Optimisation of a Linear Algebra Approach to OLAP

Filipe Oliveira - A57816
a57816@alunos.uminho.pt

Sérgio Caldas - A57779
a57779@alunos.uminho.pt

September 13, 2016

Abstract

Online Analytical Processing (OLAP) systems, perform multidimensional analysis of business data and provides the capability for complex calculations, trend analysis, and sophisticated data modelling. All the referred analysis depends on Relational Algebra, which lack algebraic properties, and qualitative and quantitative proofs for all the relational operator. The proposed solution focus on a typed linear algebra approach, encoding OLAP functionality solely in terms of Linear Algebra operations (matrices).

It has been argued that linear algebra (LA) is better suited than standard relational algebra for formalising and implementing queries in on-line multidimensional data analysis [3] [1]. This can be achieved over a small LA sparse matrix kernel which, further to multiplication and transposition, offers the Kronecker, Khatri-Rao and Hadamard products. We give preliminary experimental results obtained with one cluster of Search6 using queries of the TPC-H Benchmark.

I. Introduction

The design and development of systems that generate, collect, store, process, analyse, and query large sets of data is filled with significant challenges both hardware and software. Combined, these challenges represent a difficult landscape for software engineers.

The relational database is the current solution for big data storage. Data dependencies in databases can be seen as a binary two-way associative relation. Prior efforts have been made [3] [1] towards a Linear Algebra (LA) approach to OLAP, in order to fully represent relational algebra in terms of linear algebra operators.

OLAP is resource-demanding and calls for efficient parallel approaches. We implemented from the start a typed linear algebra solution given special importance to the data modelling since it may limit the attainable efficiency in data-intensive systems such as OLAP, a time consuming task that accesses massive amounts.

To validate the LA approach in a real case scenario and evaluate the outcome performance,the full set of TPC-H benchmarks was selected, with multiple query runs, and so far only one is reported in this work. To obtain realistic and meaningful results large datasets were considered, ranging from 1 to 32GB.

For a comparative evaluation between the LA and the objected-relational database management system we selected PostgreSQL version 9.6, with roots in open source community, to represent the relational algebra approach.

Given this proximity between database relations and linear algebra, the question arises: does the linear algebra approach presents performance improvements when compared with the relational one?

The following sections introduce the LA strategy (section II), describe the experimental work with the sequential version of the code (section III), introduce the parallelisation techniques to improve performance with experimental execution times IV, and the last section concludes with suggestions for later work.

II. Linear Algebra Strategy

I. Towards a linear algebra semantics for SQL

Inspired by point-free relational data processing, an alternative roadmap for parallel online analytical processing (OLAP) can be achieved based on encoding data in matrix format and relying thereupon solely on LA operations[2].

I.1 Encoding data in matrix format

As example of raw data consider the displayed table 1 where each row records the order key, quantity, return flag, line status and ship date from an given TPC-H benchmark lineitem table. In order to facilitate data association the column number in the corresponding lineitem table was included above each data column.

Table 1: *Collection of raw data (adapted from TPC-H benchmark lineitem table).*

#1 l_orderkey	#5 l_quantity	#9 l_returnflag	#10 l_linestatus	#11 l_shipdate
1	17	N	O	1996-03-13
1	36	N	O	1996-04-12
1	8	N	O	1996-01-29
1	28	N	O	1996-04-21
1	24	N	O	1996-03-30
1	32	N	O	1996-01-30
2	38	N	O	1997-01-28
3	45	R	F	1994-02-02
3	49	R	F	1993-11-09
3	27	A	F	1994-01-16

To obtain useful information from raw data, which in OLAP systems escalated to Terabytes of information, we need to summarise the data by selecting attributes of interest and exhibiting their inter-relationships.

Consider the following simplified TPC-H query 1: "How many items were sold per return flag and line status?". For this particular question, the necessary attributes to answer the query are present on table lineitem, being **return flag**, **line status**, and **quantity**. In relational algebra that question could be easily expressed by the following SQL code, which would produce the result presented on table 2.

Listing 1: *SQL code for the simplified TPC-H query 1*

```
SELECT l_returnflag, l_linestatus,  sum(l_quantity)
FROM LINEITEM_SAMPLE
GROUP BY l_returnflag, l_linestatus;
```

Table 2: *Simplified query-1 relational algebra result from the collection of raw data (adapted from TPC-H benchmark lineitem table).*

#9 l_returnflag	#10 l_linestatus	#5 l_quantity
N	O	183
R	F	94
A	F	27

Aggregations like the ones presented on this query occur in all TPC-H queries, hence performance of group-by and aggregation deserves a special attention. This issue will be addressed in later sections of the report.

To express OLAP queries in therms of LA, the key lies in expressing operations in the form of matrix algebra expressions. This particular example, requires three matrices, one for each attribute, with each matrix being correlated to relational algebra as the row storage of columns #5, #9, and #10.

However, to do so, we need to find a two-way association between the presented string on the rows #9 and #10 and an unique integer identifier.

Our proposed solution encodes strings recurring to Gnome **Gquarks** - a two-way association between a non-zero unsigned int and a char * - based on a thread safe hashtable.

By order of appearance, each unique string will be associated to an unique unsigned integer. Repeated strings will be associated to the prior corresponding unsigned integer. The resulting unsigned integer value range will start in the number 1.

The value 0 in GQuarks is associated to NULL. Since in the proposed solution, row and column numbers will respect C-style arrays notation, both column and row numbering will start at 0.

The GQuark unsigned integer value decremented by 1 will represent the row position on the matrix, and the register number, starting at 0, will represent the column position on the matrix.

We can now verify that at most one non-zero cell can be found in each column of the matrix, to maintain the two-way association between a register and its corresponding string (the matrix is "functional"). There is also the possibility of direct associating an register number with a value, wether integer or floating point. The referred matrices will be diagonal matrices, in which the element value is directly associated with the register.

Two types of matrices have now been presented:

1. Projection matrices - which recur to Gnome Gquarks, in which the Gquark decremented by 1, represents the row position of the matrix, and the register number, starting at 0, represents the column position on the matrix.

2. Measure matrices - direct associating a register number with a value, wether integer or floating point, where the element value is directly associated with the register number, and consequently the column and row position.

We can now build the necessary matrices for the given examples, as shown in figures 1 to 3. The two-way association between char* and unsigned integer is also presented in table 3, to aid the example comprehension.

Table 3: *Two-Way association between GQuarks and Row Number, for rows #9 and #10 presented in the collection of raw data (adapted from TPC-H benchmark lineitem table).*

String	GQuark	Row #
N	1	0
R	2	1
A	3	2
O	4	3
F	5	4

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
r0	17	0	0	0	0	0	0	0	0	0
r1	0	36	0	0	0	0	0	0	0	0
r2	0	0	8	0	0	0	0	0	0	0
r3	0	0	0	28	0	0	0	0	0	0
r4	0	0	0	0	24	0	0	0	0	0
r5	0	0	0	0	0	32	0	0	0	0
r6	0	0	0	0	0	0	38	0	0	0
r7	0	0	0	0	0	0	0	45	0	0
r8	0	0	0	0	0	0	0	0	49	0
r9	0	0	0	0	0	0	0	0	0	27

Figure 1: *Dense measure matrix produced from the collection of raw data (adapted from TPC-H benchmark lineitem table), from column #5 (quantity column).*

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
r0	1	1	1	1	1	1	1	0	0	0
r1	0	0	0	0	0	0	0	1	1	0
r2	0	0	0	0	0	0	0	0	0	1

Figure 2: *Dense projection matrix produced from the collection of raw data (adapted from TPC-H benchmark lineitem table), from column #9 (return flag column) with 2-way association achieved recurring to GQuarks.*

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
r0	0	0	0	0	0	0	0	0	0	0
r1	0	0	0	0	0	0	0	0	0	0
r2	0	0	0	0	0	0	0	0	0	0
r3	1	1	1	1	1	1	1	0	0	0
r4	0	0	0	0	0	0	0	1	1	1

Figure 3: *Dense projection matrix produced from the collection of raw data (adapted from TPC-H benchmark lineitem table), from column #10 (line status column) with 2-way association achieved recurring to GQuarks.*

II. Projection Operation

The projection statement of the example query is in listing 2.

Listing 2: *SQL code for the simplified TPC-H query 1 only for the SELECT and GROUP BY statements*

```
SELECT l_returnflag, l_linestatus
FROM LINEITEM_SAMPLE
GROUP BY l_returnflag, l_linestatus;
```

The typical case to reproduce a SELECT and GROUP BY statement on a relational algebra approach would be to remove certain columns of the lineitem table. In the LA approach the process consists of joining the projection matrices of the selected attributes. The final LA result should hold all possible combinations of the projected attributes and no duplicated tuples. That can be achieved by doing several Khatri-Rao products, one for each tuple of attributes present in the statement.

The corresponding LA encoding of the SQL statement from listing 2 is given by the equation 1, which would produce the matrix presented in figure 4.

$$Projection\ Matrix\ =\ ReturnFlag\ \otimes LineStatus \tag{1}$$

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
r0	0	0	0	0	0	0	0	0	0	0
r1	0	0	0	0	0	0	0	0	0	0
r2	0	0	0	0	0	0	0	0	0	0
r3	1	1	1	1	1	1	1	0	0	0
r4	0	0	0	0	0	0	0	0	0	0
r5	0	0	0	0	0	0	0	0	0	0
r6	0	0	0	0	0	0	0	0	0	0
r7	0	0	0	0	0	0	0	0	0	0
r8	0	0	0	0	0	0	0	0	0	0
r9	0	0	0	0	0	0	0	1	1	0
r10	0	0	0	0	0	0	0	0	0	0
r11	0	0	0	0	0	0	0	0	0	0
r12	0	0	0	0	0	0	0	0	0	0
r13	0	0	0	0	0	0	0	0	0	0
r14	0	0	0	0	0	0	0	0	0	1

Figure 4: *Projection matrix produced from the Khatri-Rao product between return flag and line status columns from lineitem table, from columns #9 and #10.*

Given the matrices A with dimensions $(m \times n)$ and B with dimensions $(o \times p)$, the resulting projection matrix has dimensions $(q \times r)$, being $q = m = o$ and $r = m \times o$. The resulting matrix dimensions can also be expressed as $(\ (\ m \times o\ ) \times n)$ as shown in figure 4.

To produce the desired tuples holding all possible combinations, the association between row number and tuple has to be created. Given two original matrices A with dimensions $(m \times n)$ and B with dimensions $(o \times p)$, and one produced projection matrix C via the Khatri-Rao product, to re-obtain both strings that produce each tuple, for every row R_C from the C matrix, the corresponding R_A from matrix A, and R_B from matrix B, are given by the expression:

$$\begin{aligned} R_A &= R_C / o \\ R_B &= R_C \,\%\, o \end{aligned} \tag{2}$$

We can now build the corresponding association between tuples and the rows of matrix C, as shown in table 4.

Table 4: *Association between the produced projection matrix from the Khatri-Rao product between return flag and line status columns from lineitem table, from columns #9 and #10, and the corresponding produced tuples for every row of matrix C.*

Row C	Column C	R_A	R_B	String A	String B	Tuple
r 3	0	0	3	N	O	(N , O)
r 3	1	0	3	N	O	(N , O)
r 3	2	0	3	N	O	(N , O)
r 3	3	0	3	N	O	(N , O)
r 3	4	0	3	N	O	(N , O)
r 3	5	0	3	N	O	(N , O)
r 3	6	0	3	N	O	(N , O)
r 9	7	1	4	R	F	(R , F)
r 9	8	1	4	R	F	(R , F)
r 14	9	2	4	A	F	(A , F)

III. Aggregation Operation

As you can state by observing table 4 the SELECT SQL statement was reproduced, however, the GROUP BY operation still needs to be applied.

To represent the GROUP BY operation we need to define an operator, a row vector commonly know as bang[3].

Given a matrix C with dimensions $((m \times o) \times n)$, a correctly typed bang vector would have dimension n and be totally filled with the value 1. Figure 5 illustrates the bang vector to be multiplied by matrix C, to group the elements located in the same row of matrix C.

$$\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \end{array} \begin{array}{c} c0 \\ \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \end{array}$$

Figure 5: *Projection matrix produced from the Khatri-Rao product between return flag and line status columns from lineitem table, from columns #9 and #10.*

The grouped LA encoding of the SQL statement from listing 2 is given by the equation 3, resulting in the projection vector presented in figure 6:

$$Projection\ Vector\ =\ (\ ReturnFlag\ \otimes LineStatus\ ) \times bang \quad (3)$$

$$\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \\ r10 \\ r11 \\ r12 \\ r13 \\ r14 \end{array} \begin{array}{c} c0 \\ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \end{array}$$

Figure 6: *Projection vector produced from the product between Khatri-Rao product shown on figure 4 and bang row vector show on figure 6.*

Considering the value 0 as the non existing relation between atributes, we can simplify the given projection vector and produce the association between return flag and line status columns from lineitem table, based on expression 2, as shown in table 5.

Table 5: *Association between the produced projection vector from the Khatri-Rao product between return flag and line status columns from lineitem table, from columns #9 and #10, and the corresponding tuples.*

Row C #	R_A	R_B	String A	String B	Tuple
r 3	0	3	N	O	(N , O)
r 9	1	4	R	F	(R , F)
r 14	2	4	A	F	(A , F)

To produce the results show on listing 1, using the linear algebra approach, we need to associate to every distinct tuple the corresponding quantity. Expression 4 adds the quantity measure matrix to the LA encoding. All measure matrices, shall therefore be represented between double square brackets, to simplify differentiation from the projection matrix type.

$$((ReturnFlag\ \otimes LineStatus\ ) \times\ [\![Quantity]\!]) \times bang \quad (4)$$

In addition to the association between row number and tuple, with the values 1 and 0 representing the existence or non-existence of the attributes association respectively, we now have a value associated to each tuple. Figure 7 illustrates the resultant matrix of the dot product between the Khatri-Rao product between return flag and line status columns, and the quantity measure matrix, from the given lineitem example table.

$$
\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \\ r10 \\ r11 \\ r12 \\ r13 \\ r14 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} c0 & c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 & c9 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
17 & 36 & 8 & 28 & 24 & 32 & 38 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 45 & 49 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 27
\end{pmatrix}
\end{array}
$$

Figure 7: *Resultant matrix of the dot product between the Khatri-Rao product between return flag and line status columns, and the quantity measure matrix, produced from the LA expression 4.*

The dot product between the resultant matrix defined in figure 7 and the row bang vector defined in figure 6 would produce the final row vector represented in figure 8.

$$
\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \\ r10 \\ r11 \\ r12 \\ r13 \\ r14 \end{array}
\begin{array}{c}
\begin{array}{c} c0 \end{array} \\
\begin{pmatrix}
0 \\ 0 \\ 0 \\ 183 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 94 \\ 0 \\ 0 \\ 0 \\ 0 \\ 27
\end{pmatrix}
\end{array}
$$

Figure 8: *Final row vector produced from the product between Khatri-Rao product shown on figure 7 and bang row vector show on 6.*

We have now defined the necessary operations to reproduce through linear algebra the relational algebra results shown in table 2. Table 6 associates the generated tuples via the Khatri-Rao operation shown in figure 7 and the row vector bang as shown in expression 4.

Table 6: *Association between the generated tuples via the Khatri-Rao operation shown on figure 7 and the row vector bang as shown on expression 4.*

Row C #	R_A	R_B	String A	String B	Tuple	Value
r 3	0	3	N	O	(N , O)	183
r 9	1	4	R	F	(R , F)	94
r 14	2	4	A	F	(A , F)	27

IV. Selection Operation

Given the obtained results, there is still one linear operation required to fully answer to a new query, closer to the TPC-H query 1: "How many items were sold per return flag and line status, between the dates 1996-04-12 and 1997-01-28?". For this particular question, the necessary attributes to answer the query remain present on the table lineitem, being the prior analysed **return flag**, **line status**, and **quantity**, with the addition of the selection attribute – **shipdate**. In relational algebra that question could be easily expressed with the extension of the prior SQL code presented on listing 1, which would produce the result presented on table 7:

Listing 3: *SQL code for the simplified TPC-H query 1*

```
SELECT l_returnflag, l_linestatus,  sum(l_quantity)
FROM LINEITEM_SAMPLE
WHERE l_shipdate >= "1996-04-12" AND l_shipdate <= "1997-01-28"
GROUP BY l_returnflag, l_linestatus;
```

Table 7: *Query-1 relational algebra result from the collection of raw data (adapted from TPC-H benchmark lineitem table).*

#9 l_returnflag	#10 l_linestatus	#5 l_quantity
N	O	102

The same result could be computed on linear algebra operations by strictly restricting the values present on the matrix shown in figure 7.

By producing a measure selection matrix initially with its diagonal cells filled with value 1, and by re-obtaining the string based on the corresponding GQuark of the selection column, comparing it with both keys (**1996-04-12** and **1997-01-28**), we can simply replace with value 0 all cells of the measure selection matrix that do not pass in the restriction.

On subsection V the non validation of the comparison would result in the elimination of the element from the sparse representation, as we shall address later.

To simplify the visualisation process of the algorithm, this method will be used in this section.

Expression 5 adds the selection measure matrix to the LA encoding. We now have projection, selection, and aggregation operations defined only in terms of linear algebra.

$$(((\ ReturnFlag \otimes LineStatus\ ) \times [\![Selection]\!]\ ) \times [\![Quantity]\!]) \times bang \tag{5}$$

Figure 9 illustrates the resultant measure selection matrix of the restriction of shipdate column, from the given lineitem example table. To assist its visualisation, table 8 associates the

row value of shipdate for every register with the result of the selection process.

Table 8: *Association between row value of shipdate for every register with the result of the selection process, for a restriction of values between **1996-04-12** and **1997-01-28**.*

Row #	l_shipdate	Value
r 0	1996-03-13	FALSE
r 1	1996-04-12	**TRUE**
r 2	1996-01-29	FALSE
r 3	1996-04-21	**TRUE**
r 4	1996-03-30	FALSE
r 5	1996-01-30	FALSE
r 6	1997-01-28	**TRUE**
r 7	1994-02-02	FALSE
r 8	1993-11-09	FALSE
r 9	1994-01-16	FALSE

$$\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \end{array} \begin{array}{c} \begin{array}{cccccccccc} c0 & c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 & c9 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 9: *Measure selection matrix produced from the restriction of shipdate column to values between **1996-04-12** and **1997-01-28**, from the given lineitem example table, from column #11.*

Note that the selection process could occur both in the projection or the aggregation process with the restriction to be realised before the bang operation.

To otimize the operations, the selection should be performed in the initial stages of the query, since it reduces the amount of computation required later.

To visualize the LA operations presented in expression 5, the product between Khatri-Rao product was shown in figure 7, the measure selection matrix was defined in figure 9, the measure quantity matrix was defined in figure 1, and the row bang vector was defined in figure 6, these led to the final row vector presented on figure 10.

$$\begin{array}{c} \\ r0 \\ r1 \\ r2 \\ r3 \\ r4 \\ r5 \\ r6 \\ r7 \\ r8 \\ r9 \\ r10 \\ r11 \\ r12 \\ r13 \\ r14 \end{array} \begin{array}{c} \begin{array}{c} c0 \end{array} \\ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 102 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \end{array}$$

Figure 10: *Final row vector produced from the product between Khatri-Rao product shown on figure 7, the measure selection matrix define in figure 9, the measure quantity matrix define in figure 1, and the row bang vector defined in figure 6.*

We have now defined the necessary operations to reproduce through linear algebra the relational algebra results shown on table 7. Table 9 associates the generated selected tuples via expression 5.

Table 9: *Association between the generated selected tuples via expression 5.*

Row C #	R_A	R_B	String A	String B	Tuple	Value
r 3	0	3	N	O	(N , O)	102

V. Encoding data in Sparse Matrix Format

The matrices used to encode OLAP information displau a high degree of both sparsity and irregularity. Regarding the minor dataset used in the experimental work, figures 11 trough 14, analyse the sparsity pattern of the necessary columns, to answer the simplified TPC-H query-1 show on expression 5.

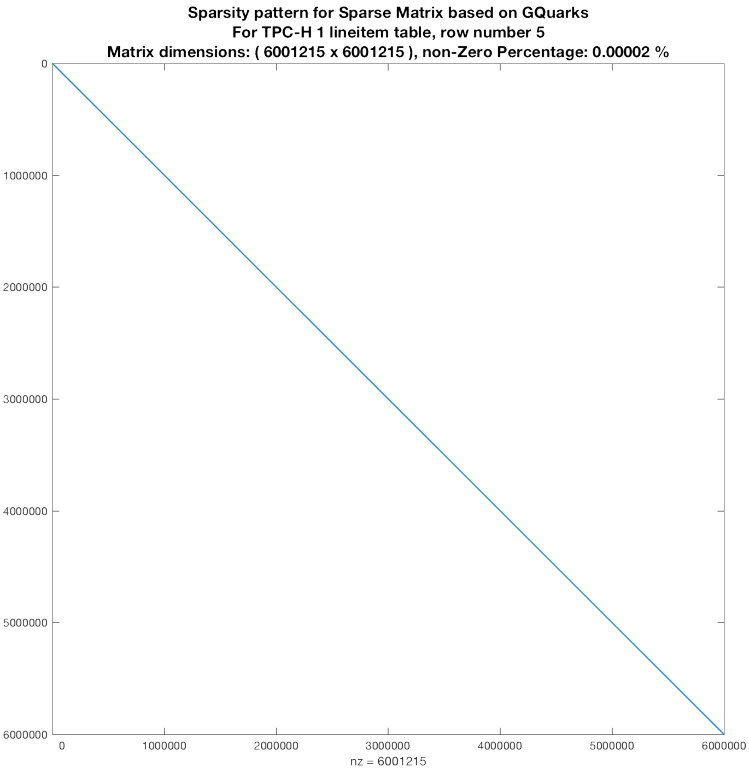

Figure 11: *Sparsity pattern analysis for attribute quantity, from the TPC-H dataset 1GB lineitem table(column #5).*

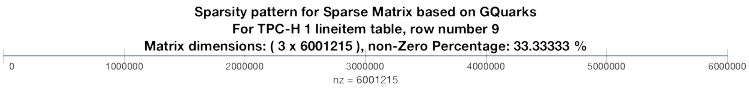

Figure 12: *Sparsity pattern analysis for attribute return flag, from the TPC-H dataset 1GB lineitem table(column #9).*

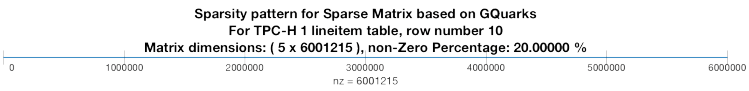

Figure 13: *Sparsity pattern analysis for attribute line status, from the TPC-H dataset 1GB lineitem table(column #10).*

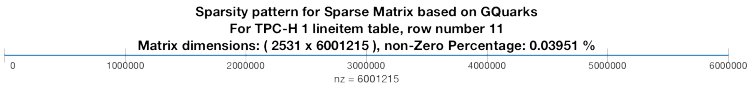

Figure 14: *Sparsity pattern analysis for attribute shipdate, from the TPC-H dataset 1GB lineitem table(column #11).*

As shown in the prior sparsity analysis, a matrix dimensions $(m \times n)$ will have, in the worst case, a non-zero number equal to n, since each column has at most one element. Therefore, to fully realise the potential of the processing units, this memory bottleneck should be overcome.

Given the example matrix of figure 11, the number of zero elements that would have to be stored and processed would be $(n-1)^2$.

Taking into account that for the 32GB TPC-H dataset the number o element arises to 192,000,551, the necessary amount of memory to store only one single precision dense matrix would be aproximately 150PB.

To explore data locality in cache memories to improve performance the design of the code should be consistent with the design of the caches. The used matrices must also be compressed to take advantage of their mathematical structure. With the prior statements in mind the storage formats chosen rely on the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats [4].

Since each column has at most one element, the CSC format construction is thereby simplified, giving place to an sequential column pointer array, bigger in size in one element when compared to the value and row index arrays. Denote that after the LA selection operation the column pointer array no longer keeps the illustrated property.

Regarding the usage of CSR format, it was chosen due to the creation of an alternative version to the LA operations that took CSC formatted matrices as input. This alternative version took CSR formatted matrices as input and relied on Intel® Math Kernel Library version 11.3.

VI. Incremental construction

Projection matrices, as defined on section I.1 are amenable to incremental construction due to the GQuarks lema, where each unique string will be associated to an unique unsigned integer. If the new data does not exist in terms of a GQuark, a new association will be created and a new unique unsigned integer will be associated to it. Regarding the consequent matrix column increase, due to register addition, no problem arises from that action.

Dimension matrices, also defined on section I.1, only depend on the matrix column increase due to the direct association of the cell value, having also no problem regarding incremental construction. Yesterday's data will still be valid and have zero migration cost with the addition of today's data.

III. Sequential Experimentation

Regarding the prior described lemas, we shall now assess wether if the linear algebra approach presents performance improvements when compared with the relational one. Before we discuss the measured performance results in the following section, we will briefly summarise characteristics of the multi-core platform in our test suite, and present an overview of the performed tunings.

Throughout all experiments, the same platform was used. The system, referenced as compute node 652-1, has two Intel® Xeon® E5-2670v2 (Ivy Bridge architecture) sharing 64 GB of DDR3 RAM, 1333 MHz, accessed through 4 memory channels. Table 10 fully characterises the hardware features of the test platform:

System	compute-652-1
# CPUs	2
CPU	Intel® Xeon® E5-2670v2
Architecture	Ivy Bridge
# Cores per CPU	10
# Threads per CPU	20
Clock Freq.	2.5 GHz
L1 Cache	320KB 32KB per core
L2 Cache	2560KB 256KB per core
L3 Cache	25600KB shared
Inst. Set Ext.	SSE4.2 & AVX
#Memory Channels	4
Vendors Announced Peak Memory BW	59.7 GB/s
Measured[1] Peak Memory BW	58.5GB/s

Table 10: *Architectural characteristics of the evaluation platform.*

The software used for both relational and linear algebra, and the corresponding versions are stated bellow:

- Linear Algebra:
  - Compiler: ICC version 16.0.0 (GCC version 4.4.6 compatibility)
    - no vectorization: -O3 -std=c99 -no-vec -farray-notation
    - vectorization: -O3 -std=c99 -farray-notation -xAVX -vec-report7
  - Intel® MKL Version 11.3
    - Link line: -lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm
- Relational Algebra (PostgreSQL version 9.6+);
  - Built with the following dependencies:
    - GCC version 4.9.0
    - Python 2.6.6

PostgreSQL was compiled specifically for the test platform, to fully take advantage of the available computing resources.

I. Tuning the relational algebra engine

Database, application, and storage servers ship with a large number of configuration parameters like buffer cache sizes, number of I/O daemons, and parameters input to the database query optimiser. Finding good settings for these parameters is a challenging task because of the complex ways in which parameter settings can affect performance. The parameters shared_buffers, effective_cache_size, and work_mem, were adjusted accordingly, with shared_buffers begin set to 2GB, effective_cache_size being set to 64GB, and work_mem being set to 25MB. In order to further speed up RA queries, indexes were created for all the database tables.

II. Tuning sparse CSC and CSR methods to assist data level parallelism

Efficient SSE vectorisation was achieved in both versions of the linear algebra approach. The usage of the performant Intel Math Kernel Library (MKL) fully assisted vectorisation of the dot product between sparse matrices and sparse matrix vector, on the CSR version.

The compiler was also instructed via auto vectorisation hints, and user mandated vectorisation. The non existence of vector dependencies, when verified, was also explicitly included.

Alignment of data and data structures can affect performance. In the memory allocation alignment all data was aligned according to cache line size. Regarding the access alignment, for AVX, alignment to 32-byte boundaries (8 SP chunks) allowed a single reference to a cache line to move 8-SP numbers into the registers. The compiler was instructed to to assume that all CSC and CSR arrays are aligned on an 32-byte boundary.

III. Tuning RA and LA approaches

We conducted experiments on the simplified TPC-H query-1, shown in listing 4, similar to the one presented in listing 3. The experiments focused a variety of datasets ranging from 1GB to 32GB. An overview of their characteristics appears in table 11.

The CSC format requires a larger overall space for data, but it allowed us to simplify several algorithms and improve the overall perfomance.

Listing 4: *SQL code for the simplified TPC-H query 1 used on the experimentation*

```
SELECT l_returnflag, l_linestatus,  sum(l_quantity)
FROM LINEITEM_1
WHERE l_shipdate >= "1998-08-28" AND l_shipdate <= "1998-12-01"
GROUP BY l_returnflag, l_linestatus;
```

Table 11: *Overview of the produced sparse matrices used in evaluation study.*

Dataset	Measure Matrix Quantity	Projection Matrix return flag	Projection Matrix line status	Projection Matrix ship date	Overall SP Space Required	
	Dimensions, Nonzeros, CSR space, CSC space	Dimensions, Nonzeros, CSR space, CSC space	Dimensions, Nonzeros, CSR space, CSC space	Dimensions, Nonzeros, CSR space, CSC space	CSR format	CSC format
1	(6001215 × 6001215)	(3 × 6001215)	(5 × 6001215)	(2531 × 6001215)		
	nnz: 6001215	nnz: 6001215	nnz: 6001215	nnz: 6001215		
	CSR: 69 MB CSC: 69 MB	CSR: 46 MB CSC: 69 MB	CSR: 46 MB CSC: 69 MB	CSR: 46 MB CSC: 69 MB	206 MB	275 MB
2	(11997996 × 11997996)	(3 × 11997996)	(5 × 11997996)	(2531 × 11997996)		
	nnz: 11997996	nnz: 11997996	nnz: 11997996	nnz: 11997996		
	CSR: 137 MB CSC: 137 MB	CSR: 92 MB CSC: 137 MB	CSR: 92 MB CSC: 137 MB	CSR: 92 MB CSC: 137 MB	412 MB	549 MB
4	(23996604 × 23996604)	(3 × 23996604)	(5 × 23996604)	(2531 × 23996604)		
	nnz: 23996604	nnz: 23996604	nnz: 23996604	nnz: 23996604		
	CSR: 275 MB CSC: 275 MB	CSR: 183 MB CSC: 275 MB	CSR: 183 MB CSC: 275 MB	CSR: 183 MB CSC: 275 MB	824 MB	1098 MB
8	(47989007 × 47989007)	(3 × 47989007)	(5 × 47989007)	(2531 × 47989007)		
	nnz: 47989007	nnz: 47989007	nnz: 47989007	nnz: 47989007		
	CSR: 549 MB CSC: 549 MB	CSR: 366 MB CSC: 549 MB	CSR: 366 MB CSC: 549 MB	CSR: 366 MB CSC: 549 MB	1648 MB	2197 MB
16	(95988640 × 95988640)	(3 × 95988640)	(5 × 95988640)	(2531 × 95988640)		
	nnz: 95988640	nnz: 95988640	nnz: 95988640	nnz: 95988640		
	CSR: 1099 MB CSC: 1099 MB	CSR: 732 MB CSC: 1099 MB	CSR: 732 MB CSC: 1099 MB	CSR: 732 MB CSC: 1099 MB	3296 MB	4394 MB
32	(192000551 × 192000551)	(3 × 192000551)	(5 × 192000551)	(2531 × 192000551)		
	nnz: 192000551	nnz: 192000551	nnz: 192000551	nnz: 192000551		
	CSR: 2197 MB CSC: 2197 MB	CSR: 1465 MB CSC: 2197 MB	CSR: 1465 MB CSC: 2197 MB	CSR: 1465 MB CSC: 2197 MB	6592 MB	8789 MB

IV. Experimental results analysis

Figure 15 plots the measured execution for a range of datasets, and both linear and relational algebra approaches.

Denote that the presented values were selected through the K-Best technique, with K=3, from 50 samples.

For the linear algebra approach we present the best solution from CSR and CSC format. The presented linear algebra solution in figure 15 is the one using CSR and format and Intel MKL version 11.3.

The experiment presented in the current section focus on the best possible sequential solutions for both relational and linear algebra versions. The parallel PostgreSQL line was only introduced to elucidate the parallel goal to achieve in the LA versions discussed in later sections of this report, and discuss future potential improvements of our linear algebra system.

As shown, the coding effort to tune sparse CSR methods to assist data level parallelism revealed itself fruitful. However, a further analysis should be produced in order to fully potentiate vectorisation opportunities and optimisations.

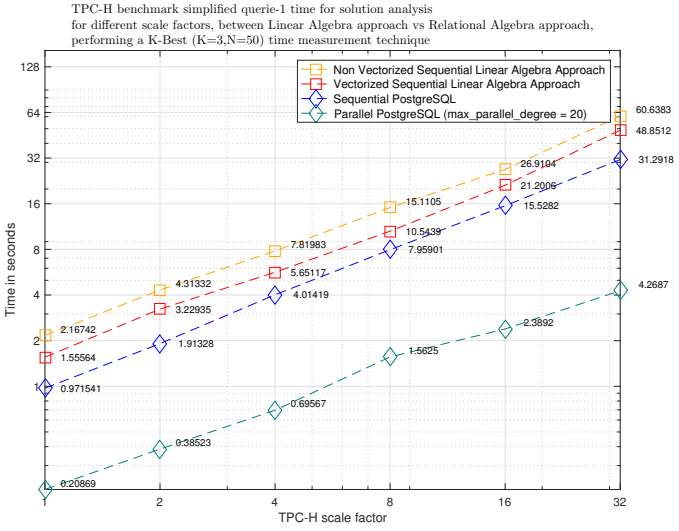

Figure 15: *Execution time for the simplified query-1 from TPC-H benchmark, for different scale factors (dataset sizes from 1GB to 32GB).*

IV. Parallelisation

I. Sequential Profiling

Prior to parallelisation every step of the linear algebra expression was profiled to identify potencial hotspots. Since the best parallel linear algebra version turned out to be the CSC version, special attention is added to it when compared to the CSR version.

Table 12 displays the profiler results for the CSC version with the largest TPC-H dataset in test - 32GB, namely the percentage of overall time for each operation present in expression 5.

LA Version	Projection	Selection	Projection . Selection	(Projection .Selection). Quantity	Bang
Seq. CSC	23.93%	43.38%	10.89%	18.12%	3.68%

Table 12: *Profiling results for the sequential CSC linear algebra version, for TPC-H 32GB dataset, for the evaluation platform.*

As shown in table 12, the most time consuming operations are the selection and the projection (Khratri-Rao operation). Further efforts to reduce the selection overhead are discussed in subsection III.

II. Parallel Experimental Results and Analysis

To parallelise both linear algebra versions, in a multicore system, we opted by OpenMP version 4.0. Using the CSC format in the linear algebra version the algorithms offer good parallelisation opportunities, since each column has at most one element.

To parallelise with the CSR format the algorithms are more challenging.

Figure 16 presents the measured execution times for a wide range of datasets, and for both linear and relational algebra approaches. The presented values were selected through the K-Best technique, with K=3 from 50 samples. For the linear algebra approach we present the the best execution times for the best CSR and CSC formats, namely the CSC format version.

As shown, the parallel linear algebra approach is faster than the parallel relational algebra, although larger TPC-H datasets reduce the gain from the linear algebra approach. Further algorithm optimisation will be discussed in subsection III.

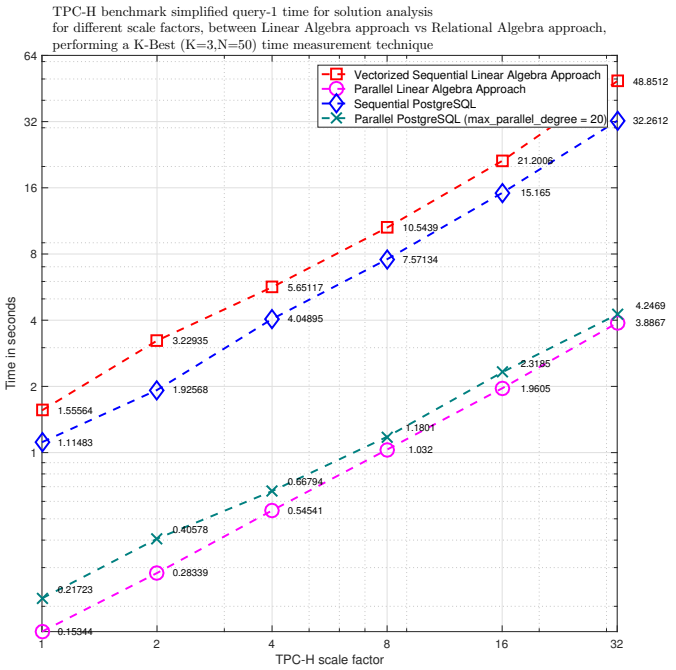

Figure 16: *TPC-H benchmark simplified query-1 time for solution analysis for different scale factors, between parallel linear and relational algebra approaches.*

III. Further Selection Algorithm Optimisation

Further profiling was made to better identify the potential bottlenecks. Table 13 compares both sequential and parallel linear algebra versions. The projection operation has decreased the percentage of overall time to complete the operation. However, the selection operation has increased its overall time percentage to almost half the overall value.

LA Version	Projection	Selection	Projection . Selection	(Projection .Selection). Quantity	Bang
Seq. CSC	23.93%	43.38%	10.89%	18.12%	3.68%
Par. CSC	13.62%	46.83%	11.49%	16.74%	3.51%

Table 13: *Profiling results for the parallel CSC linear algebra version, for TPC-H 32GB dataset, for the evaluation platform.*

After a careful examination of a portion the C source code for the CSC version selection, in listing 5, further algorithm optimisations can be made.

Listing 5: *Portion of the C source code for the CSC linear algebra version selection algorithm*

```
__assume_aligned(A_csc_values, MEM_LINE_SIZE);
__assume_aligned(A_row_ind, MEM_LINE_SIZE);
__assume_aligned(A_col_ptr, MEM_LINE_SIZE);

for ( int at_column = 0; at_column < A_number_columns; ++at_column){
  aux_csc_col_ptr[at_column] =  at_non_zero;
  const int iaa = A_row_ind[at_column] +1 ;
  non_zero = 0;
  field = (char*) g_quark_to_string ( iaa );
  returned_strcmp = strcmp( field, comparation_key);
  if (
      ( (opp_code == LESS)  && (returned_strcmp < 0 ))
      ||
      ( (opp_code == LESS_EQ)  && (returned_strcmp <= 0 ))
      ||
      ( (opp_code == GREATER)  && (returned_strcmp > 0 ))
      ||
      ( (opp_code == GREATER_EQ)  && (returned_strcmp >= 0 ))
     ){
    returned_strcmp2 = strcmp( field, comparation_key2);
    if (
        ( (opp_code2 == LESS)  && (returned_strcmp2 < 0) )
        ||
        ( (opp_code2 == LESS_EQ)  && (returned_strcmp2 <= 0) )
        ||
        ( (opp_code2 == GREATER)  && (returned_strcmp2 > 0) )
        ||
        ( (opp_code2 == GREATER_EQ)  && (returned_strcmp2 >= 0)
        )
       ){
      aux_csc_row_ind[at_non_zero] =  at_column;
      aux_csc_values[at_non_zero] =  A_csc_values[at_column];
      at_non_zero++;
    }
  }
}
```

Taking as example the TPC-H 32GB dataset, and the shipdate matrix, with dimension ($m \times n$), *namely*(2,531 $\times$ 192,000,551), to compute the selection result, in the worst case scenario, it is necessary to realize $(n * 2) = 384,001,102$ string comparisons. However, by analysing table 11, from the 384,001,102 string comparisons only 2,531 distinct strings will be tested.

Listing 6 avoids repetitive string comparisons by adding an additional auxiliar array (of size n), resulting in a worst case scenario of $(m * 2) = 5,062$ string comparisons and $n = 192,000,551$ integer comparisons.

This solution resolves two distinct problems: the excessive overhead of accessing a large amount and repetitive string comparisons, and the improvement of this solution with bigger datasets. The additional overhead of producing an auxiliar array gets diminished with the increase of the dataset and consequently the required comparisons. The larger the dataset, the better.

Listing 6: *Portion of the C source code for the CSC linear algebra version selection algorithm*

```
__assume_aligned(aux_row_bitmap, MEM_LINE_SIZE);
for ( int at_row = 0; at_row < A_number_rows; ++at_row){
  aux_csc_values[at_row] =  0;
  non_zero = 0;
  field = (char*) g_quark_to_string ( at_row+1 );
  returned_strcmp = strcmp( field, comparation_key);
  if (
      ( (opp_code == LESS)  && (returned_strcmp < 0 ))
      ||
      ( (opp_code == LESS_EQ)  && (returned_strcmp <= 0 ))
      ||
      ( (opp_code == GREATER)  && (returned_strcmp > 0 ))
      ||
      ( (opp_code == GREATER_EQ)  && (returned_strcmp >= 0 ))
     ){
    returned_strcmp2 = strcmp( field, comparation_key2);
    if (
        ( (opp_code2 == LESS)  && (returned_strcmp2 < 0) )
        ||
        ( (opp_code2 == LESS_EQ)  && (returned_strcmp2 <= 0) )
        ||
        ( (opp_code2 == GREATER)  && (returned_strcmp2 > 0) )
        ||
        ( (opp_code2 == GREATER_EQ)  && (returned_strcmp2 >= 0)
        )
       ){
      aux_row_bitmap[at_row] =  1;
    }
  }


__assume_aligned(A_csc_values, MEM_LINE_SIZE);
__assume_aligned(A_row_ind, MEM_LINE_SIZE);
__assume_aligned(A_col_ptr, MEM_LINE_SIZE);

for ( int at_column = 0; at_column < A_number_columns; ++at_column){
  aux_csc_col_ptr[at_column] =  at_non_zero;
  const int iaa = A_row_ind[at_column] +1 ;
  non_zero = 0;
  if ( aux_row_bitmap[iaa] == 1 ){
    aux_csc_row_ind[at_non_zero] =  at_column;
    aux_csc_values[at_non_zero] =  A_csc_values[at_column];
    at_non_zero++;
  }
}
```

Table 14 compares three parallel linear algebra versions: the sequential, the parallel, and an optimised parallel version. The selection operation largely diminished its percentage overall time making it possible for the overall time to be averagely distributed among operations.

LA Version	Projection	Selection	Projection . Selection	(Projection .Selection). Quantity	Bang
Seq. CSC	23.93%	43.38%	10.89%	18.12%	3.68%
Par. CSC	13.62%	46.83%	11.49%	16.74%	3.51%
Optimized Par. CSC	10.47%	20.10%	25.86%	34.60%	8.98%

Table 14: *Profiling results for the selection algorithm optimised parallel CSC linear algebra version, for TPC-H 32GB dataset, for the evaluation platform.*

IV. Final Parallel Experimental Results

Figure 17 presents the measured execution times for a wide range of datasets, and both linear and relational algebra approaches. The presented values were selected through the K-Best technique, with K=3 from 50 samples. For the linear algebra approach we present the the best solution with the CSC format, in the optimised parallel version.

As shown, the parallel linear algebra solution in both versions is faster than the parallel relational algebra, and the speedup was stable for whole range of TPC-H datasets.

Figure 18 plots the obtained speedups between sequential LA and parallel LA versions, and between the parallel LA and

RA versions.

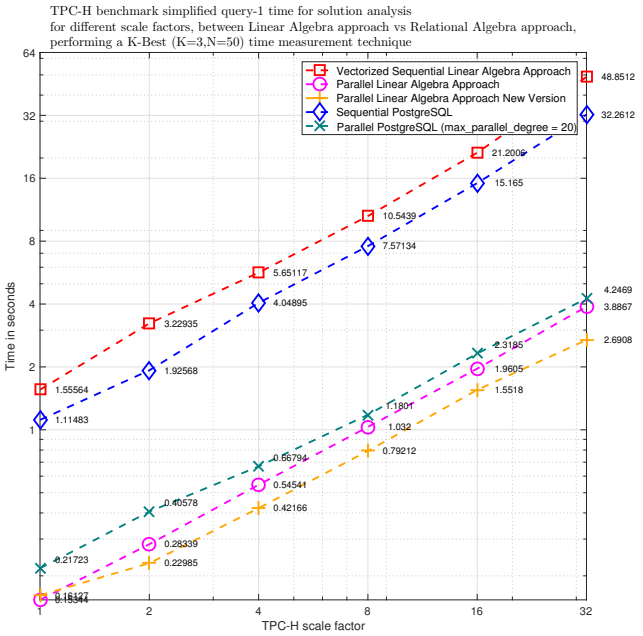

Figure 17: *TPC-H benchmark simplified query-1 time for solution analysis for different scale factors, between parallel linear and relational algebra approaches, in which linear algebra approach states the selection algorithm optimisation.*

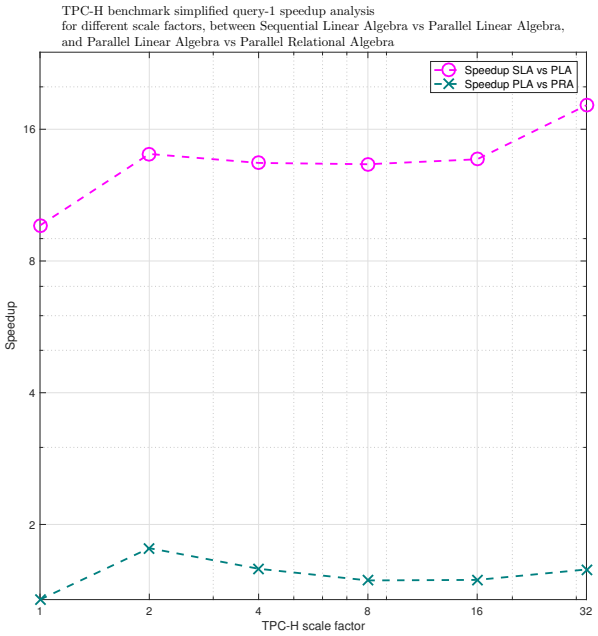

Figure 18: *TPC-H benchmark simplified query-1 speedup analysis for different scale factors, between sequential linear algebra vs parallel linear algebra and parallel linear algebra vs parallel relational algebra.*

V. Conclusion

We have designed, implemented and evaluated an efficient and parallel LA framework to provide fast responses to OLAP queries in very large databases. We started by developing a typed linear algebra solution to respond to OLAP TPC-H query 1, with faster response times than a Open Source relational algebra competitor, PostgresSQL.

A challenge in this project was to understand the linear algebra theory prior defined[3] [**?**], to fully represent relational algebra in terms of linear algebra operators. This process was longstanding and it was only managed to do it with help of our advisors. Efforts were made to correctly introduce the key ideias prior presented[3], and resolving potential difficulties that the previous implementations faced.

In spite the already 18x speedup regarding sequential linear algebra version and the 2x speedup regarding the parallel relational algebra engine, a lot of work can be done to further improve the algorithm, and fully analyse TPC-H queries results, as stated on section VI, leaving several potential exploratory paths.

VI. Future Work

The biggest problem of the linear algebra implementation can be the intermediate matrices, if a column has no nonzero element, they are not optimal for further parallelisations. This is the unique speedup

limitation

regarding the relational algebra approach. If that limitation is overcome the speedups would climb to more significant values when compared to the relational algebra version.

It is also clear that the keys for selection significantly interfere with both linear and relational algebra algorithms. In the LA specific case, a selection operation that returns a low number of nonzero elements might compromise the attainable speedup through parallelisation, since there might not be enough data to keep the computational units busy.

There is still room to greatly improve the parallelisation of this algorithm. For example, a gather/scatter analysis for distributed memory parallelism might result in larger attainable speedup. However, efforts need to me made to detect possible latency or bandwidth issues.

Another approach is to duplicate data in a shared memory environment . For the CSC format, regarding the three arrays, every thread would have the total CSC column pointer array and a portion of CSC values and CSC row indexes arrays. The computation of both splitted arrays would be thread independent and there would only be necessary to reduce the CSC column pointer.

To improve data locality the Block Compressed Sparse Row (BSR) Format could be used. However, it would largely increase the algorithm complexity for the defined methods. That improvement should only be used in the CSR linear algebra defined version.

Future work also includes extending the scope of both offline and run-time optimisations. These include:

- to fully translate the remaining TPC-H queries and benchmark both linear and relational algebra approaches;
- to investigate reordering methods to reduce total query compute time;
- to exploit index compression, further cache-blocking and TLB blocking to reduce memory traffic and to further improve locality;
- to implement other matrix partitioning schemes;
- to improve load balance when there are different data structures for each generated matrix;
- to reduce data structure conversion costs at run-time, for the CSR linear algebra version;
- to determine the most efficient number of cores for parallelism.

The usage of CUDA and MIC systems should also be explored as an heterogenous system solution. Since the algorithms require huge portions of simple computation, and it is already parallelised via OpenMP, porting the application to be Xeon Phi compatible should not present great challenges.

VII. Acknowledgements

We would like to thank our advisors for all the help they gave us and for always being available for all the questions we had. We would also like to thank Rogério Pontes for all the help he gave us in understanding how the linear algebra theory would fit in OLAP, and all the time he spent helping us improve the parallel algorithms.

References

[1] Rogério António da Costa Pontes. Benchmarking a linear algebra approach to olap master thesis. 2015.

[2] Hugo Daniel Macedo and José Nuno Oliveira. Do the middle letters of "olap" stand for linear algebra ("la"). Technical report, Technical Report TR-HASLab: 04: 2011, INESC TEC and University of Minho, Gualtar Campus, Braga, 2011.

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

[4] Malik Silva. Sparse matrix storage revisited. In *Proceedings of the 2nd conference on Computing frontiers*, pages 230–235. ACM, 2005.