Department of Informatics, University of Zürich

MSc Basic Module

Implementing Self Multiplication Inside MonetDB

Jonathan Stahl

Matrikelnummer: 13-935-440

Email: jonathan.stahl@uzh.ch

23.06.2019

supervised by Prof. Dr. Micheal Böhlen and Oksana Dolmatova

University of Zurich[UZH]

Department of Informatics

1 Introduction

Traditional database systems store data in tuple form, or in other words 'row by row'. But if the main application will be data science / business intelligence, row-store can be unfavorable. This can be remedied by a column-store database system. In data science a lot of vector based operations on columns have to be performed. Therefore, the column-store approach, with the data already in a vector for each attribute, is very interesting. But traditionally in data science the data is exported from the DBMS into tools like R or MATLAB. The processing is then conducted there. Performing linear algebra operations, such as matrix operations, directly in the DBMS could save time and memory by reusing its internal data structures and optimizer [1].

In the scope of this Master Basic Module, the necessary knowledge should be acquired to realize the Master Project: *Matrix Operations with Gathering in MonetDB*. For the project I will collaborate with Alphonse Mariyagnanaseelan and Timo Surbeck.

1.1 MonetDB

MonetDB is an open-source column store DBMS, which is in development since 1993 at the Centrum Wiskunde & Informatica (CWI). With its data stored in so called Binary Association Tables (BAT), it pioneered the column store technology. Each BAT of a table stores all values of one attribute in an indexed array like data structure. To reassemble the tuples of a table, each value in a BAT has an Object-Identifier (OID) [2].

The architecture of MonetDB consists of three layers. In the top layer the user finds the access point. By entering a SQL query the whole machinery is set in motion. First, the query is translated into a *symbol tree*. The symbol tree is a representation of the query. Each node refers to a SQL keyword and is linked with instructions for the second stage. In this stage the *relational tree* is built. A relational tree node represents a relation, which can be found in the database, or is the result of an operation. Further, a node consists of one or two child nodes, the operator type and also list of of expressions used for the operation. In this stage also the first optimizations are performed. The strategic optimizer can rearrange the tree according to heuristics and rules. The third stage is responsible for the creation of a *statement tree*. This tree is specific for column store DBMS. Each node is a statement or a statement list and refers to one or more columns. This tree allows to conveniently define operations on columns. In the final stage of the top layer, the statement tree is transformed into MonetDB Assembly Language (MAL). MAL is a language, which operates on BATs. Here is where the middle layer starts. A MAL-Plan is produced with further optimizations. On the lowest layer the MAL instructions are carried out directly on the BATs, according to the MAL-Plan [1] [2].

1.2 Problem Definition

The purpose of this MSc Basic Module was to gain enough knowledge about the internals of MonetDB to approach the Master Project. For that reason, the task was to implement self multiplication of a one-numeric-attribute relation. This new operation should take a relation `r` with one numeric attribute as an input, and a relation with one numeric attribute consisting of all values of `r` multiplied with it self as an output. An example can be found in table 1.1. It is a simple operation, but requires to understand, use and extend the structures of the top layer of MonetDB until the transformation to MAL.

r	*r'*
A	**A**
3	9
9	81
2	4

Table 1.1: Applying self multiplication to table *r*

2 Design & Implementation

The design and implementation was done in a top down approach. At first the SQL parser, which is the entry point for a query to be executed, was extended. From there the development followed the sequence in which MonetDB processes a query. Step by step the symbol tree, relational tree, statement tree and finally the MAL Plan was extended. As an example, and sometimes placeholder to test certain parts, the native cross join implementation was used. For debugging purposes, and to better understand the query processing sequence, the GNU Project Debugger (GDB) came to action.

As a starting point, MonetDB on version 11.23.13 was used. During the implementation regularly commits were pushed to the GitHub repository which can be found here:
`https://github.com/jonixis/MonetDB`

2.1 Symbol Tree

The symbol tree is a tree representation of the SQL query entered by the user. The first milestone was to introduce a new SQL keyword for self multiplication of a one-column relation. The query in listing 1 with the new keyword `mul` should result in *r'* seen in table 1.1. MonetDB uses *Yet Another Compiler-Compiler* (YACC) to generate the SQL parser. All the configuration for the SQL parser is handled in the YACC file *sql_parser.y*. To introduce the

new keyword, the code found in listing 2 was added. On the first line it is defined, that the keyword `mul` has to be followed by a table name. Next, the table reference (accessed with `$2`) is added to a list, which is then transformed into a symbol tree node. With the token `SQL_MUL` the operation type is set. In the file *rel_select.c* the case `SQL_MUL` is handled, and the symbol tree node processing continues.

```
SELECT * FROM mul r;
```

Listing 1: Example query for self multiplication

```
1  |  MUL table_ref
2  {  dlist *l = L();
3      append_symbol(l, $2);
4      $$ = _symbol_create_list( SQL_MUL, l); }
```

Listing 2: Introduce `mul` keyword to parser

2.2 Relational Tree

The symbol tree contains table names and attribute names, but those were not yet checked on their existence. During the construction of the relational tree all values are verified against the actual database. This all takes place in *rel_select.c*. If the verification is a success a new relational tree node is constructed by calling a function defined in *rel_rel.c*. The necessary code can be found in listing 3. In the case of our example query the relational tree is very simple, as seen in figure 2.1, where $\otimes$ is the operator for self multiplication.

```
1  sql_rel *rel = rel_create(sa);
2
3  rel->l = l;
4  rel->r = NULL;
5  rel->op = mul;
6  rel->exps = NULL;
7  rel->card = l->card;
8  rel->nrcols = l->nrcols;
```

Listing 3: Construct a relational tree node for self multiplication

Figure 2.1: Relational tree of the query in listing 1

2.3 Statement Tree

In a next step the relational tree is transformed to a statement tree. The transformation takes place in *rel_bin.c*. In listing 4 the corresponding code can be found. Nodes in a statement tree represent columns or a list of columns. Because the self multiplication operation needs only one relation, only the left child node is used. On line 1 the function call `subrel_bin` recursively transforms the left child nodes into statements. On line 3 to 5 the first column of the relation is extracted. On line 6 the function shown in listing 5 gets called. It returns a statement node of the type `st_multiplication` with one column.

```
1       stmt *left = subrel_bin(sql, rel->l, refs);
2       assert(left);
3       node *n = left->op4.lval->h;
4       stmt *c = n->data;
5       stmt *l = column(sql->sa, c);
6       stmt *multiplication = stmt_multiplication(sql->sa, l);
```

Listing 4: Transform relational tree node to statement tree node

```
1       stmt *
2       stmt_multiplication(sql_allocator *sa, stmt *op1) {
3           stmt *s = stmt_create(sa, st_multiplication);
4
5           s->op1 = op1;
6           s->op2 = NULL;
7           s->key = 0;
8           s->nrcols = 1; }
```

Listing 5: Construct a statement tree node for self multiplication

2.4 MAL Plan

In the last step the statement tree is translated to a MAL plan. In the file *sql_gencode.c* the function `_dumpstmt` contains a big switch case, where a case was added for the statement nodes with the new type `st_multiplication`. The MAL plan defines which BAT operations are performed. The code for this can be seen in listing 6. Again, the tree is translated recursively as seen on line 1, where `_dumpstmt` is called. On line 3 the explicit BAT operation type `"*"` is passed to a new instruction. It follows the appending of the relations single attribute `l`. In order to achieve an element wise self multiplication of this attribute it is added twice. The internal BAT operation is called `batcalc.*` and was already part of MonetDB. For example it is used in the standard query in listing 7. This can be verified by executing the query with `EXPLAIN` prepended.

```
1       l = _dumpstmt(sql, mb, s->op1);
2       assert(l >= 0);
3       q = newStmt(mb, batcalcRef, "*");
4       q = pushArgument(mb, q, l);
5       q = pushArgument(mb, q, l);
```

Listing 6: Transform statement tree node to MAL plan

```
SELECT (A*A) FROM r;
```

Listing 7: Example query for self multiplication

3 Conclusion & Discussion

With relatively few lines of code it was possible to introduce a new SQL keyword `mul`, which can be used to perform self multiplication on a single-attribute relation. Although the implementation did not need many lines of code, this Basic Module gave a thorough introduction into the architecture and mechanisms of MonetDB. By following the path of the standard cross join query, the high extendibility, especially for matrix operations, revealed itself.

Bibliography

[1] Alphonse Mariyagnanaseelan, *Optimization of Mixed Queries inMonetDB System*, https://www.ifi.uzh.ch/dam/jcr:3fe0b0d1-fbc2-4934-a447-60d2d99c09af/Bachelorarbeit.pdf, accessed 2019-06-18, 2019.

[2] Database Architectures Research Group (CWI), *MonetDB*, https://www.monetdb.org, accessed 2019-06-18.