpenMP stream version and the new versions. These limitations mainly relate to the inefficient management of the data structures used to communicate between LA pipeline operations, which has been noticeably improved by their reuse at runtime; as referred to in section 3.2. All versions have benefited from these optimisations; however, the HEP-Frame still has some limitations due to the critical sections, causing a high overhead in the studied cases. In addition, the LA queries tested show no data filtering or significant differences in computational weights, showing little advantage in ordering the sequence in which operations are performed.

Finally, the results obtained from the OMP and DSP versions were compared with the performance of the MonetDB database. Although the results show that the current versions are still far from the performance presented by MonetDB, compared to previous versions the results obtained are notorious. Regarding memory utilisation, significant advances are also achieved, with a twenty fold reduction in memory utilisation for the benchmark scale factor 32 where both the OMP version and the HEP-Frame version have a constant behaviour independently of problem size.

5.1 FUTURE WORK

From the results and analysis made, it is noted that there is still room for improvement. As it is also necessary to evaluate a larger set of queries to identify the broader limitations. The following section presents some relevant points that may have a significant impact on the performance of the current system. In addition to these points, Afonso [2018] has already mentioned other important aspects to improve, which were not addressed in this paper.

5.1.1 *Hybrid memory environment*

One of the improvements that can bring significant performance gains is the exploration of the distributed memory environment. Although current versions scale well in a shared memory environment, they are unable to take advantage of a distributed environment. As can be seen from the roofline analysis, in figure 13, and query 6 execution times in figure 7.

In this sense, the queries already present a favourable structure for the paradigm shift. Since the dependencies to be taken into account are identified by the strong dependencies and the pipeline design themselves are flexible, their implementation should be straightforward. Mainly for the OMP version, requiring only the division of the *block* computation into distinct processes and the consequent message-passing of data regarding strong dependencies.

5.1.2 *Block size*

Another important aspect to consider is the impact of *block* size on LA operations. Especially in load operations, where there are the most significant differences, as shown in figure 17.

For example, load operations benefit from *blocks* of 128Ki elements, while Hadamard operations benefit from *blocks* of smaller size (32Ki); this effect is also visible in the L1 cache miss rate of these operations. These differences are essentially due to data caching

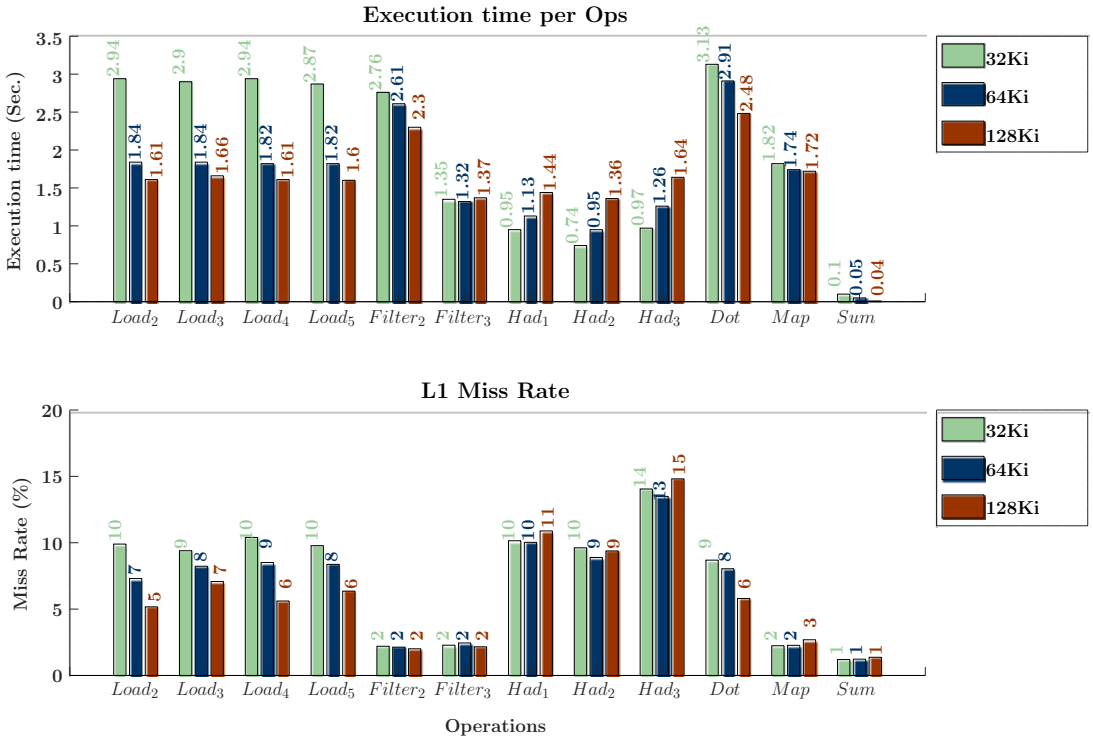


Figure 17.: Runtime and L1 Miss Rate per operation (Query 6) - Node 662

since operations are performed sequentially on the same dataset, taking advantage of its temporary location. It is dependent on the amount of data that is operated and its position in the pipeline, since operations that operate on multiple *blocks*, such as *Hadamard*, and operations that load memory have higher L1 cache misses. While operations that operate on a single *block* of data and are not loads operations have minimal L1 cache misses.

As for the miss rates of cache L2 and L3, although they present some differences, its relevance is negligible; either because of the low amount of L1 cache misses or because they have minimal differences between different *block* sizes. For example, the *Sum* operation, which has a very low L1 cache miss rate (1%), even though the L2 and L3 cache miss rate varies, only accounts for 1% of the total amount of memory access instructions.

Therefore, it is possible to increase or decrease the *block* size according to the characteristics of the operation to take better advantage of the various levels of the memory hierarchy; especially load operations, which benefit from significantly larger *block* sizes than processing operations.

So, the development of a new, more flexible *block* division model, both at the file system level and the pipeline level, can bring significant gains in query performance.

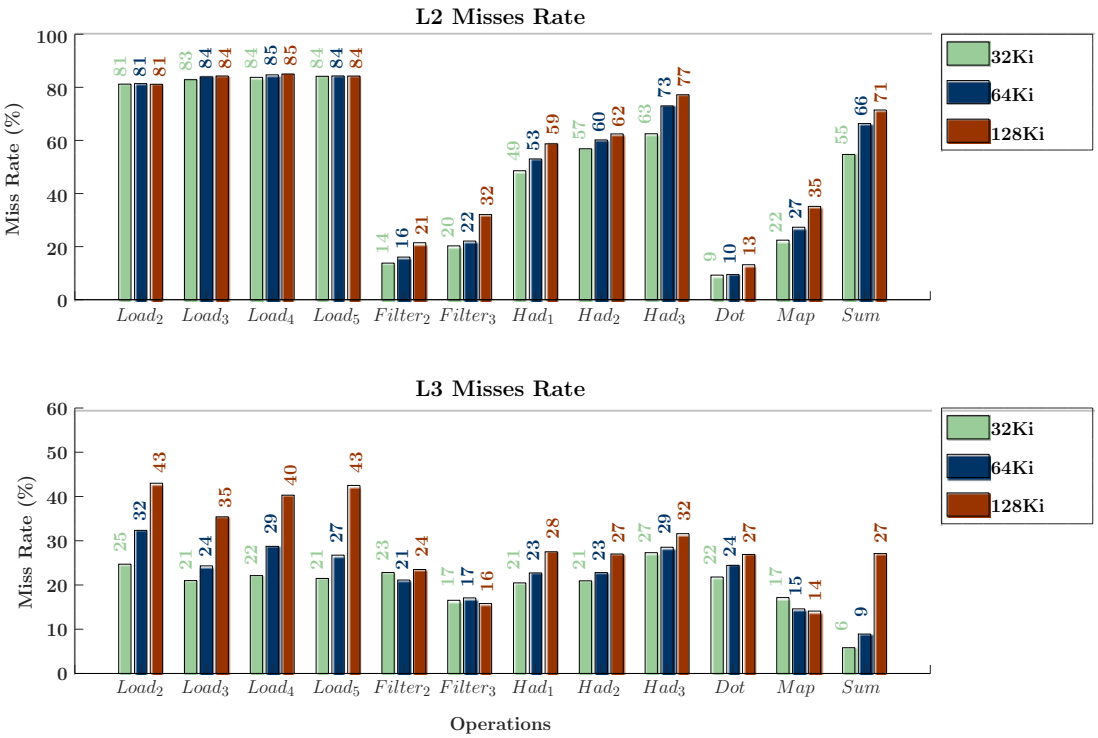


Figure 18.: L2 and L3 Miss Rate per operation (Query 6) - Node 662

5.1.3 *HEP-Frame*

The HEP-Frame version requires further analysis and the development of more queries to identify its limitations. Since, although query 6 has few strong dependencies and a pipeline prone to exploit operations-level parallelism, it has no data filtering and operations have no differences in computational weights. Besides, the adaptation of the Scheduler for this new version should also be considered as this can bring significant advantages in query execution.

BIBLIOGRAPHY

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

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*, 1809.00641, 2018.

Apache Group. Apache flink documentation. https://flink.apache.org, 2019a.

Apache Group. Apache storm documentation. https://storm.apache.org, 2019b.

O. A. R. Board. OpenMP application program interface. 2015. OpenMP Architecture Review Board, Tech. Rep.

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

Edgar Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. pages 97–104, 2004.

IBM. The strategic importance of OLAP and multidimensional analysis. Mar. 2011. IBM Software - Business Analytics.

S. Idreos, F. Groffen, N. Nes, S. Manegold, S. K. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, second edition, 2012. ISBN 9780321623218.

Hugo Macedo and José Oliveira. Matrices as arrows! a biproduct approach to typed linear algebra. *Mathematics of Program Constitution, Lecture Notes in Computer Science*, 6120:271–287, 2010.

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

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

Oracle Corporation. MySQL reference manual oracle, https://dev.mysql.com/doc/refman/8.0/en/, 2019.

André Pereira, António Onofre, and Alberto Proença. HEP-Frame: A software engineered framwork to aid the development and efficiency multicore execution of scientific code. *Proc. Int. Conf. High Performance Computing & Simulation*, 2015.

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

PostgreSQL Dev. Group. PostgreSQL documentation. https://www.postgresql.org/docs/11/index.html, 2019.

F. Rademakers, P. Canal, B. Bellenot, O. Couet, A. Naumann, G. Ganis, L. Moneta, V. Vasilev, A. Gheata, P. Russo, and R. Brun. ROOT. https://root.cern.ch//, 2012.

J. Reinders. Vtune performance analyzer essentials. 2017. California; Intel Press.

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

S. Browne et al. PAPI: A portable interface to hardware performance counters. *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

V. M. Weaver. Linux perf_event features and overhead. *Second International Workshop on Performance Analysis of Workload Optimized Systems*, pages 79–81, 2013.

Dag Wieers. Dstat: Versatile resource statistics tool. http://dag.wiee.rs/home-made/dstat/.

Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. Revisiting the design of data stream processing systems on multi-core processors. 2017. SAP Innovation Center Singapore, National University of Singapore, INRIA, SAP SE Walldorf.

A

FRAMEWORKS DESIGN

A.1 LAQ ENGINE DESIGN

The LA database has several independent modules, as shown in figure 19. The focus of this work is the transition between the query processor module and the runtime compiler, more precisely in the formulation of the query code in C ++.

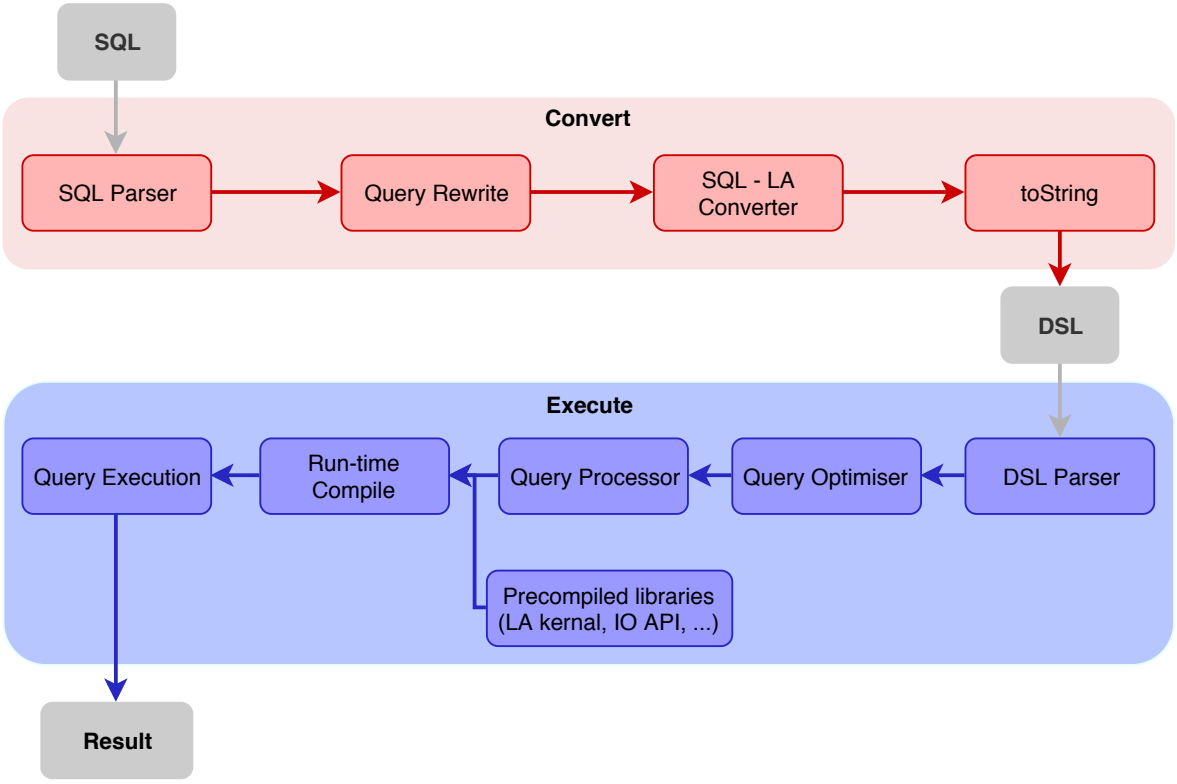


Figure 19.: LAQ - The framework structure (from Afonso [2018])

Therefore, the Query processor and Precompiled libraries modules were modified throughout this dissertation, as presented in section 3.2. In addition, it is important to mention the

importance of the Query Optimiser module, since it is related to the developed HEP-Frame version, although it has not been implemented.

Query Optimiser

This component is responsible for ordering LA operations considering their dependencies; the objective is to build a pipeline of operations that allows a more efficient query execution. For HEP-Frame versions, the pipeline order can be taken from previously executed runs.

Query Processor

This module transforms LAQ code into C ++ executable code; in this process, the LA operation libraries are imported as well as the data structures used. This module has not been updated to the point of supporting DSP or HEP-Frame code generation; however, this step does not require much effort to complete due to the modularity of the components.

Precompiled libraries

As its name suggests, this module contains the precompiled code of the components needed to create the executable queries. This is one of the modules that has been covered in this dissertation and which has been made the necessary optimisations to improve the query performance; especially the implementation of LA operations and the implementation of matrix data structures and their blocks.

A.2 HEP-FRAME DESIGN

The HEP-Frame is divided into two modules, the developed application, with its functions and data structures, and the framework that provides the tools and set of functions that allows the creation and execution of the application in this environment.

The changes made to the framework are restricted to the Core Framework component and, as noted in section 3.3, Consist of changing the functions of the *DataAnalysis* and *DataReader* classes. In addition, it is important to note that interfaces and generation tools do not yet support LAQ application generation, especially the generation of data structures.

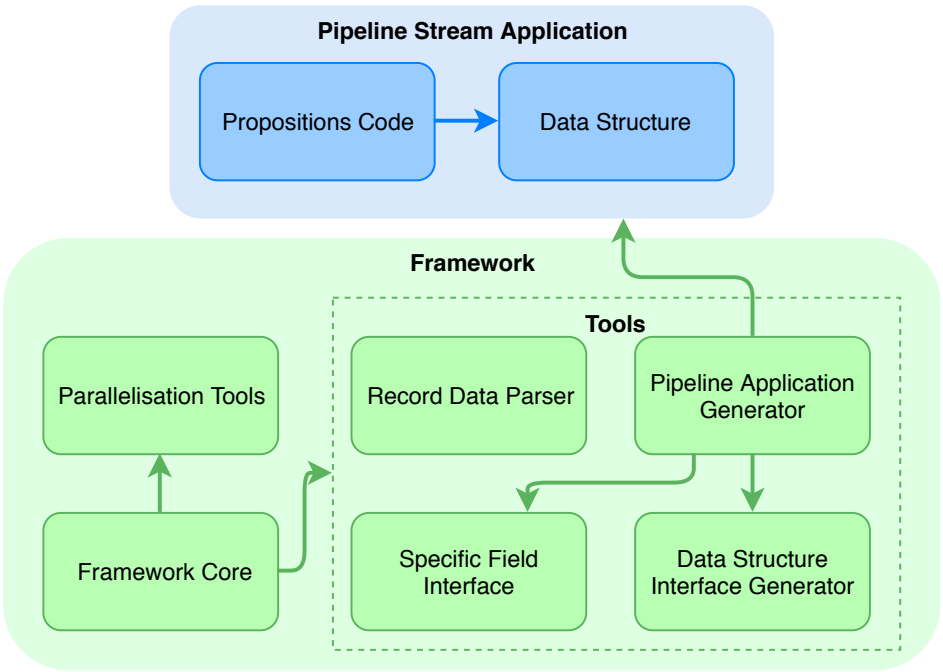


Figure 20.: HEP-Frame - The framework structure (from Pereira et al. [2015])

B

TPC-H QUERIES - SQL, LAQ AND DSP DESIGN

B.1 QUERY 14

```sql
select
    100.00 * sum (
    case
        when p_type like 'PROMO%'
            then l_extendedprice * (1 - l_discount)
        else 0
    end) / sum (l_extendedprice * (1 - l_discount)) as promo_revenue
from
    lineitem,
    part
where
    l_partkey = p_partkey
    and l_shipdate >= date ':1'
    and l_shipdate < date ':1' + interval '1' month;
```

Listing B.1: TPC-H query 14 - SQL code

```
A = filter(lineitem.shipdate>="1995-09-01" AND lineitem.shipdate<"1995-10-01")
B = lift( lineitem.extendedprice * (1 - lineitem.discount) )
C = hadamard( A, B )
D = filter( match( part.type , "PROMO.*" ) )
E = dot( D, lineitem.partkey )
F = hadamard( C, E )
G = sum( F )
H = sum( C )
I = lift( 100.00 * G / H )
return ( I )
```

Listing B.2: TPC-H query 14 - LAQ code

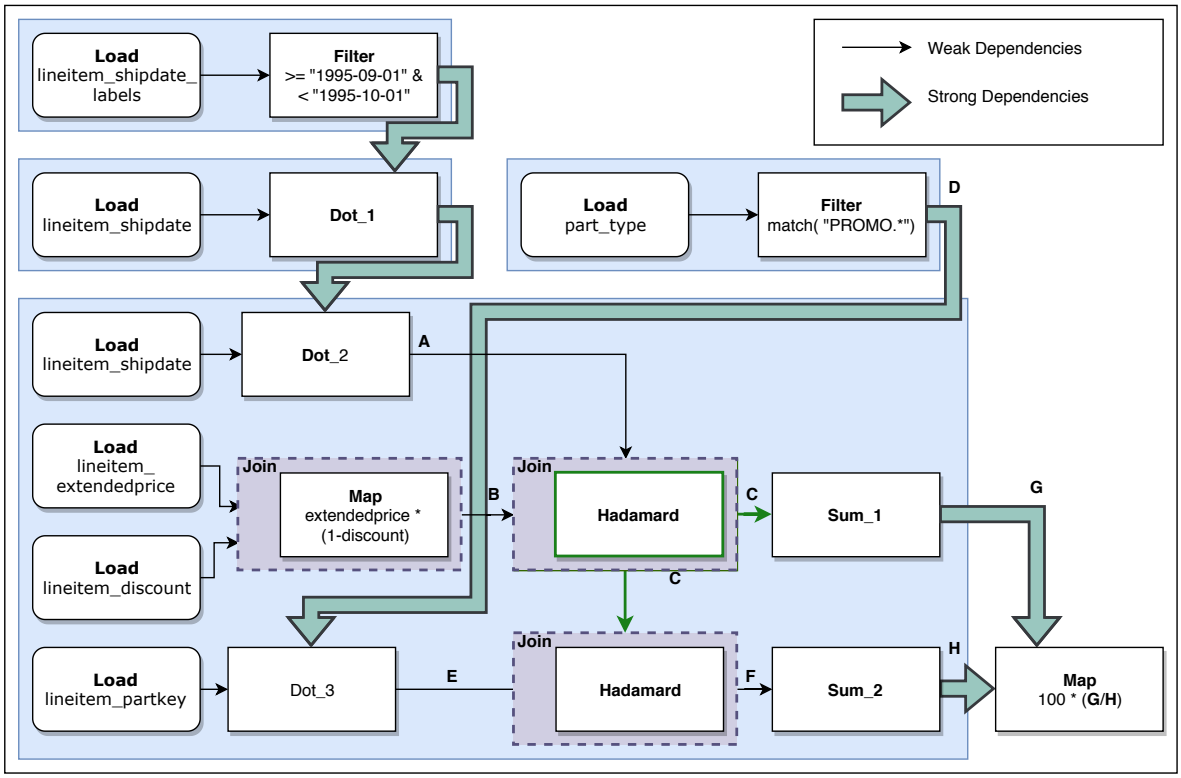


Figure 21.: TPC-H Query 14 - DSP design