Laq-Olap - Design of a Columnar Database System based on Linear Algebra Querying

Lucas Pereira and Tiago Baptista

Master in Computer Science and Engineering (2019/20)
University of Minho, 4700-320 Braga, Portugal
`www.uminho.pt`

Abstract. This report addresses performance and interface improvements to an online analytical processing (OLAP) approach based on linear algebra (LA), as an alternative to the conventional relational algebra (RA). The initial OLAP approach was evaluated with the TPC-H benchmark and it proved faster than conventional database systems, but it did not manage to overcome the performance of the columnar MonetDB system. The current work identified the key features of MonetDB to improve the LA approach. The LA system interface accepts SQL queries, which are individually converted into type diagrams (TD) and a domain specification language (DSL) of the processing steps. The current work went further and designed and built a graphic user interface (GUI) that interactively creates and edits diagram schemes to replace SQL queries.

Keywords: OLAP, Columnar Database, Linear Algebra (LA)
Database Interface

1 Context

The project reported in this document was developed under the Masters in Computer Science and Engineering at the University of Minho. The project was proposed by Professors Alberto José Proença and José Nuno Oliveira.

The background of this work is an existing OLAP approach (Afonso et al., 2018) that had been evaluated with the TPC-H benchmark (Nambiar et al., 2009). It proved faster than conventional database systems, but it did not manage to overcome the performance of columnar systems, namely MonetDB (Idreos et al., 2012).

2 Introduction

In the age of information, almost every company or service depends on data storage reliability, ease and speed of access. This dependency on fast access to data increases when we look into the scenery of big companies and the handling of big data banks. So, there is a need for highly efficient and reliable database systems capable of managing and processing massive quantities of information that they make available to users or applications to operate upon.

The columnar database model is currently among those that offer more efficiency in data warehouses (OLAP systems) that support corporate decision making. With this in mind, Afonso et al. (2018) tried to approach the columnar database model in a way inspired by linear algebra, showing its advantages over the more standard relation algebra with some extent of success. Using the *TPC-H* benchmark, the approach beat all the row oriented databases under test but failed to match the results of *MonetDB*.

The main motivation for this work is to study the approach of (Afonso et al., 2018) and try and tune it in order to beat the performance of *MonetDB*. Another objective is to build the prototype of a Web-interface that should make query planning easier to build, using a graphical approach based on type diagrams that dispense with SQL verbalization.

To achieve this there was a long process of study and contextualization with all the work previously done as well as the study of MonetDB internals and its algebra.

2.1 Research Plan

The question that this project tried to answer was, if it was possible to improve the *LAQ* approach so that the solution could outperform all of the columnar databases and also, if it was possible to represent graphically linear algebra queries using type diagrams to accomplish it.

Obviously given the complexity of the problem and the limited amount of time available to develop it, there was the expectation of at least develop the graphical interface and advance with some theoretical explanations, trying to explain where are the differences between the *LAQ* engine when compared to *MonetDB* and other columnar databases.

Our contribution with this work is :

– The development of a graphic interface that allows the creation of Type Diagrams (TD) that allow to write queries;
– The thoroughly study of MonetDB engine and various related applications;
– The analysis of the *MonetDB* engine and the comparison between it's query plans and the *LAQ* engine.

3 State of the Art

In order to be able to understand the developed work, it is necessary to have a clear understanding of some key concepts, namely OLAP, Linear Algebra (LA) and Relational Algebra (RA) as well as the *LAQ* engine.

OLAP refers to Online Analytical Processing, it is a technology for data discovery, complex analytical calculation and data visualization. This data warehouses deals with huge amounts of data and performs queries over it to create views on specific information, so that the response time of those queries is much inferior than a normal query on non processed data.

The most used approach uses Relational Algebra (RA), but a new approach with Linear Algebra (LA) is now defying the RA, with great success even outperforming it. This typed LA gives place to a complex data querying engine based on a minimal LA kernel. Linear Algebra provides a new way to build query plans as paths of typed LA diagrams, ensuring type correctness by construction. An evaluation of the approach using queries of the *TPC-H* benchmark suite, which includes a comparison with two widely used and industry proven databases, *SQL* and *PostgreSQL*. (Afonso et al., 2018)

The more recent Linear Algebra (LA) relies on the efficiency of the operators and in the specific case of the *LAQ* engine on his optimizations and parallel execution.

This engine was based on the approach defended by (Oliveira and Macedo, 2017), specially the LAoP (Linear Algebra of Programming) that exposes linear algebra as an evolution of the (relational) algebra of programming. The main goal of the engine is to represent data and perform analytical querying in a single, unified framework (Afonso et al., 2018).

On the other hand MonetDB was first established at Centrum Wiskunde & Informatics in 1993, is a high-performance column-oriented relational database system. The great motivation for the development of this system arose from the need to work with large amounts of data from Data Mining and Data Warehouse applications. The MonetDB has several characteristics of which stand out:

- Front-end Independence;
- Column orientation;
- Binary Association Tables.

There are five variations of this database system in particular:

- MonetDB/XQuery : Database solution for XML use;
- MonetDB Server : MonetDB Database Server Solution;
- MonetDB/X100 : Recommended solution for OLAP applications;
- MonetDB/GIS : Spatial Database Solution;
- MonetDB/SQL : Relational database solution;

Comparing relational database systems with columnar systems, it can be said that the former are row-oriented and are often used for traditional applications. Columnar databases are optimized for column organization and are therefore more used for analytical applications.

Column-oriented storage is an important factor in analytical query performance as it significantly reduces overall disk I/O requirements, decreasing the amount of data you need to load from disk.

These databases were created to increase the horizontal scale using distributed clusters, therefore lowering costs. As such, they are ideal for data warehousing and big data processing.

3.1 LAQ Operators

In this section it's presented a simplified and general overview on Afonso et al. work, and the algebra used by the *LAQ* engine. The operators used are the following:

- Khatri-Rao - A $\triangledown$ B = C, each row of matrix A is multiplied by the whole matrix B (row by row). Matrices A and B have dimensions (i$\times$k) $\times$ (j$\times$k), respectively, and the result matrix will have ((i * j) $\times$ k);
- Dot product A $\cdot$ B = C, each element in a row i of matrix A is multiplied by the element in the column j on matrix B. Matrices A and B have dimensions (i $\times$ k) $\times$ (k $\times$ j) (number of columns in matrix A needs to be equal to number of rows in matrix B.
- Hadamard-Schur A $\times$ B = C, each element of matrix A is multiplied by the element in the same positions in matrix B. Matrices A and B must have the same dimensions and therefore, matrix C has them too;
- Filter - , relational selection;
- Fold - fold(A) = M $\cdot$ $!^{\circ}$, using composition(dot product) with $!^{\circ}$ (pronounced Bang), aggregation function *sum* and *count* are implemented,
- Lift - application of a mathematical expressions.

3.2 MonetDB Assembly Language

MonetDB kernel language is named MAL(MonetDB Assembly Language). It has been designed for speed of parsing, ease of analysis, and ease of target compilation by query compiler according to MonetDB documentation. MAL instructions are interpreted by the engine and are considered a "specification of intended computation and data flow behavior" and its translation and representation depends on each specific use case.

In order to better understand MonetDB internals, there is the need to know more about the way information is stored. Relational tables are represented using vertical fragmentation, storing each column in a separate (object identifier (OID), value) table, called a binary association table(BAT). The storage relies on a low-level relational algebra called the BAT algebra, which takes BAT's and scalar values as input and the complete result is always stored in (intermediate) BAT's. The result of an SQL query is a collection of BAT's. But what is a BAT ? BAT is implemented as an ordinary C array and there are object identifiers(OID's) that map to the index in the array.

Furthermore, MonetDB kernel is an abstract machine, programmed in the MonetDB Assembly Language (MAL), where each relational algebra operator corresponds to a MAL instruction, with zero degrees of freedom, and each BAT algebra operator maps to a simple MAL instruction.

MonetDB provides several tools to analyse MAL instructions, such as Stethoscope (lately replaced by Pystethoscope) that allows to get access to the query plans and also the MAL instructions for the query execution.

In the tables at the appendix are represented and explained all the major MAL operators. This information was very important in order to make the comparison with *LAQ*. Only understanding the operations and it's results was possible to make a good comparison and parallelism between MAL and LAQ.

MonetDB Client Given that the MonetDB server is already running, the way to interact with it is through the application provided named Mclient. There are a range of input languages that this application can take as input, in our case we want SQL.

4 Web interface

The goal of creating a visual interface for the *LAQ* engine is to simplify the query creation by users that know how to specify queries using type diagrams as described by Afonso et al.. Taking as example the SQL query :

```sql
select e country , e branch, sum (j salary )
    from empl , jobs
    where j code = e job
    group by e country , e branch
    order by e country;
```

The query types and structure can be represented over the type diagram shown in fig 1.

$$\begin{tikzcd}
 & D & & \\
1 & \#j \arrow[l, "j^{salary}"'] \arrow[u, "j_{desc}"] \arrow[d, "j_{code}"'] & N & \\
 & K & \#e \arrow[l, "e_{job}"'] \arrow[u, "e_{name}"'] \arrow[r, "e_{branch}"] \arrow[d, "e_{country}"] & B \\
 & & C &
\end{tikzcd}$$

Fig. 1. Type Diagram.

Based on this representation of *SQL*, queries can be represented by using measures and dimensions. After some research and experimenting with dot, we came to the conclusion that it wasn't flexible enough and to achieve our goals it would be needed a more high level library. The chosen library was *react-graph-vis*, a library for the front end framework *ReactJS*. This library provides a more high level approach to *dot* format, it uses a similar syntax but makes the interactivity part more easy to create and handle. Since there is the need to have an appealing and reactive tool this seemed to be the better option for the development of a solution.

After having sorted the user interface technologies, it was needed a way to receive the input data, analyse it and send it to the *LAQ* engine. To solve this problem, their must be added a new layer to the application, a back end solution. For the case it was chosen *nodeJS* because it provides a robust and scalable structure as well it was a technology on which we were familiarized. Also Haskell was seen has a possible tool to make type checking, but the developed work didn't made to that phase. So the defined application architecture is defined in fig. 2.

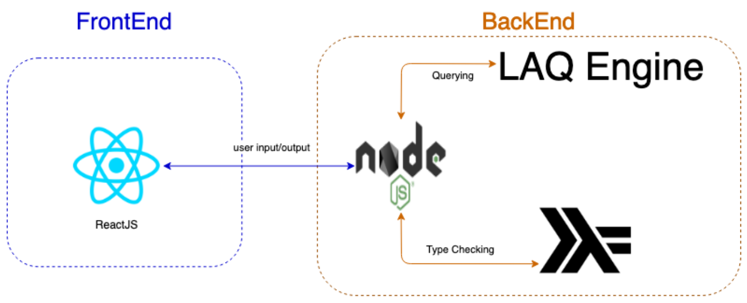


Fig. 2. Web application architecture.

Firstly there was needed to extend the initial type diagram (TD) representation, for that it was added :

- Entities must be added with prefix # ;
- Primary keys attributes must be added with postfix "_pk" ;
- Foreign keys attributes must be added with postfix "_fk" ;

Having this, it was possible to start the development and thinking of a way to give semantics to the visual graph. For that we used key value structures for each entity/table and also for dimensions and measures.Taking as example the type diagram in the fig. 1 the output format that sent to the backend is a *JSON* object as seen in the next code section :

```
tables : [{"nodes":[{"id":0,"label":"#j","color":"#e04141",

"attribute":"edges":[{"from":"0","to":"2","label":"j.desc"},

{"from":"0","to":"7","label":"j.salary"},{"from":"0","to":"4","label":"j.code"},
{"from":"1","to":"4","label":"e.job"},{"from":"1","to":"3","label":"e.name"},

{"from":"1","to":"5","label":"e.country"},{"from":"1","to":"6","label":"e.branch"}]}]

dimensions : [{"origin":"#j","destiny":"D","label":"j.desc"},

{"origin":"#j","destiny":"K","label":"j.code"},

{"origin":"#e","destiny":"K","label":"e.job"},

{"origin":"#e","destiny":"N","label":"e.name"},

{"origin":"#e","destiny":"C","label":"e.country"},

{"origin":"#e","destiny":"B","label":"e.branch"}]

measures : [{"origin":"#j","destiny":"1","label":"j.salary"}]

pk:[]

fk:[]
```

This data was regarded as necessary considering the parameters that the *LAQ* engine needs and also considering the input of the operations .

In order to confirm that the stored data that is sent to the back end is enough and complete, there was created an alloy model. There was the attempt to answer the questions, what must we represent ? The data, the operations (relations) ? The data structures where the data and operations will be stored ? For that we built an abstract representation of type diagrams and then created a concrete example (the same as shown in fig. 1) .

It was created a first more abstract and general model as shown in appendix B, this model has the definitions of Node and Arrow as well as the types of the information flowing in the front end application. Then it was made a specific model to the example from fig. 5 specifying facts to replicate the system beahaviour.

This work confirmed that the initial representation was correct and enough according to the developed model.There was the attempt to use alloy as a type checker instead of the original idea of Haskell but the idea was never accomplished.

The final interface is fully functional and tested allowing to create type diagrams, as well as export and import it.There is a implemented back end but it is only used to the importation and exportation of diagrams.Taking as example the type diagram in fig. 1, it is possible to identify the dimensions :

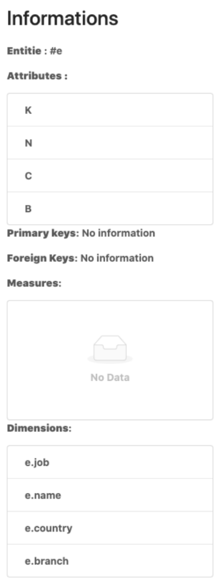
– branch
– name

- country
- job
- desc
- code

And also the measure :

- salary

In the developed prototype it is possible to have access to that information by clicking the nodes/entities as seen in fig. 3 and fig. 4 .



Informations

Entitie : #e

Attributes :

K

N

C

B

Primary keys: No information

Foreign Keys: No information

Measures:

No Data

Dimensions:

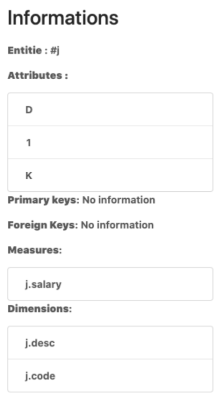
e.job

e.name

e.country

e.branch

Fig. 3. #e.



Informations

Entitie : #j

Attributes :

D

1

K

Primary keys: No information

Foreign Keys: No information

Measures:

j.salary

Dimensions:

j.desc

j.code

Fig. 4. #j.

It is also possible to view the type diagram represented in 1, change it's layout, edit nodes (entities) and arrows (relations) names as is shown in 5.

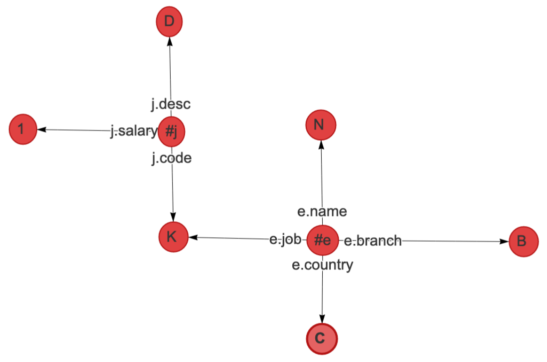


Fig. 5. Application Type Diagram.

Currently, the following functionalities operational:

- Create, edit or delete nodes(entities) ;
- Create, edit or delete arrows(relations) ;
- Consult nodes informations as seen in fig. 3 and fig. 4;
- Import and export diagrams in a JSON file ;

5 Establishing a relation

In this section is presented the analysis work done in the comparison of the two approaches.

5.1 Comparison between MAL and LAQ

In this section is presented the analysis and comparison between MAL and LAQ query plans, algebra operation execution times and well as a parallelism between the operators.

In order to compare the two approaches and show their similarities there was built the graph in fig 6. In this graph the equivalent operations are highlighted in colors. The LAQ query plan was taken from the work of Afonso et al. on the TPC-H query 3 and the MAL plan was taken using MonetDB client (it is also available using pystethoscope in MonetDB 11.37.7). After having the results side by side it can be said that the operations have correspondences, having as main difference the structure in which they are operating, one works with BAT(MonetDB) and the other with matrices (LAQ).

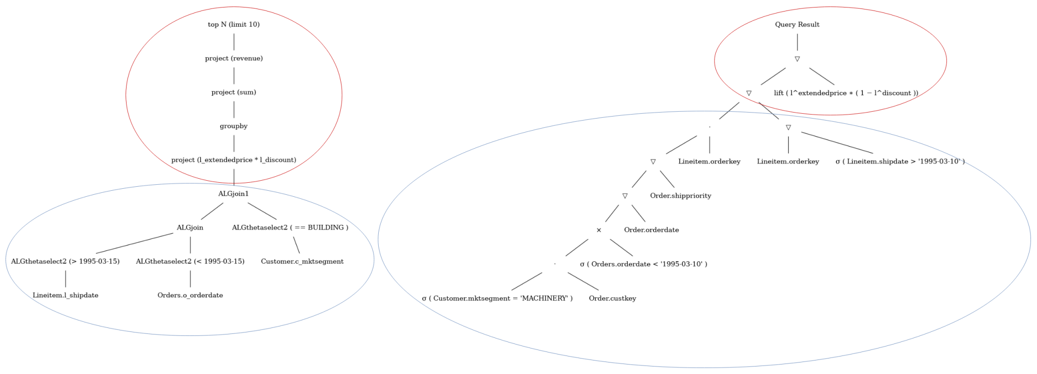

top N (limit 10)
project (revenue)
project (sum)
groupby
project (l_extendedprice * l_discount)
ALGjoin1
ALGjoin
ALGthetaselect2 (== BUILDING)
ALGthetaselect2 (> 1995-03-15)
ALGthetaselect2 (< 1995-03-15)
Customer.c_mktsegment
Lineitem.l_shipdate
Orders.o_orderdate
Query Result
▽
▽
lift (l^extendedprice ∗ (1 − l^discount))
·
▽
▽
Lineitem.orderkey
Lineitem.orderkey
σ (Lineitem.shipdate > '1995-03-10')
▽
Order.shippriority
×
Order.orderdate
·
σ (Orders.orderdate < '1995-03-10')
σ (Customer.mktsegment = 'MACHINERY')
Order.custkey

Fig. 6. Comparison graph.

There are some similarities, namely :

– Booth plans, LAQ and MAL, are look-a-like;
– MAL ALGjoin(and ALGjoin1) are equivalenn to LAQ Khatri-rao, Dot and Hadamard-Schur product ;

There are also some main differences :

– MAL joins results on Binary Association Tables (BATs);
– LAQ joins results on Matrices;

Since both approaches are from columnar databases, type diagrams can be used to approach and modulate queries. The main difference relies on the algebra on which the type diagrams will be translated. By one side LAQ uses linear algebra as a mean to encode queries as on the other side, MonetDB uses it's own algebra - MAL. The first has some fundamental properties, know and defined over it's operators. The second is not studied and we don't have enough details and information about it's implementation. Since the operators have some similarities it could be that MAL operators and BATs are a different approach to linear algebra, but in this work that hypothesis was not fully explored.

6 Conclusions and Future Work

Given the dimension of the project there was only time to achieve some milestones. In terms of the front end the objectives were fulfilled, it was created a application that allowed to draw type diagrams that translate SQL queries.

In terms of the back end there was many work that was left behind. Namely the connection between the backend server and the *LAQ* engine, the parser for the data from the front end, the integration with haskell and the conclusion of the SQL parser from the *LAQ* engine.

Furthermore, there was a deep study of *monetDB* internals, algebra and operations that allowed to make structural comparisons with the *LAQ* engine and algebra .

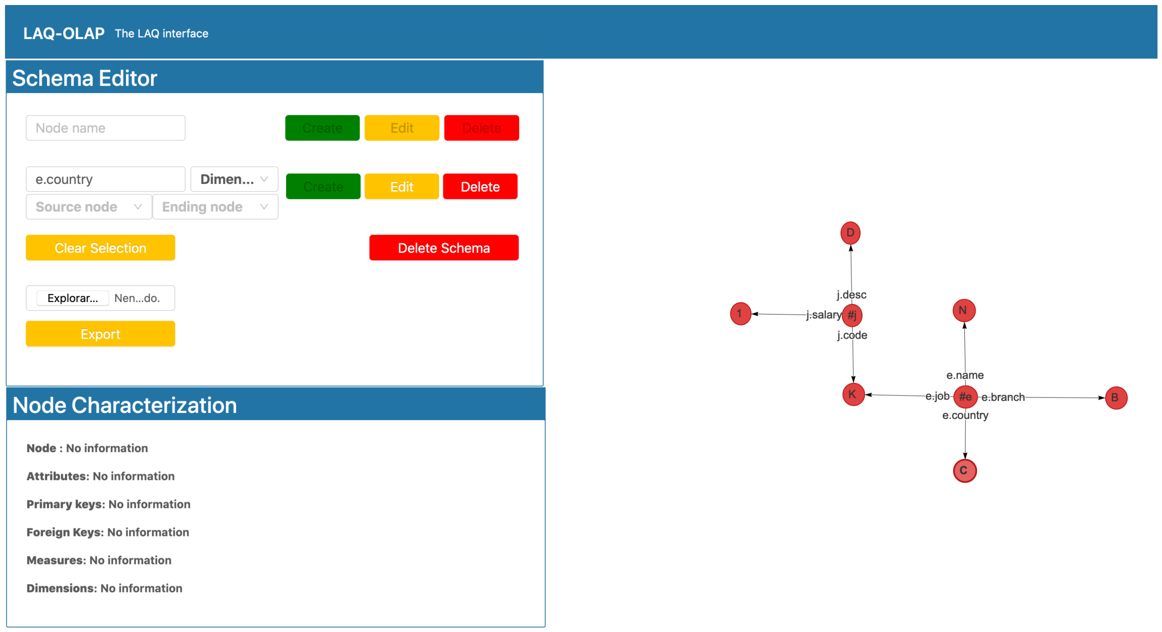
In terms of structure and used operators there is a clear parallelism and similarity as referred previously in section 5, this factor leaves open the possibility of integration on this way of thinking and typing queries into MonetDB using our developed application, of course with the necessity of adding a interpreter to convert our json objects into some format that MonetDB and specifically MAL could interpret.

Since the LAQ engine has some limitation when it comes to SQL conversion, identified by Afonso in his Master Thesis, namely:

- any attribute in the SELECT statement must be in the GROUP BY or under an aggregation function, otherwise it can't be converted into LAQ;
- Type diagrams with circular pattern can't be converted into LAQ;

Adding all this factors an approach to bring type diagrams way of thinking into MonetDB could be a plus to an engine that it is already super optimized, and could benefit from a visual interface with a typed and well defined way of thinking and modulate queries.

A Application overview



Node : No information

Attributes: No information

Primary keys: No information

Foreign Keys: No information

Measures: No information

Dimensions: No information

Fig. 7. Application overview.

B Metamodels - Alloy General Model

```
abstract sig Node {
   name : one Label,
   color : one Color
}

abstract sig Arrow {
   label : one Label,
   source : one Node,
   target : one Node
}

fact{
   name.~name in iden -- injective
   label.~label in iden -- injective
   no name.~label
}
```

```
sig Entity extends Node {}

sig Type extends Node {}

sig Var extends Type {}

sig CType extends Type {}

sig Prop {}

sig Dim extends Arrow {}

sig Meas extends Arrow {}

sig Label, Color {}
```

C Metamodels - Alloy Employees example

```
open model

// Definition of Entities
one sig j,e extends Entity {}

// Definition of diagram variables (attributes)
one sig B,N,C,D,K extends Var{}

one sig One extends CType{}

// Definition of diagram relations

// Entity Jobs
one sig j_code extends Dim {}

one sig j_desc extends Dim {}

one sig j_salary extends Meas {}

// Entity Employees
one sig e_job  extends Dim {}

one sig e_name extends Dim {}

one sig e_branch extends Dim {}
one sig e_country extends Dim {}

// Facts to model the example
fact {
        // all arrows that have as origin j
        source.j   = j_code + j_desc + j_salary

        // all the target for arrows that leave j
        target.D   = j_desc
        target.One = j_salary
        target.K          = j_code + e_job // in this case we must include the one that com

        // all arrows that have has origin e
        source.e   = e_job + e_name + e_branch + e_country

        // all the target for arrows that leave e (e_job is already done)
        target.N          = e_name
        target.B         = e_branch
        target.C          = e_country
}
```

D MAL

D.1 Operators

MAL	Address	Comment
groupby	ALGgroupby	Produces a new BAT with groups indentified by the head column. (The result contains tail times the head value, ie the tail contains the result group sizes.)
find	ALGfind	Returns the index position of a value. If no such BUN exists return OID-nil.
fetch	ALGfetchoid	Returns the value of the BUN at x-th position with 0 ¡= x ¡ b.count
project	ALGprojecttail	Fill the tail with a constant
projection	ALGprojection	Project left input onto right input.
projection2	ALGprojection2	Project left input onto right inputs which should be consecutive.

D.2 BAT copying

MAL	Address	Comment
copy	ALGcopy	Returns physical copy of a BAT.
exist	ALGexist	Returns whether ’val’ occurs in b.

D.3 Selecting

The range selections are targeted at the tail of the BAT.

MAL	Address	Comment
select	ALGselect1	Select all head values for which the tail value is in range. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value.
select	ALGselect2	Select all head values of the first input BAT for which the tail value is in range and for which the head value occurs in the tail of the second input BAT. The first input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value.
select	ALGselect1nil	With unknown set, each nil != nil
select	ALGselect2nil	With unknown set, each nil != nil
selectNotNil	ALGselectNotNil	Select all not-nil values.
thetaselect	ALGthetaselect1	Select all head values for which the tail value obeys the relation value OP VAL. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.
thetaselect	ALGthetaselect2	Select all head values of the first input BAT for which the tail value obeys the relation value OP VAL and for which the head value occurs in the tail of the second input BAT. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.

D.4 Sort

MAL	Address	Comment
sort	ALGsort11	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort12	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.

Continued on next page

Continued from previous page

MAL	Address	Comment
sort	ALGsort13	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort21	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort22	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort23	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort31	Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort32	Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.
sort	ALGsort33	Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.

D.5 Unique

MAL	Address	Comment
unique	ALGunique2	Select all unique values from the tail of the first input. Input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value. The second input BAT is a list of candidates.
unique	ALGunique1	Select all unique values from the tail of the input. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value.

D.6 Join operations

MAL	Address	Comment
crossproduct	ALGcrossproduct2	Returns 2 columns with all BUNs, consisting of the head-oids from 'left' and 'right' for which there are BUNs in 'left' and 'right' with equal tails

Crossproduct

MAL	Address	Comment
join	ALGjoin	Join
join	ALGjoin1	Join; only produce left output
leftjoin	ALGleftjoin	Left join with candidate lists
leftjoin	ALGleftjoin1	Left join with candidate lists; only produce left output
outerjoin	ALGouterjoin	Left outer join with candidate lists
semijoin	ALGsemijoin	Semi join with candidate lists
thetajoin	ALGthetajoin	Theta join with candidate lists
bandjoin	ALGbandjoin	Band join: values in l and r match if r - c1 ¡[=] l ¡[=] r + c2
rangejoin	ALGrangejoin	Range join: values in l and r1/r2 match if r1 ¡[=] l ¡[=] r2
difference	ALGdifference	Difference of l and r with candidate lists
intersect	ALGintersect	Intersection of l and r with candidate lists (i.e. half of semi-join)

Joining

D.7 Projection operations

MAL	Address	Comment
firstn	ALGfirstn	Calculate first N values of B
reuse	ALGreuse	Reuse a temporary BAT if you can. Otherwise, allocate enough storage to accept result of an operation (not involving the heap)
slice	ALGslice\$_{\text{oid}}$	Return the slice based on head oid x till y (exclusive).
slice	ALGslice	Return the slice with the BUNs at position x till y
slice	ALGslice\$_{\text{int}}$	Return the slice with the BUNs at position x till y
slice	ALGslice\$_{\text{lng}}$	Return the slice with the BUNs at position x till y
subslice	ALGsubslice\$_{\text{lng}}$	Return the oids of the slice with the BUNs at position x till y

D.8 Common BAT Aggregates

These operations examine a BAT, and compute some simple aggregate result over it.

MAL	Address	Comment
count	ALGcount\$_{\text{bat}}$	Return the current size (in number of elements) in a BAT.

Continued on next page

Continued from previous page

MAL	Address	Comment
count	ALGcount\\$_{\text{nil}}$	Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{\text{nils}}$==TRUE.
count	ALGcountCND\\$_{\text{bat}}$	Return the current size (in number of elements) in a BAT.
count	ALGcountCND\\$_{\text{nil}}$	Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{\text{nils}}$==TRUE.
count$_{\text{nonil}}$	ALGcount$_{\text{no}}$\\$_{\text{nil}}$	Return the number of elements currently in a BAT ignoring BUNs with nil-tail
count$_{\text{nonil}}$	ALGcountCND\\$_{\text{no}}$\\$_{\text{nil}}$	Return the number of elements currently in a BAT ignoring BUNs with nil-tail

D.9 Default Min and Max

Implementations a generic Min and Max routines get declared first. The @emph{min()} and @emph{max()} routines below catch any tail-type. The type-specific routines defined later are faster, and will override these any implementations.

cardinality - ALGcard **min** - ALGminany, ALGminany$_{\text{skipnil}}$ **max** - ALGmaxany, ALGmaxany$_{\text{skipnil}}$

PATTERN **avg** - CMDcalcavg

D.10 Standard deviation

The standard deviation of a set is the square root of its variance. The variance is the sum of squares of the deviation of each value in the set from the mean (average) value, divided by the population of the set.

stdeb - ALGstdev **stdevp** - ALGstdevp **variance** - ALGvariance **variancep** - ALGvariancep **covariance** - ALGcovariance **covariancep** - ALGcovariancep **corr** - ALGcorr

Bibliography

João M. Afonso, Gabriel D. Fernandes, João P. Fernandes, Filipe Oliveira, Bruno M. Ribeiro, Rogério Pontes, José N. Oliveira, and Alberto J. Proença. Typed linear algebra for efficient analytical querying. *CoRR*, abs/1809.00641, 2018. URL `http://arxiv.org/abs/1809.00641`.

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

Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012. URL `http://sites.computer.org/debull/A12mar/monetdb.pdf`.

Raghunath Othayoth Nambiar, Matthew Lanken, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction processing performance council (TPC): twenty years later - A look back, a look ahead. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, volume 5895 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2009. https://doi.org/10.1007/978-3-642-10424-4_1. URL `https://doi.org/10.1007/978-3-642-10424-4_1`.

J.N. Oliveira and Hugo Daniel Macedo. The data cube as a typed linear algebra operator. In Tiark Rompf and Alexander Alexandrov, editors, *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 6:1–6:11. ACM, 2017. https://doi.org/10.1145/3122831.3122834. URL `https://doi.org/10.1145/3122831.3122834`.