Converting **SQL** into a **Linear Algebra DSL**

Luís Albuquerque, Rafael Fernandes

Supervision: Pedro Henriques , Alberto Proença.

Tutoring : João Afonso.

University of Minho

Abstract

In traditional database systems OLAP is still based on the relational model, where SQL is the standard language for data query. Novel DBMS are exploring linear algebra operations and types theory as a more efficient and robust engine to access data, using a new Linear Algebra Query (LAQ) language. To maintain SQL to use this engine a SQL driver is required to convert SQL into LAQ. In this work we developed a prototype of a SQL driver that generates efficient LAQ expressions and we validated it with 2 queries from the TPC-H benchmark.

Keywords: LAQ, Linear Algebra, SQL, SQL Parser, DataBase, OLAP, C++

Contents

List of Figures

1. Introduction

Our colleagues at *University of Minho* have been studying the application of *Linear Algebra* to optimize data bases and are currently developing their own column-oriented database with it, as an alternative to the more traditional row-oriented databases. To prove their work they need to compare their data base's performance with others using the same queries.

And that is where our job begins: provide all the tools necessary to translate several SQL queries from TPC-H benchmark re-writing SQL language to suit their DSL and output it in the most efficient way (*See chapter* 4.4).
We divided this processes into two smaller ones: the SQL Parser and the SQL Rewriter/-Converter.

1.1. SQL Parser

SQL's language is highly redundant and most of its queries/functions do not have a one to one conversion. To facilitate this process we had to create a parser that covers most of SQL's syntax and stores it on structures suited for it. After a long period of testing and debating we created two structures that archive that: *Ltree*,to store and control the order of each expression on the *WHERE* clause;Graph,to store every field present on queries as well as each tables's metadata, provided by the user.

1.2. SQL Rewriter/Converter

As stated before, most of SQL's queries/functions don't have a one to one conversion, so we had to create programs to control and ultimately convert our SQL parsing tree, constituted by those two structures described above. As previous stated: we should always try to output the DSL grammar in the most efficient way, to archive this we used our knowledge of linear algebra, with João Afonso's supervsion.

2. Analysis and Specification

2.1. Informal Description of the Problem

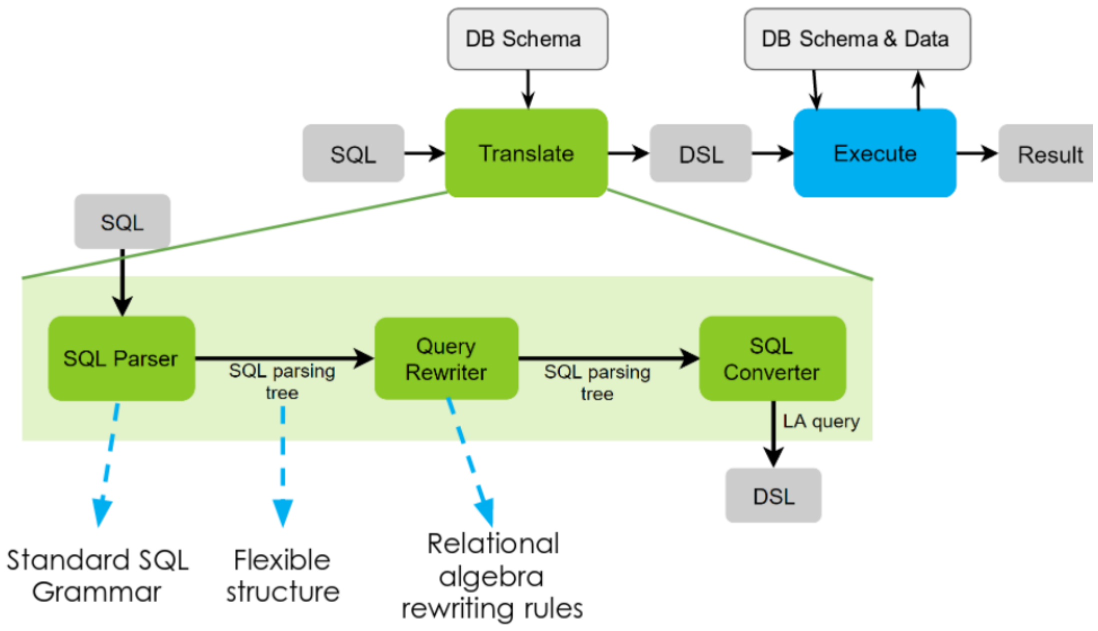


Figure 1: Overall process

As can be seen in the image (green), to execute our part of the process we need the data base's schema because to be able to solve the tree we need information that's not present on SQL's queries. SQL's syntax has lots of tokens/functions that their users can use, we tried to abstract as much as we could but in some cases we needed to completely rewrite it to make sence. The biggest problem was the WHERE clause because the expressions that follow it up had a hierarchical structure, which creates the necessity of a binary tree, since it was constituted by ANDs, ORs and expressions, we called it *Ltree* that both stands for **L**ogical tree or **L**eaf tree. Graph on the other hand has all the other information from SQL's queries that are useful for the DSL as well as a copy of WHERE's expressions to ease the parsing.

2.2. *Requirements Specifications*

The required work can be implemented as a set of tasks:

- *Develop (or download) a SQL grammar for the parser;*
- *Define and implement a generic structure to support query manipulation;*
- *Implement some rewriting rules (keeping the query semantic), e.g., CASE WHERE,or IN EXISTS Semi join;*
- *Convert the SQL parsing tree into a DSL description of the LA approach;*

To further understand their request here is an example of a input/output they need:

querie.png

```
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 0.50 - 0.01 and 0.50 + 0.01
    and l_quantity < 3;
```

Figure 2: Input

querie.png

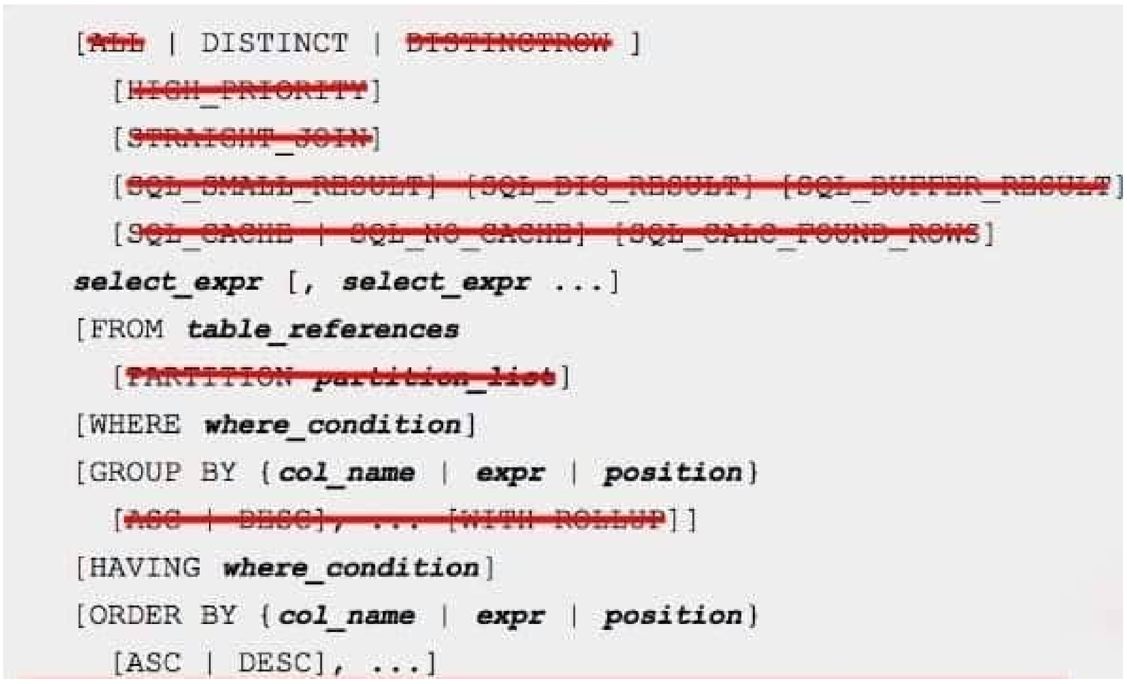
```
A = filter( l.shipdate >= "1995-03-10" AND l.shipdate < "1996-03-10" )
B = filter( l.discount >= 0.49 AND l.discount <= 0.51 )
C = hadamard( A, B )
D = map( l.extendedprice * l.discount )
E = filter( l.quantity < 3 )
F = hadamard( D, E )
G = hadamard( C, F )
H = sum( G )
return( H )
```

Figure 3: Output

3. Development Stages

3.1. Parser

Since we couldn't find SQL's grammar that fitted our needs, we had to learn SQL's syntax and create our own. Which currently, even tho it doesn't cover all the cases (SQL's syntax is huge) we made it easy to add new cases to the parser, not only that, we added some cases that, while useless right now, will be useful eventually. As for now, we only store the contents of SELECT, FROM, WHERE and GROUP BY clauses.



```
[~~ALL~~ | DISTINCT | ~~DISTINCTROW~~ ]
  [~~HIGH_PRIORITY~~]
  [~~STRAIGHT_JOIN~~]
  [~~SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT~~]
  [~~SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS~~]
select_expr [, select_expr ...]
[FROM table_references
  [~~PARTITION partition_list~~]
[WHERE where_condition]
[GROUP BY {col_name | expr | position}
  [~~ASC | DESC], ... [WITH ROLLUP~~]]
[HAVING where_condition]
[ORDER BY {col_name | expr | position}
  [ASC | DESC], ...]
```

Figure 4: SQL's syntax

3.2. SQL Parsing Tree

We tried a variety of models for our SQL Parsing Tree, the early ones where written on C since only recently we started using C++ to create/manipulate them. Most of them rely only on one structure because we, until the latest stages of development, didn't know we needed to follow a specific order present on the WHERE clause (thus the need of a Ltree) which was quite impactful on our project because it completely changed the order with we returned, and because they depend on previous returned values, the content itself changed, making it way more complex from what we had previously.

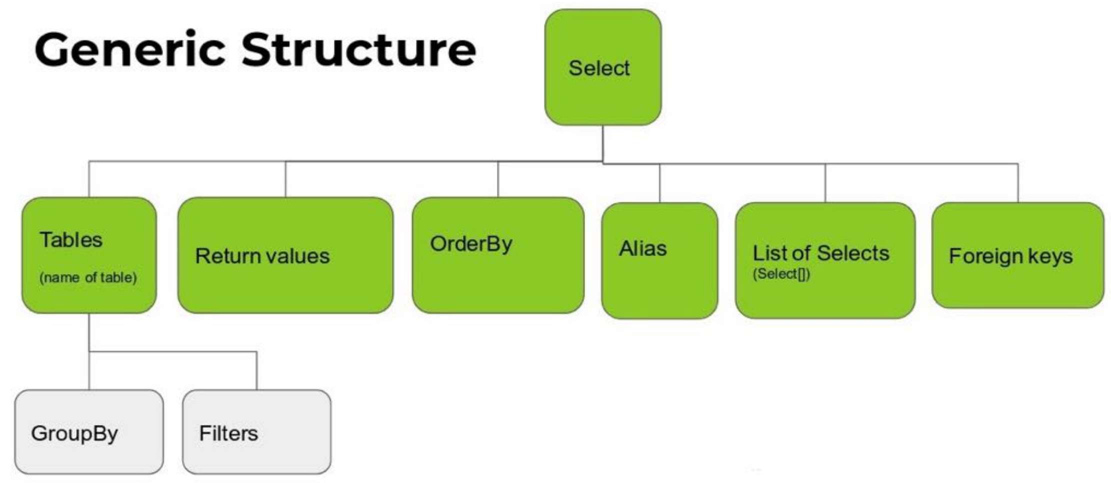


Figure 5: First Prototype

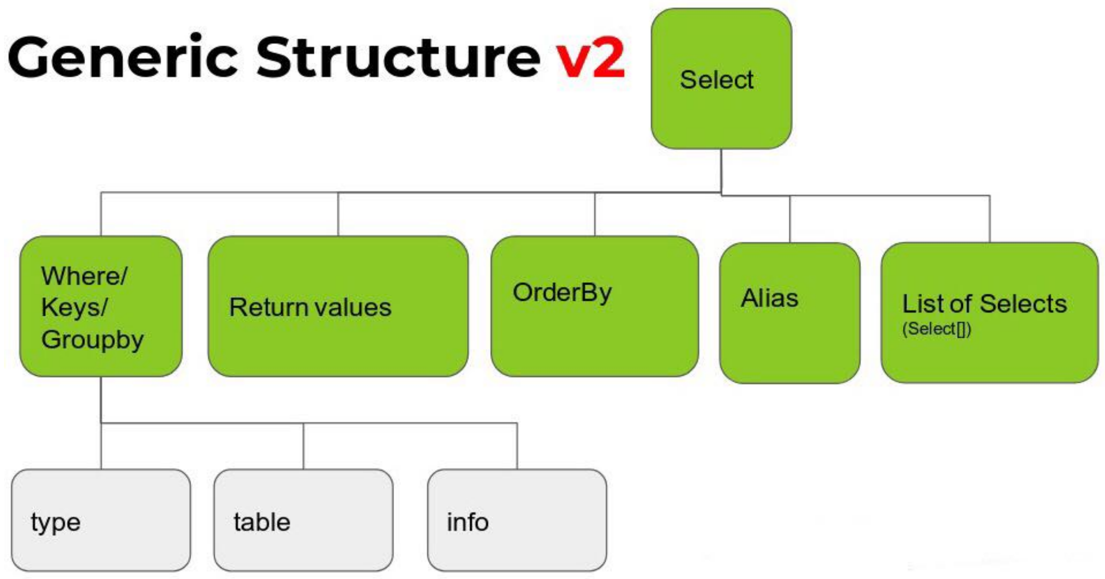
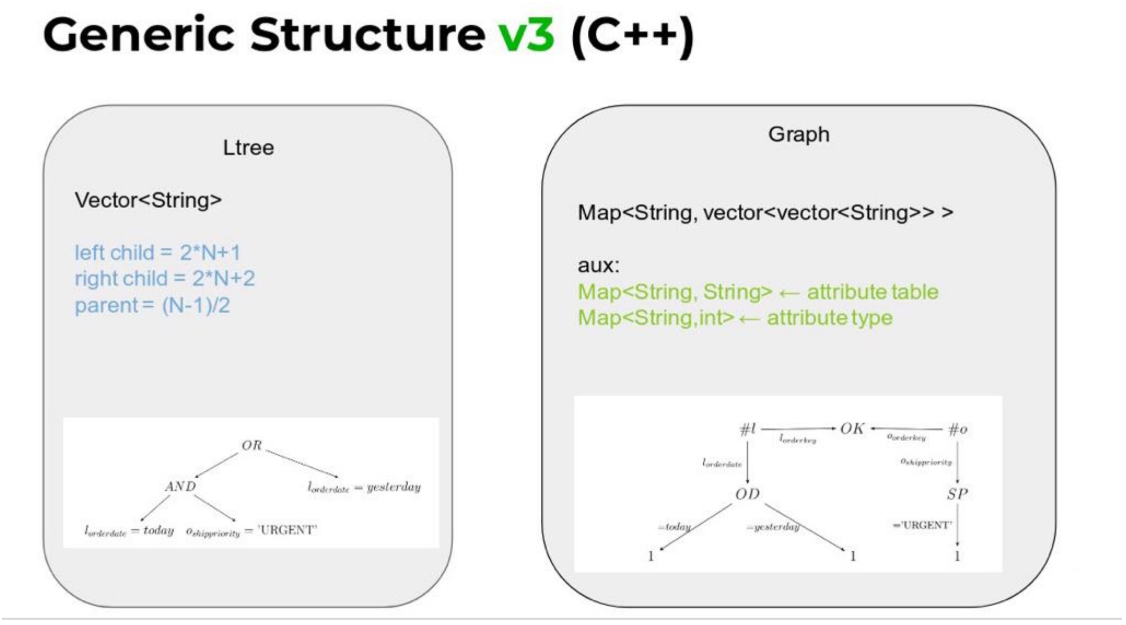


Figure 6: Second Prototype



Generic Structure v3 (C++)

Ltree

Vector<String>

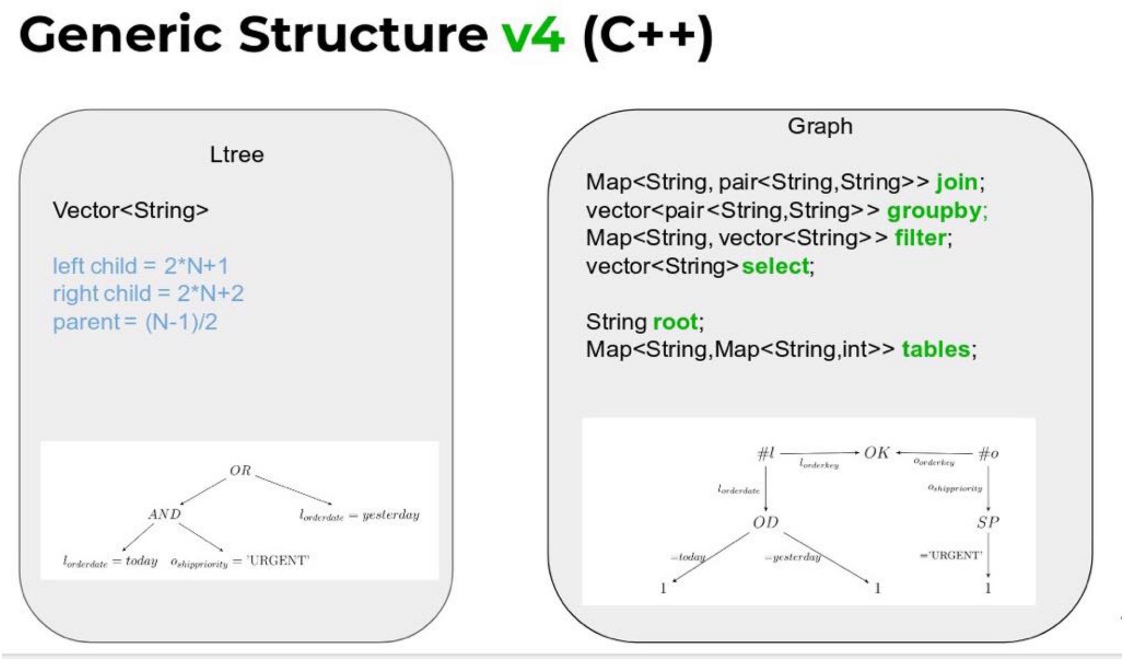
left child = 2*N+1
right child = 2*N+2
parent = (N-1)/2

Graph

Map<String, vector<vector<String>> >

aux:
Map<String, String> ← attribute table
Map<String,int> ← attribute type

Figure 7: Third Prototype



Generic Structure v4 (C++)

Ltree

Vector<String>

left child = 2*N+1
right child = 2*N+2
parent = (N-1)/2

Graph

Map<String, pair<String,String>> **join**;
vector<pair<String,String>> **groupby**;
Map<String, vector<String>> **filter**;
vector<String> **select**;

String **root**;
Map<String,Map<String,int>> **tables**;

Figure 8: Final Structure

4. Current stage

4.1. File organization

Our work is partitioned into several files, to facilitate reading and comprehension, and to facilitate the work of those who wish to use it in the future.

- **SQLtoLAQ.ll** Is lex, which is to recognize and establish a syntax for each word in the SQL language and associate it with a token, and in some cases also associate a value with it.

- **SQLtoLAQ.y** which is the parser, where we read the file, define the SQL grammar we receive as input, interpret SQL and, depending on the type of query, we call the respective functions that construct, manipulate and print structures.

- **SQLtoLAQ.hpp** This is where Par, Graph, and Ltree are defined, and it is where all the headers of the class methods are found. It is in this document that much of the documentation of the functions can be found, so it is easy to see what functions have already been defined and what are done to build new functions. This file has not only the headers of the class methods but also all the function declarations that are used in all the files.

- **SQLtoLAQ.cpp** This file is where the class methods and all the functions are found.

4.2. Structures

Currently our program doens't read data base's Schema, and this process can only be archived by inserting directly on the structure *Graph*, but the process of adding that feature will be quick once we know how will we receive them.
This are our current structures:

Graph is constituted by:

- **root:** graph's root;
- **join:** SQL's JOIN clause content and respective keys;
- **groupby:** SQL's GROUP BY clause content;
- **tables:** Data bases's schema without keys;
- **filters:** SQL's WHERE clause content;
- **select:** SQL's SELECT clause content;

Ltree is constituted by:

- **ltree:** logical tree as described previously;

Code:

```
class Graph {
    string root;
    vector<vector<string>  > join;
    vector<pair <string,string> > groupby;
    map<string,map<string,string> > tables;
    map<string, vector<string> > filter;
    vector< pair<string,string> > select;
}

class Ltree {
    vector<string> ltree;
}
```

We also created the necessary functions to manipulate them, using C++'s libraries to ease all the process. For a list of all classes's functions see Appendix A.

4.3. Parser

As stated before we tried to abstract as much as we could from SQL's grammar due it redundancy, which means that our parser covers most of it but at the same time covers almost no rules in special, this forces the users to input to always a functional SQL querie or else, instead of throwing an error, it will run anyway but, as expected, it's output will not make sense. All cases covered are present on Appendix B.

4.4. Parsing Tree

As requested by the group currently working on *LAQ*, while solving our parsing tree we try to output what would be the "natural/fastest" for LAQ grammar. To understand this we need to explain some characteristics of LAQ:

- Dot product is expensive and slow so we should avoid it at all cost;
- We can't solve an OR clause if its childs (expressions it connects) aren't of the same table;
- SELECT clause's content always appears on GROUPBY (unless it is an aggregation function), which means that if we already joined groupby's content we dont need to add select's content that matches it, leaving only the aggregation functions;

- Aggregations functions need it's content joined with the resulting filter before they apply themselves to it;

Summing up this is how our solver works:

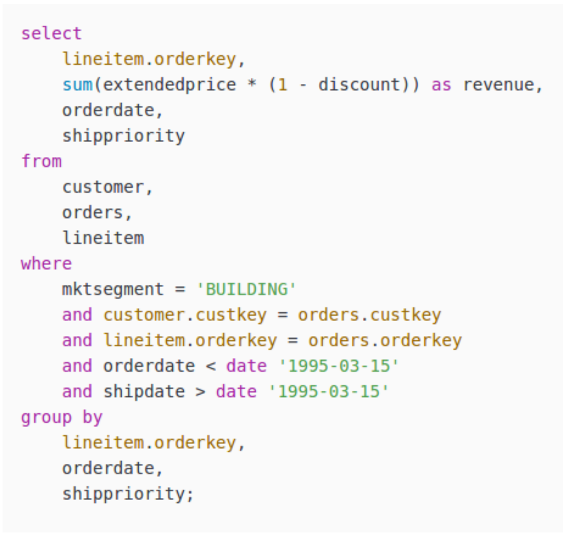
- We do a bottom up crossing on both Ltree and Graph but, depending on the current Ltree's node the graph we have to solve may or may not have all attributes.
- To maintain it's consistency we solve a copy of our main graph and merge with it after its solved.
- If when solving Ltree we come across a "AND" whose childs consist only of expressions and other "AND"s we can/should follow graph's order to solve them as it minimizes the amount of dot products used.
- Dimensions (a type of attribute) must always combine filters of itself before combining with the rest of it's table's attributes/filters, measures on the other hand can be combined in any order as long as they are of the same table.
- Groupby's content must be the last combining element of a table and should follow the order present on SQL's queries.
- When a table is constituted only by one filter and it isn't graph's root it should move it's content to the tables which it has foreign keys (in SQL terminology: it the tables that are pointing to it) .
- Since Select's content must be present on groupby if it isn't a aggregation function, when we finish solving our main graph we just need to include it on return.
- Aggregation functions on the other hand need some changes: first we take its content and use map() on it, then we krao() its result with graph's filter (at this point there is only one filter on it) and finally apply the aggregation function to the result of the previous step and add it to return;
- When applying dot product to all tables (in case an "OR" with dependencies) this application should always be top-down as any other approach would result on attributes moving to the next tables more then the necessary.

This how in general this process is in pseudo-code:

```
void graphWorkAux(string type):
  finds one of graphs tips;
  join filters on the same attribute that are dimensions on that table;
  join every content of that table;
  join respective groupbys;
  if there is more tables then remove it;

void graphWork(string type):
  while (is not only one attribute):
    graphWorkAux(type);

void resolveS(int indice, string type):
  array with indice childs;
  clones main graph with only filters present in v;
  solves the clone (graphWork);
  update tree;
  removes from main graph solved filters ;

void resolve(int indice):
  if( it's an "OR" we need to solve it's childs first):
    resolve(ind_left_child(indice));
    resolve(ind_right_child(indice));
    if(it has no dependencies solve it)):
      resolveS(indice,"OR");
    else we apply dot product to every table and try again:
      dot_all();
      resolve(indice);
  if(it's an "AND"):
    if(one of its children is an "OR" solve its children first)):
      resolve(ind_left_child(indice));
      resolve(ind_right_child(indice));
      resolveS(indice,"AND");
    else we follow graph's order which uses less dot products:
      resolveS(indice,"AND");

void returnf():
  array<string> aux;
  if(not select *):
    for(every entry on select):
      if(not an aggregation funtion):
        add to aux;
      else:
        print: letter1 = map(aggregation's content);
        print: letter2 = krao( letter1, result of parsing);
        print: rename = aggr-function (letter2);
        add rename to aux;
    print: return(every entry of aux);
  else:
    print: select * (not done yet);

main():
    resolve(0);
    dot if the only filter isn't on graph's root;
    returnf();
```

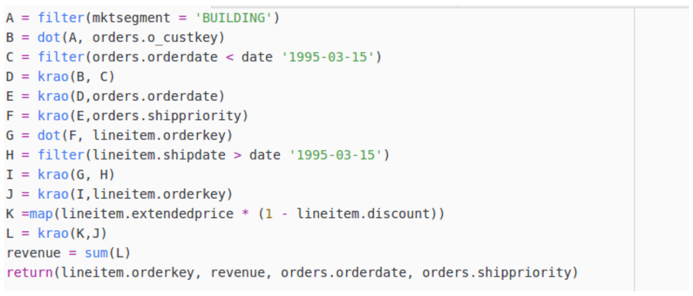
5. Tests/Results

Unfortunately we can not show our parser since by the time we were making this report it was not working so, after manually inputting SQL's queries on our structures we were able to test our solver which had good results. We also manage to avoid having to force the user to input before each attribute it's respective table by searching every individual string on data base's schema.
While our output has to be the optimal for LAQ's performace, time efficiency was not needed in this project since after outputting LAQ's version of a querie there is no need to input that same querie again.
Here are some results:



```
select
    lineitem.orderkey,
    sum(extendedprice * (1 - discount)) as revenue,
    orderdate,
    shippriority
from
    customer,
    orders,
    lineitem
where
    mktsegment = 'BUILDING'
    and customer.custkey = orders.custkey
    and lineitem.orderkey = orders.orderkey
    and orderdate < date '1995-03-15'
    and shipdate > date '1995-03-15'
group by
    lineitem.orderkey,
    orderdate,
    shippriority;
```

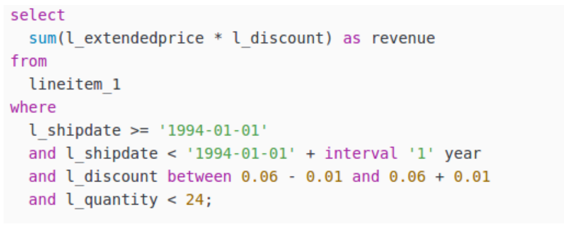
(a) SQL's code



```
A = filter(mktsegment = 'BUILDING')
B = dot(A, orders.o_custkey)
C = filter(orders.orderdate < date '1995-03-15')
D = krao(B, C)
E = krao(D,orders.orderdate)
F = krao(E,orders.shippriority)
G = dot(F, lineitem.orderkey)
H = filter(lineitem.shipdate > date '1995-03-15')
I = krao(G, H)
J = krao(I,lineitem.orderkey)
K =map(lineitem.extendedprice * (1 - lineitem.discount))
L = krao(K,J)
revenue = sum(L)
return(lineitem.orderkey, revenue, orders.orderdate, orders.shippriority)
```

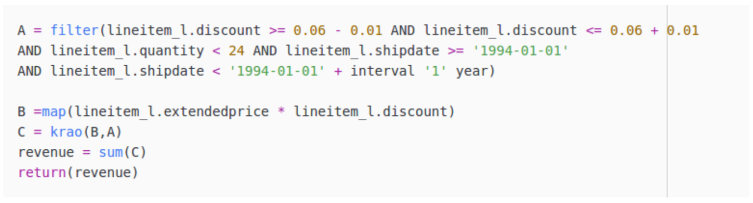
(b) LAQ's code

Figure 9: Querie 3 from TPC-H



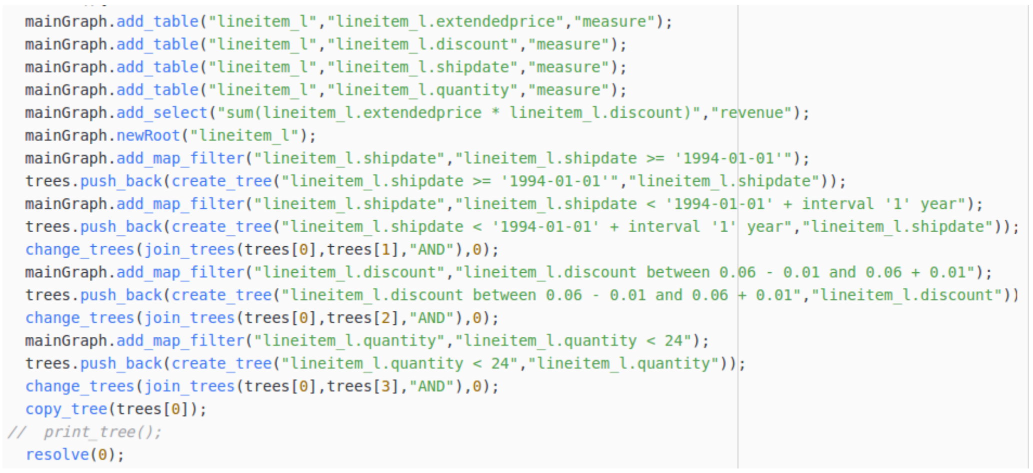
```
select
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem_1
where
  l_shipdate >= '1994-01-01'
  and l_shipdate < '1994-01-01' + interval '1' year
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24;
```

(a) SQL's code



```
A = filter(lineitem_l.discount >= 0.06 - 0.01 AND lineitem_l.discount <= 0.06 + 0.01
AND lineitem_l.quantity < 24 AND lineitem_l.shipdate >= '1994-01-01'
AND lineitem_l.shipdate < '1994-01-01' + interval '1' year)

B =map(lineitem_l.extendedprice * lineitem_l.discount)
C = krao(B,A)
revenue = sum(C)
return(revenue)
```

(b) LAQ's code



```
  mainGraph.add_table("lineitem_l","lineitem_l.extendedprice","measure");
  mainGraph.add_table("lineitem_l","lineitem_l.discount","measure");
  mainGraph.add_table("lineitem_l","lineitem_l.shipdate","measure");
  mainGraph.add_table("lineitem_l","lineitem_l.quantity","measure");
  mainGraph.add_select("sum(lineitem_l.extendedprice * lineitem_l.discount)","revenue");
  mainGraph.newRoot("lineitem_l");
  mainGraph.add_map_filter("lineitem_l.shipdate","lineitem_l.shipdate >= '1994-01-01'");
  trees.push_back(create_tree("lineitem_l.shipdate >= '1994-01-01'","lineitem_l.shipdate"));
  mainGraph.add_map_filter("lineitem_l.shipdate","lineitem_l.shipdate < '1994-01-01' + interval '1' year");
  trees.push_back(create_tree("lineitem_l.shipdate < '1994-01-01' + interval '1' year","lineitem_l.shipdate"));
  change_trees(join_trees(trees[0],trees[1],"AND"),0);
  mainGraph.add_map_filter("lineitem_l.discount","lineitem_l.discount between 0.06 - 0.01 and 0.06 + 0.01");
  trees.push_back(create_tree("lineitem_l.discount between 0.06 - 0.01 and 0.06 + 0.01","lineitem_l.discount"))
  change_trees(join_trees(trees[0],trees[2],"AND"),0);
  mainGraph.add_map_filter("lineitem_l.quantity","lineitem_l.quantity < 24");
  trees.push_back(create_tree("lineitem_l.quantity < 24","lineitem_l.quantity"));
  change_trees(join_trees(trees[0],trees[3],"AND"),0);
  copy_tree(trees[0]);
//  print_tree();
  resolve(0);
```

(c) Parser's function calls

Figure 10: Querie 6 from TPC-H

6. Conclusion

In this work, we have achieved the goal of converting SQL into LAQ.

However, we have not been able to cover the translation of all SQL queries in LAQ. This was due to the fact that the work was very confusing, without very concrete ideas about what to do and how to do it. Throughout the meetings, we were shaping the work, organizing ideas, sometimes creating structures, sometimes re-creating, at some point until we moved from C language, where we started work for C ++. But in fact that many times that we advanced we reached a point that we were forced to reformulate the work since it did not obey the specification. Which took us a long time to complete the project.
But at the end of the day we were able to architect the parser and lex and treat the data correctly so that our work is a pipe that enters SQL and leaves LAQ.

7. Future work

Although our program covers most of SQL queries, it does not cover all of them. In order for LAQ to be used as a database language, in the future it would be necessary to add to the grammar other expressions that our grammar does not accept / resolve, and indicate how these new cases would be treated. Most of the functions that manipulate the data and structures are already created so in most cases the resolution of a query that is not resolved at this moment would be just a set of functions already defined. At work we focused on the request, and on what LAQ was up to date. Some SQL query are already accepted by the parser, but since they are not accepted / we do not know how to handle the data, they are ignored when processing the data. With the evolution of the language of exit some things may have to be changed, that is why we leave the best organization possible, because in this case who add will have to understand what we have done.

Appendix A. Classes

```
class Graph{
        Graph clone(vector<string> v);
        void newRoot(string newRoot);
        int num_attributes(string str);
        void add_filter(string Table,string filter, int type);
        void remove_filter(string Table,string filter);
        int search_filter_in_table(string table, string filter);
        vector<string> search_filter(string filter);
        int search_table(string table);
        void add_table(string Table, string filter, string type);
        void remove_table(string Table);
        void add_join(string fk,string table1, string table2);
        void remove_join(string fk);
        void add_groupby(string Table,string filter);
        void remove_groupby(string Table,string filter);
        void add_map_filter(string Filter, string expression);
        void remove_map_Table_filter(string Table);
        bool search_map_filter(string s);
        string search_map_filter2(string s);
        string search_table_filter(string s);
        void remove_map_filter(string Table,string Filter);
        int is_measured(string Table, string filter);
        int is_dimension(string Table, string filter);
        void add_table(string Table, string name);
        int isThe_same_Table(string attribute1, string attribute2);
        int isThe_same_Table_array(vector<string> v);
    }
    class Ltree{
        Ltree();
        Ltree(string s);
        string Parent(int indice);
        string left_child(int indice);
        int ind_left_child(int indice);
        string right_child(int indice);
        int ind_right_child(int indice);
        vector<string> childs_aux(vector<string> v1, vector<string> v2);
        vector<string> childs(int ind);
        void erasechilds(int ind);
        void add(string value ,int indice );
        int indice(string value);
        void swap(int indice1,int indice2 );
        string common_ancestor( int indice1, int indice2 );
        void merge(vector<string> vec);
        vector<string> split_aux(vector<string> res,int ind1, int ind2);
        vector<string> split(int indice1, int indice2);
        void insere(vector<string>::iterator it_ind, string s);
        void add_tree(vector<string> v, int ind, int n=0);
        int ind_Parent(int indice);
        vector<int> parents(int i);
        string all_same_table_aux(int indice);
        bool all_same_table(int indice);
        bool dependencies(int indice);
        string relacao_entre(vector<string> v, int s);
        int push_aux(int ind);
        void pushLT(int ind);
        void rewrite(int ind);
    }
```

Appendix B. Parser

```
SelectBlock    : SELECT     selectList
                 FROM       fromList
                 WHERE_
                 GROUPBY_
                 ORDERBY_
                 ';'

WHERE_         : WHERE       whereList
               |
               ;

GROUPBY_       : GROUPBY    groupbyList
               |
               ;

ORDERBY_       : ORDERBY    orderbyList
               |
               ;

selectList     : selectListN
               | '*'
               ;

selectListN    : selectListNSub
               | selectListN ',' selectListNSub
               ;

selectListNSub : Term
               | Term AS NAME
               ;

fromList       : subfromList
               | fromList ',' subfromList
               ;

subfromList    : NAME
               | Join NAME
               | Join NAME ON Literal '=' Literal
               | Join NAME AS NAME
               | Join NAME AS NAME ON Literal '=' Literal
               | '{' SelectBlock '}'
```

```
                | '{' SelectBlock '}' AS NAME
                ;

Join            : JOIN
                | INNER JOIN
                | LEFT JOIN
                | RIGHT JOIN
                | FULL JOIN
                ;

whereList       : Exp
                ;

ExpR            : Exp AND Exp
                | Exp OR  Exp
                ;

Exp             : Term
                | ExpR
                | '(' ExpR ')'
                | NAME IN Inlist
                | NAME BETWEEN Literal AND Literal
                | EXISTS '(' SelectBlock ')' ';'
                | Literal LIKE REGEX
                ;

Term            : Factor
                | Term BBOP Term
                | Term IBOP Term
                ;

Factor          : Literal
                | NAME '(' Args ')'
                | NOT Factor
                ;

Args            : Args1
                |
                ;

Args1           : Term
                | Args1 ',' Term
                | Args1 ' ' Term
                ;

groupbyList     : groupbyListSub
                | groupbyList ',' groupbyListSub
                ;

groupbyListSub : HAVING NAME '(' Literal ')' BBOP Literal
                | Literal
                ;

orderbyList     : orderbyListSub
                | orderbyList ',' orderbyListSub

                ;

orderbyListSub : NAME
                | NAME order
                ;

order           : ASC
                | DESC

Literal         : NAME
                | NAME '.' NAME
                | DATE
                | CONSTANT
                | BOOL
                | ANY
                | ALL
                ;
```

References