Efficient Linear Algebra-based OLAP queries

Bruno Ribeiro*, Fernanda Alves† and Gabriel Fernandes‡
Parallel and Distributed Computing
Formal Methods in Software Engineering
Universidade do Minho
Email: *a73269@alunos.uminho.pt, †a64365@alunos.uminho.pt, ‡a71492@alunos.uminho.pt

***Abstract*—Online Analytical Processing (OLAP) systems selectively extract data and display results with different points of view. The mainstream approach uses Relational Algebra (RA), but a new approach with Linear Algebra (LA) is now challenging the RA, with fast and robust outcomes. This approach focuses on the efficiency of LA operations by validating and testing with the TPC-H Benchmark suite and comparing its performance with PostgreSQL 9.6. The approach will also explore parallel query execution. Linear Algebra development requires handling data labels, data representation, operations, and extract useful properties and multi-precision, to efficiently represent, access information. Measured execution times for different dataset sizes on two crucial queries in TPC-H (3 and 6) show consistent faster execution times of the LA-OLAP.**

I. Introduction

The growth of technological data adds an extra effort from the already used and efficient Online Analytical Processing (OLAP). These systems embrace these challenges not only in software but also in hardware.

This term, coined by Edgar F. Codd [1], refers to *Online Analytical Processing*, which is a technology that uses a multidimensional view of aggregate data to provide quick access to information for the purposes of advanced analysis.

The Relational Algebra (RA) is the current solution for big data storage. However, this approach is not perfect. RA lacks algebraic properties on the most common operations and does not provide qualitative and quantitative proofs for all the relational operators. Also, it is hard to frame OLAP operators in Codd's Relational Algebra.

A new approach is required to address large databases and to efficiently query them. This work focuses on the development of an efficient Typed Linear Algebra approach, encoding all the necessaries operators in terms of LA (matrices) applied on a DSL structure, following previous work from Filipe [2] and Rogério [3].

A parallel work has been done [4] aiming to translate SQL query to the DSL structure. This conversion must be capable of calculating the OLAP operation that can be completely formal both on the quantitative and qualitative side.

When comparing different components/paradigms, we can use self-developed metrics, or we can use a more robust, complete and credible way to do so: a database benchmarking tool to set a common ground to that comparison, the TPC-H [5], which provides a suite of 22 queries relevant in an OLAP and industrial context. TPC-H aims to deal with large volumes of data and with complex queries, but this work is limited to dataset sizes that fit in main memory.

II. Efficient Computing

The process of developing software is not simple, so, when one of its requirements is efficiency, the complexity of its development increases significantly. To this type of software development, we call High-Performance Computing.

Eugene Loh said in the article *The Ideal HPC Programming Language* [6] that:

"In HPC, the mindset is usually to program for performance rather than programmability even before establishing whether a particular section of code is performance sensitive or not"

In fact, the process of building efficient code needs to be methodical to achieve better performance as we can, and the developer must know the current features in HPC.

One of the things that are important to know is the *Instruction Level Parallelism*, which indicates how many instructions of a program can be executed simultaneously. In that level, the current compilers have a significant importance to achieve better performance, which does a lot of work for us.

Another thing to know is how memory is organized, that is *Memory Hierarchy Levels*. In fact, one way to achieve better performance is to improve the memory accesses of a program, reducing cache misses and take advantage of prefetching mechanisms from both operating system and hardware. Thus, it is necessary to take a closer look at data structures, preferring structures of arrays rather than arrays of structures, which provides better data alignment and assist *Data Level Parallelism* by using vector instructions.

Considering data level parallelism, current processors architectures offer vector instructions that perform at most eight 32-bit floating points operations (or four 64-bits floating point operations) per clock cycle, with 256 bits vector registers. Current architectures are moving to AVX-512, working with vector instructions of 512 bits, which doubles the size of that registers and throughput.

Another key factor is the available parallelism in CPU devices: current devices have multiple processors providing multiple independent processing units for executing in parallel several threads or processes.

Today, we assist to the growth of co-processors like graphics processing units (GPUs), which are adapted to scientific computation to execute parallel algorithms, and the recent *many-core* processors, like Intel Knights Landing, are distinct from traditional multi-core processors in the way that they are optimised for a higher degree of explicit parallelism.

These emerging technologies offer the possibility to develop parallel programs that use multiple cores to perform the work simultaneously and more efficiently.

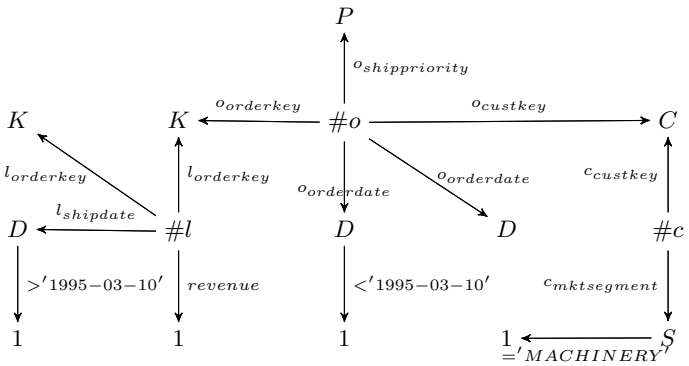
III. Linear Algebra Strategy

To introduce our approach, we will start by explaining an execution of a query. Our queries are expressed in a *Domain-Specific Language* (DSL) which contains the instructions and operations to execute the query, that derive from linear algebra notation. This process is a case of study from João Afonso and João Fernandes [4], which are also responsible to translating SQL queries into a DSL query structure, following linear algebra theory and principles.

Although we start from our DSL, we will show what is the equivalent in SQL language, just to explain what this query does:

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

This SQL query can be described by a type diagram (see below), which is then converted to the following DSL query:




```
A = filter_to_matrix( o_orderdate, <'1995-03-10' )
B = filter_to_vector( c_mktsegment, ='MACHINERY' )
C = filter_to_vector( l_shipdate, >'1995-03-10' )
D = dot( B, o_custkey )
E = krao( l_orderkey, C )
F = krao( A, D )
G = krao( F, o_shippriority )
H = dot( G, l_orderkey )
I = krao( E, H )
J = map( \x -> 1 - x, l_discount )
K = hadamard( l_extendedprice, J )
L = krao( I, K )
M = bang( sum, L )
```

The structure at the DSL query follows an imperative language pattern.

There are two things that we notice immediately: a `filter` operator that filters data that falls under a certain condition, and some operations are applied to structures that represent just the attributes and not the whole table.

The main goal of this work was to efficiently take advantage of vector capabilities and parallelization, adapting the data structures (matrices and vectors) .

We start by identifying and list all attributes needed to perform this query. To ease data association, the number after the attribute name is the corresponding column in their table:

- Customer table:
  - `custkey` (#1)
  - `mktsegment` (#7)
- Orders table:
  - `orderkey` (#1)
  - `custkey` (#2)
  - `orderdate` (#5)
  - `shippriority` (#8)
- Lineitem table:
  - `orderkey` (#1)
  - `extendedprice` (#6)
  - `discount` (#7)
  - `shipdate` (#11)

Tables I, II and III contain data from the customers, orders and lineitems, just for the previous selected attributes.

TABLE I: Customer table

#1 c_custkey	#7 c_mktsegment
1	MACHINERY
2	AUTOMOBILE
3	AUTOMOBILE
4	MACHINERY
5	HOUSEHOLD
6	AUTOMOBILE
7	AUTOMOBILE
8	BUILDING
9	MACHINERY
10	HOUSEHOLD
15	BUILDING
20	AUTOMOBILE

TABLE II: Orders table

#1 o_orderkey	#2 o_custkey	#5 o_orderdate	#8 o_shippriority
1	4	1991-01-02	0
2	6	1996-12-01	0
3	9	1993-10-14	0
4	10	1995-10-11	0
5	15	1994-07-30	0
6	20	1992-02-21	0
7	1	1996-01-10	0
32	9	1995-07-16	0
33	4	1993-10-27	0
34	6	1998-07-21	0

TABLE III: Lineitem table

#1 l_orderkey	#6 l_extendedprice	#7 l_discount	#11 l_shipdate
1	21168.23	0.04	1996-03-13
1	45983.16	0.09	1996-04-12
1	13309.60	0.10	1996-01-29
1	28955.64	0.09	1996-04-21
1	22824.48	0.10	1996-03-30
1	49620.16	0.07	1996-01-30
2	44694.46	0.00	1997-01-28
3	54058.05	0.06	1994-02-02
3	46796.47	0.10	1993-11-09
3	39890.88	0.06	1994-01-16

To load and handle that data, those values should be structured in matrices/vectors. We have two types of matrices: projection matrices and measure matrices. For both types, we will consider that each column represents a record.

In projection matrices, each row represents a label, such as a name or a date. If we say, for instance, that the record X has a customer ID Y, in the matrix, a one should be at the position corresponding to the column associated with that record, and the row associated with that label, in this case, the customer ID. At the same time, zeros indicate that a record has not that label. We may refer to these matrices as bitmaps.

In a record from a purchase with only one buyer/customer, for instance, if we have 1000 customers, we will have a column with 999 zeros.

The sparsity of these matrices has an impact on performance. We can avoid storing all those irrelevant zeros by storing it in an alternative way. Since this is an implementation detail, we will investigate further on sections ahead.

For the measure matrices, we will have values like integers and decimals/floats hence the name of these types of matrices. They are used to indicate quantities, prices, and anything we want to measure. If we just want to associate a measure to a attribute, we do not need to have labels. So these matrices are usually vectors, where we associate a measure to a record, which is an integer/double in a column.

To represent some of these attributes in their matrices, we will use a simple measure vector, from the `lineitem` table with is `l_discount` attribute.

$$\texttt{l\_discount} = \begin{bmatrix} 0.04 & 0.09 & 0.10 & 0.09 & 0.10 & 0.07 & 0.00 & 0.06 & 0.10 & 0.06 \end{bmatrix}$$

To represent the `l_orderkey` from `lineitem` table we will use a bitmap matrix:

$$\texttt{l\_orderkey} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since `l_orderkey` is a foreign key relating to `o_orderkey`, with ten distinct `o_orderkey`, so ten different labels. Since there are ten records in `lineitem` table, the resulting matrix is 10×10. In `lineitem` table, there are seven records with the `l_orderkey` 1, one record with `l_orderkey` 2 and three records with `l_orderkey` 3. The `l_orderkey` bitmap matrix is shown above.

Record one has the `l_orderkey` corresponding to row one. To retrieve and store labels from/to rows, an auxiliary mechanism is required (detailed later). For now we will focus on the correct representation and handling of matrices.

Given these two examples of data represented by matrices in the LA approach, the sequence of the thirteen steps in the DSL query are detailed below.

The first operation in the DSL is a filter:

```
A = filter_to_matrix( o_orderdate, <'1995-03-10' )
```

`A` is a bitmap matrix containing the result of all records of `o_orderdate` attribute that passed the condition of being less than `'1995-03-10'`. Below we can see matrix `A`, and can be confirmed by the values from Table II.

$$\texttt{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Next we have another filter:

```
B = filter_to_vector(c_mktsegment, ='MACHINERY')
```

It results in a bitmap vector `B` which result is all records of the customers whose attribute has a `c_mktsegment` equal to `MACHINERY`. Thus, the result vector `B` is shown below, in which can be confirmed by the values in Table I.

$$\texttt{B} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Then it comes:

```
C = filter_to_vector(l_shipdate, >'1995-03-10')
```

It returns a bitmap vector `C` containing all the records from `lineitem` that have a `l_shipdate` greater than `1995-03-10`, those are the items that were shipped after the 10th of March 1995. Resulting in:

$$\texttt{C} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Let us focus on the next step of the DSL:

```
D = dot(B, o_custkey)
```

Executing a dot product between a 1×12 vector and a 12×10 matrix, mathematically, will result in a 1×10 vector. Bear in mind that we have twelve different customers, so the `customers` table has twelve records, and the `orders` table

has ten records, as we can see from Table I and Table II, respectively.

The result of this step are the orders made by customers that have `c_mktsegment` as `MACHINERY`, in which results in a bitmap vector `D`. That dot product, at once, joined the `orders` and `customers` table and filtered it regarding an attribute from the `customers` table. The result is:

$$\mathtt{D} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Now we have a Khatri-Rao operation:

```
E = krao(l_orderkey, C)
```

This instruction applies the Khatri-Rao product to the `l_orderkey` matrix and `C` vector. It selects all the records of `l_orderkey` that passed the condition represented by the bitmap vector C explained before. Matrix `E` is presented below:

$$\mathtt{E} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Since the bitmap vector `C` only filters the first seven records of the attribute `l_shipdate`, Khatri-Rao operation will select those first seven records, as shown above.

After that, we have:

```
F = krao( A, D )
```

It applies the Khatri-Rao product between matrices `A`, which is a bitmap matrix with all order dates according the given condition, and `D`, which is a bitmap vector with all orders from customers with `c_mktsegment` as `MACHINERY`. Then, it results in a bitmap matrix `F` with all orders' dates selected according to the criteria imposed by the vector `D`. The result matrix is:

$$\mathtt{F} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Then another Khatri-Rao operation appears:

```
G = krao( F, o_shippriority )
```

It executes the Khatri-Rao product to produce the bitmap matrix G with all labels combinations between the bitmap matrices F and `o_shippriority`, in which reproduces the `GROUP BY` clause between the attributes `o_orderdate` and `o_shippriority` imposed in the SQL shown previously. This kind of operation is called projection.

Since the `o_shippriority` attribute only has one distinct label, it is a bitmap vector full of ones, and by applying the Khatri-Rao product, the bitmap matrix G will be equal to the bitmap matrix F previously shown.

A dot product instruction appears in this step:

```
H = dot( G, l_orderkey )
```

It is similar to the dot product presented in the calculation of the vector D, but now the is applied to two matrices. The first one is the calculated in the previous step, and the second one is the bitmap matrix that represent all order keys in the lineitem table. Since both matrices have dimensions 10×10, the result matrix will have the same size. Thus, this step performs a join operation between the tables `orders` and `lineitem`, resulting in the following matrix:

$$\mathtt{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The next instruction is:

```
I = krao( E, H )
```

It is similar to the operation that calculates bitmap matrix `G`, in which performs a projection operation between matrices E and `H` using the Khatri-Rao product. Thus, will be performed a `GROUP BY` clause between the attributes `l_orderkey` (presented in matrix `E`) and the attributes `o_orderdate` and `o_shippriority` (presented in matrix `H`). Since each matrix has the size of 10×10, the result bitmap matrix `I` has the size of 100×10, because all combinations between all the labels of these three attributes generates an huge matrix (100×10) with only six non-zeros in the first row:

$$\mathtt{I} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

The next step a different operator:

```
J = map(\x -> 1-x, l_discount)
```

It performs an algebraic operation, receiving a lambda function and the vectors/matrices needed, and through each value of the measure vector `l_discount`, it applies the lambda function. In this case, it will subtract one by every element of the vector, which results in a measure vector `J` that contains the percentage to pay of the full price. The resulting vector is:

$$\mathtt{J} =$$

$$\begin{bmatrix} 0.96 & 0.91 & 0.90 & 0.91 & 0.90 & 0.93 & 1.00 & 0.94 & 0.90 & 0.94 \end{bmatrix}$$

The next step is an Hadamard operation:

```
K = hadamard( l_extendedprice, J )
```

It performs an element-wise multiplication between the vectors `l_extendedprice` and `J`, calculated in the previous step. This operation results in a measure vector `K` containing the price paid by each item. Since this step perform a multiplication between all elements of both vector, the Hadamard product could be used instead of the `map` operation. Note that the presenting result has four decimal places although our implementation deals with double precision. Although it is an horizontal vector, we represent it here as vertical to better fit. Vector `K` is shown below:

$$\mathtt{K} = \begin{bmatrix} 20321.5008 \\ 41844.6756 \\ 11978.6400 \\ 26349.6324 \\ 20542.0320 \\ 46146.7488 \\ 44694.4600 \\ 50814.5670 \\ 42116.8230 \\ 37497.4272 \end{bmatrix}$$

The final Khatri-Rao operation is:

```
L = krao( I, K )
```

It applies a Khatri-Rao product between matrix `I` and vector `K`. This operation will inject the measured values (`K`) in the projection matrix `I` with the `GROUP BY` clause completely performed.

$$\mathtt{L} = \begin{bmatrix} 20321 & 41844 & 11978 & 26349 & 20542 & 46146 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

Last but not least, the instruction:

```
M = bang( sum, L )
```

It performs the aggregate function horizontally, resulting in the final vertical vector that represents the result of the query. After the grouping of data by attributes, we have a 100×10 matrix that was the result of multiple Khatri-Rao products that increased the matrices size, as seen before. To get the sum of each measure per label, the revenue, a operation that sums all the elements in each row need to be performed. This is implemented using the `bang` operator, which applies an arithmetic operation to all the elements in every row. Vector `M` is shown below:

$$\mathtt{M} = \begin{bmatrix} 167183.2296 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$$

Every other vector value was zero except the first one, which means that we only got one row from the query as a result, with a revenue of 167183.2296.

Since the previous Khatri-Rao products grouped the data by `l_orderkey`, `o_orderdate` and `o_shippriority`, combining them all, the single result appears in the first row, where its `l_orderkey`, `o_orderdate` and `o_shippriority` have the same value as their first records in their respective tables.

The post-query step to take is to retrieve the labels corresponding to the grouped attributes. Table IV shows this activity.

TABLE IV: Label retrieval

Attribute	Row	Label
`l_orderkey`	0	1
`o_orderdate`	0	`1991-01-02`
`o_shippriority`	0	0
Final Matrix Value		
167183.2296		

To execute this query in a SQL database system, the result would be:

```
l_orderkey | o_orderdate | o_shippriority | revenue
-----------+-------------+----------------+-------------
         1 | 1991-01-02  |              0 | 167183.2296

(1 rows)
```

This proves that the implementation is valid and completely functional.

IV. Design & Implementation

To handle the needed data and operations along the whole development, some design and implementations were formulated during the process. This section presents some relevant implementation details related to:

- Data Representation
- Data Labels
- Operations
- Class of operators

A. Data Representation

Since this a linear algebra approach, virtually every data structure is a matrix. Usually, these are bitmaps where one means the existence of a record and zero means the opposite.

Almost all bitmap matrices in a database have few non-zeros values, i.e., they are highly sparse. To store this information

as a dense matrix leads to a waste of memory space and a negative impact on performance. To overcome this, the best approach is to select a matrix representation format that efficiently stores sparse matrices.

Since the records are represented by columns in the matrix, the Compressed Sparse Column (CSC) [7] format seems to be the most appropriate.

The CSC format has three arrays: `values`, `row_index` and `column_pointer`: the `values`, with the values of each non-zero element to be represented; the `row_index`, with the row indexes of each non-zero value; the, `column_pointer` array, with size $N_{columns}+1$, specifies in the array `values` the position of the non-zero elements of a given column, together with the number of non-zero values of each column by subtracting `column_pointer[column]` by `column_pointer[column+1]`. This array is used to iterate through all columns, which is a key feature to process record after record in a matrix.

We can have a better understanding in Figure 1.

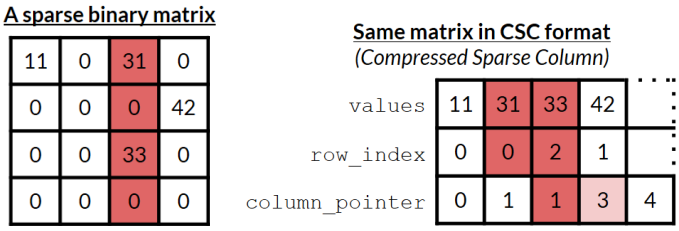


Fig. 1: Representation of a sparse matrix in CSC format

One of the problems found during the execution of queries was that if a query have many Khatri-Rao operations, will result in a matrix expanded in number of rows, and it is possible that the current representation of the row index values with long int (64 bits) will not be large enough to represent indexes large that 2^{64} (10^9). The same applies when the number of records is that large. In these cases, operations will require the use of multi-precision libraries.

B. Data Labels Representation

One of the challenges of the LA-OLAP approach is how to represent the data labels of the different attributes of a table. This labelling is relevant in the query execution, since filter operations operates on the value of labels, and the tables with the queries results depend from them.

The LA approach deals with matrices, where each of them represents an attribute from a table. Each row of an attribute matrix represents the different labels of that attribute, and the number of rows of that matrix is equal to the number of the distinct labels that the attributes has.

The approach uses the row index as the label ID and associates this to the label value, and vice-versa. This way it is possible to identify and retrieve what each row/label value is referring to.

The solution uses hash tables to establish these correlations. For performance matters, it uses a double hashing strategy to achieve a complexity of $O(1)$ on both sides of the association. If only one hash table was used, one of the sides of the association would have complexity of $O(N)$. Both sides support the association between label values and IDs for data loading and the association between IDs and label values for filter operations by labels, to build the result table of a query.

Consider an attribute `orderdate` that corresponds to a date of a specific order request and an example date like `'1995-01-01'` for the first record of an orders table; the approach do the following associations:

```
// Key <-> Value

hashtable_ids[20] = '1995-01-01'
hashtable_labels['1995-01-01'] = 20
```

The GHashTable API from GLib library [8] implements this double hashing solution, and both variables of IDs and label values are shared by both hash tables, to speedup memory accesses.

- Pros:
  - Supports foreign keys
  - Can retrieve value from key and vice-versa
  - Useful in pre and post query processing
- Cons:
  - Larger memory requirements, due to double hashing strategy

This disadvantage is due to, for each attribute such as `orderdate`, two GHashTable data structures are required, which may add a slight memory overhead. However, it is preferable to have a larger structure to increase performance, than to save memory space with an inefficient performance.

Figure 2 shows the pro of these structures to support foreign keys. The solution is to share these structures between primary and foreign keys, ensuring that always the primary key attributes are loaded first, keeping the referential integrity rule always true.

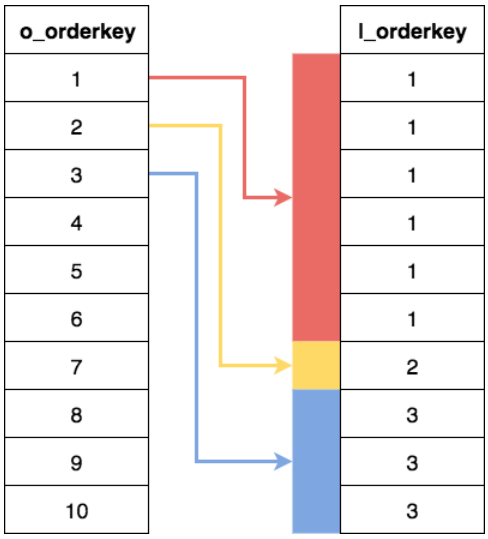


Fig. 2: Association between primary and foreign keys.

C. Operations

This subsection details the different operations needed by the LA approach. It includes the following operations:

- Dot product
- Hadamard product
- Khatri-Rao product
- Attribute filter
- Matrix bang
- Element-wise map product

These six operations can currently solve the queries needed to prove the LA approach. More specific cases in new queries may require the development of new operations.

Simple matrices A and B will be used to illustrate each operation. Some operations are applied only for sparse matrices, so we will use both a sparse and dense matrix C. C_D represents a C dense matrix and C_S a C sparse matrix.

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 6 \\ 2 & 0 \end{bmatrix}$$

$$C_D = \begin{bmatrix} 1 & 2 & 2 & 4 \end{bmatrix}_D \qquad C_S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_S$$

1) Dot Product: The dot product is the most common product in linear algebra. This product is used to matrix-matrix, matrix-vector and vector-matrix multiplication, as the following example (matrix-matrix):

$$A.B = \begin{bmatrix} (1\times1)+(2\times2) & (1\times6)+(2\times0) \\ (0\times1)+(5\times2) & (0\times6)+(5\times0) \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 10 & 0 \end{bmatrix}$$

The dot product receives a matrix A with dimensions $(m \times n)$ and a matrix B with $(n \times p)$ dimensions, and calculate each element of a result matrix C, where each element c_{ij} is given by multiplying the elements A_{ik} (across row i of matrix A) by the elements of B_{kj} (down column j of B), for k from 1 to m summing the results over k.

The same is applied to vectors, being the only restrictions that the number of columns of matrix A as to be equal to the number of columns of matrix B.

2) Hadamard Product: The Hadamard product is a operation that takes two matrices of the sames dimensions, and produces another matrix where each element ij of a result matrix C is the product of elements ij of the original two matrices. This product is associative, distributive and also commutative.

$$A \times B = \begin{bmatrix} 1\times1 & 2\times6 \\ 0\times2 & 5\times0 \end{bmatrix} = \begin{bmatrix} 1 & 12 \\ 0 & 0 \end{bmatrix}$$

In our approach, this operation is only used to multiply vectors with the same dimensions $(m \times n)$, in which the dimensions m and n depends if the vectors are horizontal or vertical.

3) Khatri-Rao Product: The Khatri-Rao product multiplies each line of a matrix A by the whole matrix B, which results in a matrix expanded in rows. Consider that a matrix A with dimensions $(n \times m)$ and a matrix B with dimensions $(k \times m)$, the Khatri-Rao product results in a result matrix C with dimensions $(n \times k) \times m$.

$$A \bigtriangledown B = \begin{bmatrix} 1\times1 & 2\times6 \\ 1\times2 & 2\times0 \\ 0\times1 & 5\times6 \\ 0\times2 & 5\times0 \end{bmatrix} = \begin{bmatrix} 1 & 12 \\ 2 & 0 \\ 0 & 36 \\ 0 & 0 \end{bmatrix}$$

Note that the number of columns of the input matrices must be equal, and the number of rows of the result matrix C is equal to the multiplication of the number of rows of both matrices A and B. Thus, the result matrix C have all combinations between each lines of both matrices A and B.

4) Attribute Filter: This operators is used to apply a filter to the labels of an attribute by one specific condition, resulting in a sparse vector with all values that passed the condition.

Consider the following scenario: we have four labels (1,2,2,4) represented in C_D and in C_S, for the sparse format. If we want to select the labels greater than or equal to 2, our filter will perform like:

$$filter(C_S, \geq 2) = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} . C_S =$$
$$\begin{bmatrix} (0\times1)+(1\times0)+(1\times0) & (0\times0)+(1\times1)+(1\times0) \\ (0\times0)+(1\times1)+(1\times0) & (0\times0)+(1\times0)+(1\times1) \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}$$

This operation has the condition that the number of columns of filter vector must have the same number of rows of the attribute matrix applied to the filter.

5) Matrix Bang: This operation is very useful to aggregate every element in a row or a column, i.e., to apply an aggregation function to the measure values. This aggregation functions consists in applying sum, count, average, min or max functions to every element in a row/column. Thus, this operation is called a *bang* because it *smashes* a matrix into a vector.

To simply exemplify this operation, we will use a matrix A and aggregate the rows using sum operation, as following:

$$bang(sum, A) = \begin{bmatrix} 1+2 \\ 0+5 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \end{bmatrix}$$

Intuitively, the result vector will have the same number of rows/columns (depending on the applied direction) of the input matrix in which this operation was applied, resulting in a vector with the calculated measures.

6) Element-wise Map Operator: The element-wise map operator is used when is necessary to apply an arithmetic operation over each elements of one or more vectors. All the vectors used by this operation must have the same size and,

as a result, a new vector with the same size is generated with the results of the arithmetic operation over each elements.

Consider the following example: we have a dense vector and we need to duplicate each element and add his square. This operator will have the following behaviour, note that both vectors are horizontal, but for a better visualisation here we represented as vertical vectors:

$$map(\backslash x \to 2 \times x + x^2, C_D) =$$

$$= 2\times \begin{bmatrix}1\\2\\2\\4\end{bmatrix} + \begin{bmatrix}1^2\\2^2\\2^2\\4^4\end{bmatrix} = \begin{bmatrix}2\times 1+1\\2\times 2+4\\2\times 2+4\\2\times 4+16\end{bmatrix} = \begin{bmatrix}3\\8\\8\\24\end{bmatrix}$$

This operation appears as a solution to achieve a better performance in algebraic operations, to remove unnecessary intermediate operations to perform the whole operation, removing the memory used for these intermediate vectors as well.

D. Class of operators

In this LA approach, a set of different operations is needed to execute queries. Not all of them are easy to implement, and most of them have slightly variations. So, these variations could be explored to build specific case operators to achieve a better performance in queries execution, but this could lead to many different implementations of the same operations. To gather all the differences in each operator, a class of operators is built, in which each operation implements different methods, that are specific operators.

Several critical issues with a severe impact on performance were identified and efficient solutions were developed to take advantage of current processor architectures. These issues include:

- Simplify operators with sparse matrices where each column has only one element
- Perform a matrix-vector dot product, when the matrix is represented with the CSC format (column-oriented)
- Explore the specificities of a bitmap matrix/vector
- Unavailability of Khatri-Rao product implementations for CSC sparse matrices

When each column has just one element, we can assume that each record is represent by column, and one record only points to one row (label), reducing broadly the complexity of operators, reaching a better performance. However, note that this assumption cannot be true for all cases, specifically when a query has a sub-query, so that must be take into account.

The matrix-vector dot product using the format CSC presents relevant performance issues. The matrix-vector multiplication multiplies each row of the matrix by the column vector. Since this operation work with rows and the CSC format is oriented to columns, it is clear that this operation has a negative performance in a query execution. To overcome this, two operations can be used to perform the same result. Initially a Khatri-Rao product is used to introduce the vector values on the matrix. Next, a Matrix Bang operation with the sum aggregate function, performing the smash of the matrix horizontally. With this solution, a better performance was achieve in comparison to the matrix-vector multiplication.

Both matrices and vectors could be bitmap or measured. The difference between them is that the first one only represents the existence of attributes, and the latter contains data such as quantities, prices and other type of quantitative data. So, since in bitmap matrices/vectors the data values are always ones, it is possible to reduce significantly the memory accesses, which improves significantly the performance of these operators.

Since Khatri-Rao is a specific case of Kroencker product, it was even harder to find an implementation of it using the CSC format. However, since it is an easy product to understand, it was implemented efficiently.

V. Validation & Comparative Evaluation

A. Testbed

1) The hardware system:

All our tests and execution times measurements were done in the HPC cluster SeARCH [9].

Only one node was used throughout the whole study, and its fully characterisation is presented in Table V.

TABLE V: 652 SeARCH node

Processor	
Manufacturer	Intel® Corporation
Model	Xeon® E5-2670v2
Architecture	Products formerly Ivy Bridge EP
Processor Base Frequency	2.50 GHz (up to 3.30 GHz)
#CPUs	2
#Cores (per CPU)	10
#SMT ways (per CPU)	2
Lithography	22 nm
ISA Extensions	SSE4.2 & AVX
L1 Cache	10x 32KB (code) + 10x 32KB (data)
L2 Cache	10x 256KB
L3 Cache	25MB shared
Associativity	8-way / 8-way / 20-way
#Memory Channels	4
Main Memory	
RAM Memory	64GB
Peak Memory Bandwidth	59.7 GB/s

2) Compilation details:

All compiling is done using:

- Intel C Compiler (ICC) version 17.0.3 (GCC version 4.9.0 compatibility)
  - Optimisation level: 3 (`-O3`)
  - Other flags: `-g -Wall -Wextra -std=c99`
  - Vectorisation report flags: `-qopt-report=5 -qopt-report-phase=vec`
- GLib version 2.0
- OpenMP

3) Tuning PostgreSQL engine:

To compare the LA approach with a traditional SQL server PostgreSQL was chosen for that comparison.

PostgreSQL has full and complex development and releases done by an open source community over the years. The installed version is one of the latest releases (V. 9.6.2), which supports parallel queries essential to test a parallel approach. The PostgreSQL performance is considered one of the best among the open-source SQL database engines.

Many tweaks and tuning were done to maximise its performance to be a stronger comparative point, by following the official PostgreSQL tuning documentation [10], where some parameters are worth to mention:

- `shared_buffers`: sets the amount of memory the database server uses for shared memory buffers; The suggested value is 25% of the memory of the system, so it was installed for 16 GB.
- `effective_cache_size`: sets the amount of memory that the PostgreSQL query planner can use to build a efficient plan to execute a query; The suggested setting for this parameter is 75% of the main memory, which corresponds to 48 GB.
- `work_mem`: specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files; in this parameters, we set to 25 MB, the size of the L3 cache of the system.

PostgreSQL has been compiled and installed on the previously referenced machine, having also been configured with the parameters explained above.

4) The dataset:

To query using both LA and RA approaches, a common dataset was built with a third-party script [11] that generates data according to TPC-H specifications, to populate the PostgreSQL database, and will also be used by LA-OLAP.

We generated datasets with a scale factor from 1 to 128 GB, approximately. However, since we just work with queries that fit in main memory, only the datasets 1 to 32 GB were used.

B. Experimental details

To be fair with all the measurement process, certain rules must be defined and specified:

- Exclusive access to the cluster node during execution time measurements;
- Fair comparison, namely to guarantee that both engines have the whole dataset in RAM before measuring execution times;
- Perform at least 10 runs for each dataset and use the average of the best three execution times within a tolerance of 5%

VI. Results and Analysis

The LA-OLAP engine was tested to check if the outcomes are the same as the ones provided by a RA-OLAP engine. Then performance tests were performed and LA results were compared with the obtained from PostgreSQL, with several dataset sizes from 1 to 32 GB with and without parallelism. All operators in the LA parallel version were parallelized, except the bang operator. Besides the total execution time, the running time of each individual query step were analysed to spot some crucial performance hot-spots.

The first focus of this work was in query 3, because of its higher complexity and diversity of operations. Query 3 was used as the key example in Section III to explain the execution of the DSL operation in a query.

A. Validation

To guarantee DSL correctness of the results, a validation process to each query was performed: queries outcomes were compared with PostgreSQL. This validation was made for query 1 (simplified version), 3, 4, 6 and 10, from the set of 22 queries in the TPC-H benchmark suite.

B. Query execution time evaluation

Figure 3, plots execution times for sequential and parallel implementation of both LA-OLAP and PostgreSQL, using all cores of the 652 cluster node.

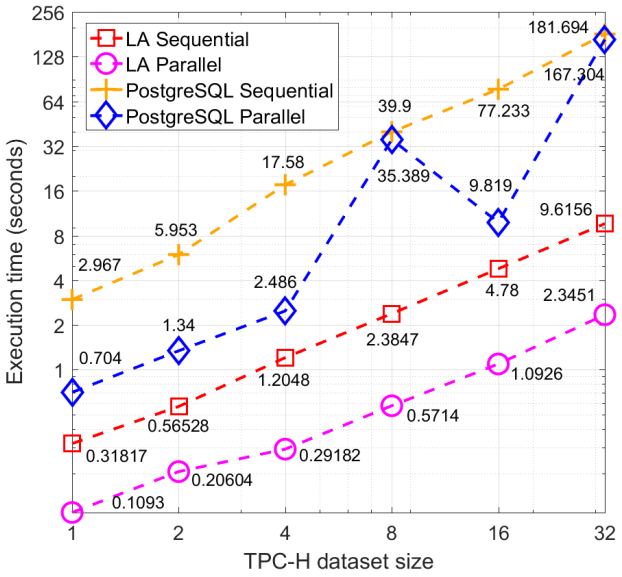


Fig. 3: Query 3 Execution time for several TPC-H scale factors, for both LA and RA with and without parallelism

First of all, for the PostgreSQL sequential times is noticeable that the execution time *grows* more than the TPC-H dataset size, i.e., the TPC-H scale factor increases the double for each step, and the execution time tends to increase more than the double.

The parallel version of PostgreSQL behaves better than the sequential version. However, for the 8 and 32 GB dataset, it seems that it was not executed in parallel. That is absolutely right. The DBMS[1], to execute a query, calculates a *query plan* through heuristic calculations, and then, some plans may not

[1]Database Management System

include the execution in parallel. That was what PostgreSQL did on those datasets. But, for the other values, still an efficient parallel execution.

For the LA results, is possible to see that for both sequential and parallel versions, the execution time is faster. An astonishing result, for example, LA sequential version to dataset size of 32 GB (9.6 seconds) beats the parallel PostgreSQL to just 4 GB (17.6 seconds) that have almost the double of the execution time. Apart from this efficient sequential version, LA parallel version has increased LA sequential more than 4 times, except for 2 GB dataset.

For 32 GB dataset, PostgreSQL sequential executes query 3 in 181.7 seconds and LA in just 9.6 seconds, 18 times faster. The LA parallel version has an execution time of 2.35 seconds against 167.3 seconds, 71 times faster. Notice that for this dataset size, the PostgreSQL version is not so much efficient, as mentioned. The LA parallel version has improved, reducing the execution time to less than 25% time of the LA sequential version.

In query 6, PostgreSQL always executed an optimised query plan and the parallel version grows linearly. Both LA versions are faster than any PostgreSQL version as well. To have a clue, for the 32 GB dataset, in sequential mode, PostgreSQL takes 207.9 seconds to execute query 6, and LA version takes only 6 seconds, that is more than 34 times faster. For parallel version, LA increases execution time to more than 7 times less. The query 6 execution times are in Appendix A.

C. Query profiling

To realize how much percentage every step/operation takes in the overall execution time and have a greater perception of the impact of each operation, query 3 and 6 are executed for a 32 gigabyte dataset in both sequential and parallel versions, doing these steps measurements.

First, the profiling of query 3 is observable in Figure 4, the steps that take more time in the sequential version are the 3, 5, 10 and 11. These steps correspond to a filter, a Khatri-Rao product, a map and an Hadamard product, respectively.

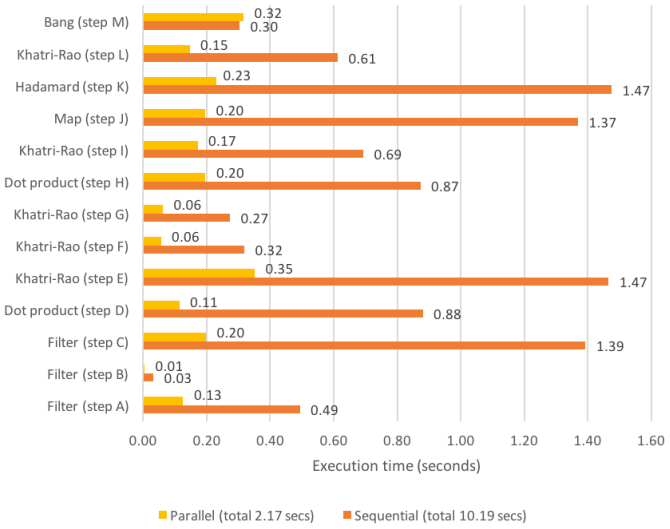


Fig. 4: Query 3 profile, step-by-step (dataset 32 GB)

Query 3 parallel version, as noticed in query execution time, has increased successfully the efficiency of LA solution. The parallel most time-consuming step takes less time than 70% of every step in the parallel version.

The same is concluded from query 6, where the steps that took most of the time were two filters and two Hadamard products (steps A, B, D and F respectively). Note that, except for bang operator, all of each steps of parallel version is lighter than any other step in sequential. Together, all of the parallel version take less time than just the step F (Hadamard product) of the sequential version. This information is in Appendix A.

VII. Conclusion

The concern about handling big data efficiently, lead us to challenge Relational Algebra solution with a Typed Linear Algebra approach.

At the current point of our work, LA operators were completely implemented and tested, validation was made for query 1 (simplified version), 3, 4, 6 and 10. Queries 3 and 6 were also completely tested in a HPC cluster using the TPC-H Benchmarks suite and verified against PostgreSQL database. Query 10 was tested but due to integer/float overflow, because of multiple and consecutive Khatri-Rao products, we needed to re-write all the source code to implement multi-precision variable types.

A broad study and profiling across the queries and general databases requirements were made during all the development to identify, explore and resolve bottlenecks over the various stages of our implementation. Sequential and parallel operators were optimised several times. Low-level operators were designed to enhance this solution, since depending on each operation characteristics, a different operation optimisation could be done. Consequently, a more suitable operation is applied to achieve better performance results and keep the right query results.

Over this report, the LA approach was explained as well as main decisions to design and implement this solution successfully and efficiently. At this time, a lot of further work can be made through the results accomplished and we took a step further towards a real competitive approach to RA database solutions.

VIII. Future work

Due to lack of time and the difficulty of the project, only a few aspects were explored such as the development of a linear algebra class of operators and several low-level operators, new data labels representation, and new queries implementation. It is now necessary to point out some ideas to further exploration.

The first one is the implementation of a class of linear algebra database engine, that uses the current LA operators to execute the different operations. This simple engine could be a good starting point to implement TPC-H queries faster, and therefore, get their profile and performance information much more easily and quicker. This also helps to find out new nuances of linear algebra applied to the LA approach.

A DSL parser needs to be implemented to parse and build an intermediate structure like a syntax abstract tree, and a query

executor that traverse the intermediate structure and processes the query, given the result table.

One interesting point of that intermediate structure is that it can allow both parallel processing of independent operations and parallel processing of operations by load balancing them.

Another interesting thing is to explore other sparse formats to represent our matrices, rather the current one. We believe that we can find alternatives that explore and improve data locality, like the *Block Compressed Sparse Column/Row* (BCSC/BCSR). However, this format would largely increase algorithm complexity for the defined methods, and could give us irrelevant performance gains. We should never forget that there is a trade-off between how much a sparse format can save us memory space, and how easily we can re-adapt our algorithms to use it.

Besides the ideas presented previously, other concerns can be taken. They are ordered by importance and priority:

- Add the multi-precision library GMP to the implementation, test it, and measure the overhead introduced by it to know how much it affects overall performance. Key to implement query 10
- Study how to represent measure attributes, since in TPC-H query 10, a measure attribute needs to be represented in both bitmap matrix and measure vector
- Study the representation of composite primary keys. Currently we represent each key separately, but in relational databases they only have meaning together
- Develop a generic operator to build the result table of queries
- Develop a generic operator to calculate algebraic operations that could be processed element to element
- Fully translate the remaining TPC-H queries and benchmark both linear and relational algebra approaches
- Adapt current DSL definition to support all TPC-H queries
- Investigate how to use CSC format to implement mechanisms to partition data, introducing our approach to a distributed memory way to execute queries
- Experiment and test with *many-core* hardware to see how efficient our approach is in these powerful devices

IX. Acknowledgements

We would like to thank our advisors Alberto Proença and José Nuno Oliveira for all the help they gave us in their expertise areas, High Performance Computing and Formal Methods, respectively. We would also like to thank Filipe Costa for all the previous work he developed in this project, and for helping us to better understand the challenges and next steps of this Linear Algebra approach to OLAP.

References

[1] E.F. Codd, S.B. Codd, and C.T. Salley. Providing olap to user-analysts: An it mandate. *Codd & Associates*, 1993.
[2] Filipe Costa and Sérgio Caldas. Optimisation of a linear algebra approach to olap. 2016.
[3] Rogério Pontes. Benchmarking a linear algebra approach to olap. 2015.
[4] João Afonso and João Fernandes. Towards an efficient linear algebra representation for olap. 2017.
[5] TPC. TPC-H. http://www.tpc.org/tpch/.
[6] Eugene Loh. The ideal hpc programming language. *ACM*, 2010.
[7] Colorado State University. Sparse Matrix Compression Formats. http://www.cs.colostate.edu/~mcrob/toolbox/c++/sparseMatrix/sparse_matrix_compression.html.
[8] GNOME. GLib Reference Manual - Hash Tables. https://developer.gnome.org/glib/2.28/glib-Hash-Tables.html.
[9] Universidade do Minho. Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). http://search6.di.uminho.pt/wordpress/.
[10] PostgreSQL. Tuning Your PostgreSQL Server. https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server.
[11] David Phillips. TPC-H DBGEN. https://github.com/electrum/tpch-dbgen.

APPENDIX A
QUERY 6

A. SQL

```
SELECT
        sum(l_extendedprice * l_discount) as revenue
FROM
        lineitem
WHERE
        l_shipdate >= date '1995-03-10'
        and l_shipdate < date '1995-03-10' + interval '1' year
        and l_discount between 0.05 - 0.01 and 0.05 + 0.01
        and l_quantity < 3;
```

B. DSL

```
A = filter_to_vector( l_shipdate, >='1995-03-10', <'1995-03-10')
B = filter_to_vector( l_discount, >='0.49', <='0.51' )
C = filter_to_vector( l_quantity, <='3' )
D = hadamard( A , B )
E = hadamard( C , D )
F = hadamard( l_extendedprice , l_discount )
G = hadamard( E , F )
H = bang( sum, G )
```

C. Sequential & Parallel Results and Profiling

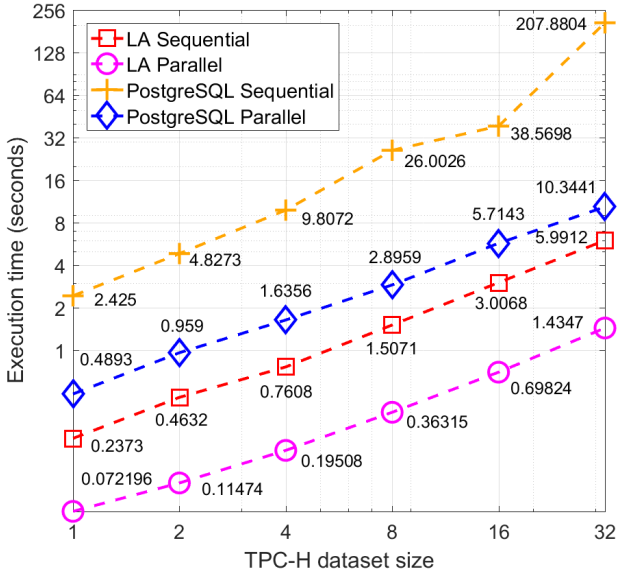


Fig. 5: Query 6 Execution time for several TPC-H scale factors, for both LA and RA with and without parallelism

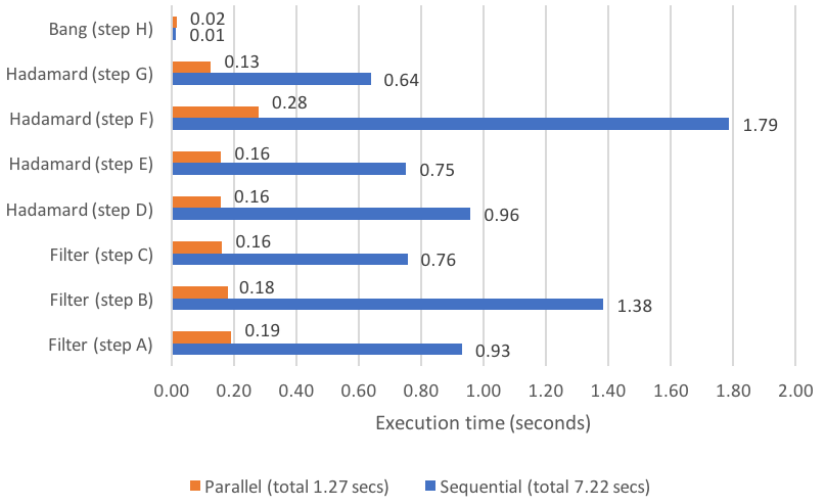


Fig. 6: Query 6 profile, step-by-step (dataset 32 GB)