Department of Informatics, University of Zürich

MSc Basic Module

Gathering SQRT Calculation in MonetDB

Alphonse Mariyagnanaseelan

Matrikelnummer: 15-712-698

Email: `alphonse.mariyagnanaseelan@uzh.ch`

June 30, 2019

supervised by Prof. Dr. Michael Böhlen and Oksana Dolmatova

University of Zurich^{UZH}

Department of Informatics

1 Introduction

Scientific data, as well as data generated by businesses, frequently needs to be processed with complex mathematical operations during the data analysis. Traditional database management systems (DBMS) do not offer sophisticated operations beyond aggregation functions and simple vector operations that are performed on one relation. The current approach is to export the data to a statistics software, process the data there, and, if necessary, import the data back into the DBMS. This is not only prone to errors, but also does not use the benefits that a DBMS could provide, e.g. a high level descriptive language to formulate a query (SQL), type safety, constraint checks and the query optimizer.

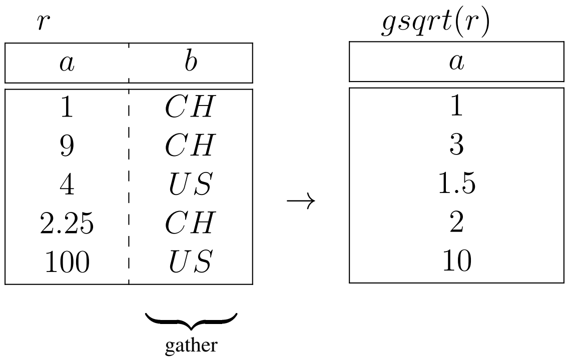
Many linear algebra operations, like the QR decomposition for example, can be formulated as vectorized algorithms. Such vectorized operations can be translated elegantly and executed with high performance in column-store DBMSs due to the way they store the data. The QR decomposition [2], matrix addition, as well as the optimization of matrix addition when combined with a selection [1] have already been implemented in MonetDB, an open-source column-store DBMS developed since 1993 at the Centrum Wiskunde & Informatica (CWI) [3]. As a next step we want to be able to perform matrix operations on groups of a relation, comparable to the grouping and aggregation operation we already know from SQL.

2 Problem Definition

The goal of this project is to implement the gathered `SQRT` operation in the column-store DBMS MonetDB, in order to acquire the necessary knowledge for the Master Project *Matrix Operations with Gathering in MonetDB*, which I will carry out with Jonathan Stahl and Timo Surbeck.

For some applications we want to be able to perform matrix operations on groups of a relation. For each group, a matrix operation must be performed, which yields a matrix as result, and thus may also consist of multiple tuples. This is in contrast to the traditional `GROUP BY` clause of SQL, where for each group one result tuple is produced. Due to that reason, we call this new operation *gathering*.

Considering relation r in Figure 2.1, we first gather by the attribute b and then we apply the



r

a	b
1	CH
9	CH
4	US
2.25	CH
100	US

$\rightarrow$

$gsqrt(r)$

a
1
3
1.5
2
10

Figure 2.1: Relations r and $gsqrt(r)$

square root to attribute a. While gathering, the physical ordering inside the groups must be preserved. Since this cannot be guaranteed on the level of relational operations, this must be implemented on a lower level. We can implement this as a function on Binary Association Tables (BAT), the data structure MonetDB uses to store columns of tables.

```
gsqrt(b1:bat[:dbl], b2:bat[:dbl]) :bat[:dbl]
```

Listing 2.1: Signature of `gsqrt` MAL function

The signature for the MonetDB Assembly Language (MAL) function is shown in Listing 2.1. The function takes two BATs with double-precision floating point numbers (`:dbl`) and returns one BAT. The first BAT `b1` denotes the how the tuples are split into groups and the second BAT `b2` contains the values on which the square root is applied. Further, we want to be able to use this function in the statement tree, which can be seen as a lower level version of the query tree which acts on attributes instead of relations.

3 Design & Implementation

We first implement the BAT algebra function `gsqrt` as defined in Listing 2.1. Implementing this function consists of two parts, one part involves the concrete implementation of the function in C, the other part is related to linking that function to a MAL instruction, so that it can be used like any other MAL function.

Then we extend the matrix definition in SQL with the `GATHER BY` clause and create a simple new matrix operation called `SQRT` that uses this clause. This matrix operation is implemented very similar to the matrix addition in [1], so we will not look at the details of the implementation in the symbol tree, relation tree and statement tree here.

```
SELECT b ! a
FROM r;
```

Listing 3.1: SQL syntax of `gsqrt` as operator

To test the BAT function, we use the exclamation mark (`!`) as an operator, where the left hand side is used as the grouping attribute and the right hand side is used as the value attribute, on which the square root is applied. We do this due to the fact that such an operator can be trivially called in the `SELECT` clause of an SQL query, as shown in Listing 3.1, without extending the three aforementioned trees.

3.1 Implementation of `gsqrt` C Function

For the implementation of the `gsqrt` function, the signature looks as shown in Listing 3.2. The arguments for the functions correspond to the arguments of the MAL function described in Listing 2.1.

```
BAT* BATcalcgsqrt(BAT *b1, BAT *b2)
```

Listing 3.2: Signature of C function (simplified)

First we sort the BAT `b1`, since sorting will naturally group the tuples according to the values in the given BAT. The signature of `BATsort` is shown in Listing 3.3. The function will sort BAT `b` considering order `o` and groups `g`. The sorted version of `b` is stored in BAT `sorted`, the Object Identifiers (OID) are stored in the BAT `order` and the Group Identifiers (GID) are stored in the BAT `groups`.

```
gdk_return BATsort(BAT **sorted, BAT **order, BAT **groups,
    BAT *b, BAT *o, BAT *g, int reverse, int stable)
```

Listing 3.3: Signature of `BATsort`

The implementation of the sorting of BAT `b1` and aligning of BAT `b2` for the gathering is shown in Listing 3.4. We do not need the actual sorted version of BAT `b1`, we only need the OIDs and the GIDs. The OIDs for the order are stored in `b1o`, the GIDs for the groups are stored in `b1g`. We use stable sort to preserve the physical order whenever possible. Next, the BAT `b2` with the values is aligned according to `b1o`.

```
BATsort(NULL, &b1o, &b1g, b1, NULL, NULL, 0, 1);
bno = BATproject(b1o, b2);
```

Listing 3.4: Implementation of sort and align

Since square root does not depend on other values of the group, the BAT `b1g` is of no use here. However, if it were needed, we could have referred to the GIDs at this point and performed certain operations group wise. If we wanted to gather by multiple attributes, we could use the OIDs (`b1o`) and GIDs (`b1g`) to further sub-sort the BATs.

Next we iterate over the Binary Units (BUN), as shown in Listing 3.5. BUNs are tuples of OIDs and values, in our case floating point numbers. We iterate over the BUNs of the BAT `bno`, which is a copy of the BAT `b2` with the BUNs ordered according to the groups. The square root is calculated for each BUN, if it's not nil, and the result is then inserted in the BAT `bn`.

```
for (i = start, k = start; k < end; i++, k++) {
    if (bno[i] == dbl_nil) {
        bn[k] = dbl_nil;
        nils++;
    } else {
        bn[k] = sqrt(bno[i]);
    }
}
```

Listing 3.5: Implementation of BUN loop (simplified)

While iterating over the BUNs we count the number of nils. The variable `nils` usually serves multiple purposes: It counts either the number of nils, or it is set to a special constant to indicate that some error happened (overflows, conversion errors, division by zero). In our implementation we do not perform error checks yet and simply return the number of nils.

Instead of writing the for-loop ourself we could have used the BAT iterator. We decided to follow the same coding style that has been used to implement the other operators for now.

3.2 Implementation of `gsqrt` MAL Instruction

To be able to use the `gsqrt` function with the MonetDB Assembly Language, we add a pattern with the signature of the MAL function, a function that connects the MAL function to our function shown in Section 3.1 and a comment.

```
pattern !(b1:bat[:dbl],b2:bat[:dbl]) :bat[:dbl]
address CMDbatGSQRTsignal
comment "Return sqrt(B2), grouped by B1, signal error on overflow";

pattern gsqrt(b1:bat[:dbl],b2:bat[:dbl]) :bat[:dbl]
address CMDbatGSQRTsignal
comment "Return sqrt(B2), grouped by B1, signal error on overflow";
```

Listing 3.6: Implementation of MAL Instruction

As mentioned before, we define the exclamation mark (`!`) as a function with exactly the same arguments as `gsqrt` and define the exclamation mark to be considered as an arithmetic operator in the parser, which allows us to use this MAL function as shown in Listing 3.1.

3.3 Implementation of SQRT Matrix Operation

We add a new operator `SQRT` which will perform a matrix operation, as shown in Listing 3.7. It makes use of the matrix reference, extended with a `GATHER BY` clause, which is defined to be optional to be compatible with other matrix operations (e.g. matrix addition).

```
SELECT *
FROM SQRT (r ON a GATHER BY b);
```

Listing 3.7: SQL syntax of `SQRT` matrix operation

To be able to use out MAL function `gsqrt` in the statement tree, we create a statement type called `st_matrixsqrt` and create a case for it in the file *sql_gencode.c*. There we call the operator as shown in Listing 3.8. Instead of `gsqrt` we could have used the exclamation mark (`!`) operator as well, which would lead to the same result.

```
l = _dumpstmt(sql, mb, s->op1);
r = _dumpstmt(sql, mb, s->op2);
q = newStmt(mb, batcalcRef, "gsqrt");
q = pushArgument(mb, q, r);
q = pushArgument(mb, q, l);
```

Listing 3.8: Use `gsqrt` in statement tree

As implemented right now, the `GATHER BY` clause is part of the matrix reference and is optional. That means, the clause may be specified even for operations that do not actually consider it, e.g. the matrix addition, in which case the clause is silently ignored. We might change this in the future, so that the `GATHER BY` clause would be part of the operator specification instead, for example as shown in Listing 3.9.

```
SELECT *
FROM SQRT (r ON a) GATHER BY b;
```

Listing 3.9: SQL syntax of `SQRT` matrix operation

4 Conclusion & Discussion

Creating a new MAL function that operates on BATs can be done in roughly two steps. The implementation is written in C, then the MAL signature for the function is written and made to point to the according implementation while considering input and output types. The MAL function can then be called from within a statement tree node. Especially creating a new operator for the `SELECT` clause is simple, since no other modification is needed, besides in the parser.

```
SELECT *
FROM SQRT (r ON a, b, c GATHER BY d);
```

Listing 4.1: SQL syntax of `SQRT` matrix operation, multiple attributes in application part

The implementation of the gathering function as done in this project is yet quite inefficient, as for example the grouping is calculated multiple times if the operation is applied on multiple attributes, e.g. as shown in Listing 4.1. Also, it is currently not possible to gather by multiple attributes. Those features are out of scope for this project, but will be necessary for the upcoming Master Project *Matrix Operations with Gathering in MonetDB*. There, the gathering should be rewritten to make it a more general operation, so that it can be reused for different matrix operations.

Bibliography

[1] A. Mariyagnanaseelan *Optimization of Mixed Queries in MonetDB System*, 2018.

[2] D. Katsiuba *QR Decomposition Integration in MonetDB System*, 2017.

[3] Database Architectures research group (CWI), MonetDB *https://www.monetdb.org*, last accessed June 2019.