Universität Zürich
Institut für Informatik

HS 2016
Dr. Michael Böhlen

Student Name: Dzmitry Katsiuba
Matriculation Number: 14-705-354

Informatik-Vertiefung
Datenbanken
MonetDB

About MonetDB

MonetDB is an open source column-oriented database management system developed at the Centrum Wiskunde & Informatica (CWI) in the Netherlands. It was designed to provide high performance on complex queries against large databases, such as combining tables with hundreds of columns and millions of rows. MonetDB distribute several server and client programs. For solving my task i needed following programs:

1. monetdbd - The MonetDB Database Server daemon.Once started it runs in the background and facilitates, amongst others, management of local (and remote) mserver5's.
2. mserver5 - The MonetDB server, version 5. It runs as a console program. Client programs (except monetdb) connect to this server process.
3. mclient - A command-line program to interactively communicate to a running mserver5 process.

Parsing the query

After entering a query into the mclient, the query is analysed in order to detect the predefined keywords (tokens). The tokens help the YACC parser generator to generate a shift-reduce parser. The input to Yacc is a grammar (sql_parser.y) with snippets of C++ code ("actions") attached to its rules.

As soon as the rule is recognised, the parser calls the code snippets associated with this rule. This snippets help to build a symbol tree, which has following elements: selectNode, symbol, symbdata, dlist, dnode. The hierarchy and organisation of elements is shown in figure 1.

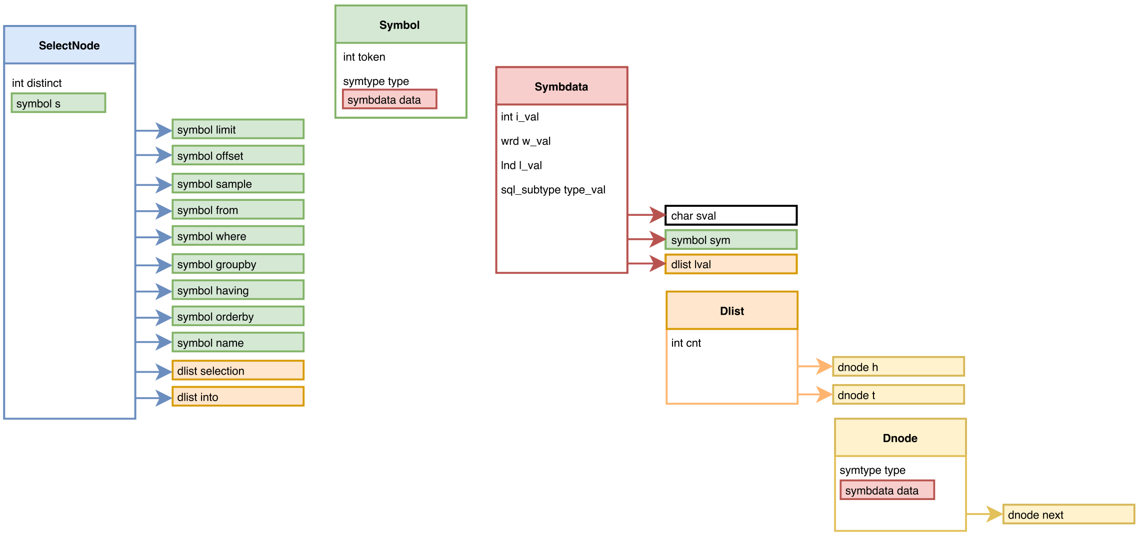

Figure 1: Symbol tree elements

Building a symbol tree

SELECT * FROM SCALAR_SQR(table1);

To make parsing through the example query possible, it was necessary to define a new token, add additional rule to existing grammar, and write snippets of code for this rule. After parsing and executing the corresponding code the symbol tree shown in figure 2 is produced:

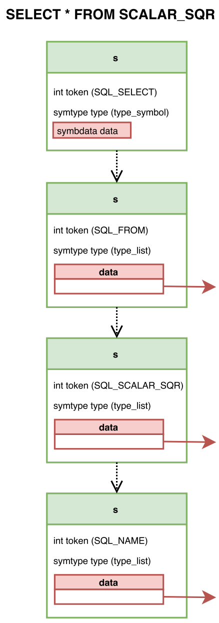

Figure 2: Symbol tree

Building a relation tree

In order to make first optimisation on the tree possible, after the symbol tree is build, it is converted to a relation tree. This optimisation looks on the tables, which will be involved in producing the results, and optimise the operations on this tables. At this point all symbols will be transformed to relations, depending on the tokens in them. Tokens from symbols will not just be taken for the relations, groups of operations are formed instead. This groups allow better analysing of query and finding optimisation options. Relations are connected with each other or base elements (with help of 2 "children")and have the structure as shown in figure 3. Child connections help to build a relation tree.

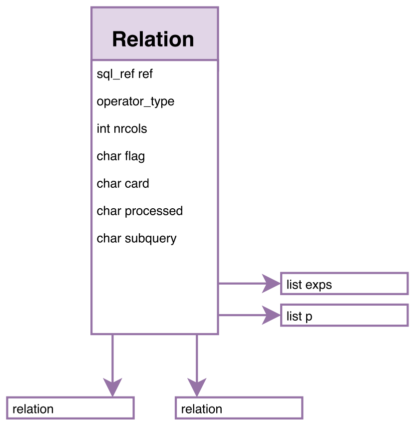

Figure 3: Relation tree element

To implement the SCALAR_SQR function it was necessary to add a new function which helps to deal with the new operation and build an appropriate relation tree.

According to the task, the input for SCALAR_SQR function is one table with some numeric columns. After parcing the query the name of the table will be saved in one symbol (symbol with a table reference). That means, after symbol-relation transformation is done our relation tree will also have only one child.

```
rel
==================================================
project
| sqr op
| | table(sys.table1) [ table1.zahl1 ] COUNT
| ) [ ]
) [ table1.zahl1 ]
```

The resulting relation tree from SCALAR_SQR symbol tree looks like in Figure 4:

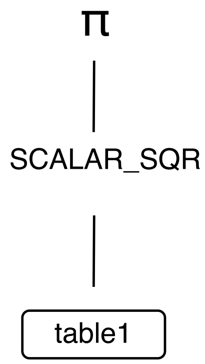

Figure 4: Relation/Query tree

Building a statement tree

Statement tree is a special property of MonetDB. It is an intermediate step between relation tree and low-level execution plan (MAL-Plan). Compared to relation tree, where we have an overview of table operations, statement tree describes the column operations for every point of relation tree. For SCALAR_SQR function i could use the existing column operations of Project and Table(Alias) statements (see figure 4). The SCALAR_SQR statement was to be implemented.

SCALAR_SQR statement takes the very left relation from the relation tree (because we have only one table as input). In the next step it takes all numeric columns from the input table and creates a list of aliases. Aliases make it also possible to change column names, if the user wants it. After that, for every column from that list a scalar_sqr function is called, which multiplies every column by the column itself. The information in columns is saved in form of Binary Association Tables (BATs). There is a module in MonetDB, that contains the commands and patterns to manage BATs. So i used the existing BAT-Algebra-commands (multiplication of 2 BATs) to become the results. The results are again collected in one list, which is later used in the projection statement.

The resulting statement tree from SCALAR_SQR relation tree looks like in Figure 5 (tree for a table with 3 numeric columns A, B, C):

No optimisation takes place at this level.

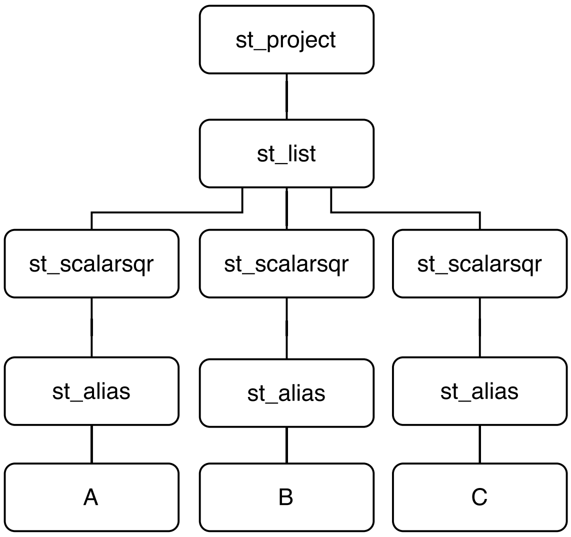

Figure 5: Relation/Statement tree

Example

sql> select * from table1;

zahl1
1
2
3

3 tuples (0.296ms)

sql> select * from SCALAR_SQR(table1);

zahl1
1
4
9

3 tuples (1.293ms)

Giving EXPLAIN before our query to the mclient shows us an execution plan of that query. In our example we can see, how the system reads the table, calculates and gives out the result. The highlighted line is the point of actual execution of SCALAR_SQR functionality.

```
mal
==================================================
function user.s6_1autoCommit=true():void;
X_34:void := querylog.define("select * from scalar_sqr(table1);","default_pipe",22);
X_18 := bat.new(nil:oid,nil:str);
X_26 := bat.append(X_18,"sys.table1");
X_21 := bat.new(nil:oid,nil:str);
X_28 := bat.append(X_21,"zahl1");
X_22 := bat.new(nil:oid,nil:str);
X_29 := bat.append(X_22,"int");
```

```
X_23 := bat.new(nil:oid,nil:int);
X_31 := bat.append(X_23,32);
X_25 := bat.new(nil:oid,nil:int);
X_33 := bat.append(X_25,0);
X_2 := sql.mvc();
C_3:bat[:oid,:oid] := sql.tid(X_2,"sys","table1");
X_6:bat[:oid,:int] := sql.bind(X_2,"sys","table1","zahl1",0);
(C_9,r1_9) := sql.bind(X_2,"sys","table1","zahl1",2);
X_12:bat[:oid,:int] := sql.bind(X_2,"sys","table1","zahl1",1);
X_14 := sql.delta(X_6,C_9,r1_9,X_12);
X_15 := algebra.leftfetchjoin(C_3,X_14);
X_16 := batcalc.*(X_15,X_15);
sql.resultSet(X_26,X_28,X_29,X_31,X_33,X_16);
end user.s6_1;
```