University of Zurich^UZH

Department of Informatics

Noah Berni

Informatik Vertiefung Database Systems

April 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

1 Introduction

The main target of the project is to extend the PostgreSQL kernel by a new SQL operator. We should be able to multiply two relations with the new operator, by considering the relations as matrixes with the columns "r"(row), "c"(column) and "v"(value). At the end of the day the output relation should be the matrix multiplication of the two input relations. For example, out of relations Input1 and Input2, we should get the result relation Output. On the left side, the matrix that is represented by each relation is shown.

Table 1: Matrix form of Input1

5	6
2	8
3	5

Table 2: Input1

r	**c**	**v**
0	0	5
0	1	6
1	0	2
1	1	8
2	0	3
2	1	5

Table 3: Matrix form of Input2

3	6
1	2

Table 4: Input2

r	**c**	**v**
0	0	3
1	0	1
0	1	6
1	1	2

Table 5: Matrix form of Output

21	42
14	28
14	28

Table 6: Output

r	**c**	**v**
0	0	21
0	1	42
1	0	14
1	1	28
2	0	14
2	1	28

To do so, we need to edit the process between transmitting a query to the server and receiving a result from it, but first we need to know how PostgreSQL is constructed. PostgreSQL is built in three stages: The Parser, the Optimizer and the Executor. The Parser controls if the entered query has correct syntax and creates a query tree. This query tree is given as an input to the Optimizer. The optimizer creates possible ways (paths), which lead to the correct result the query is looking for and calculates the costs for all possible paths. The path with the lowest cost is selected and expanded into a detailed plan tree, which the executor can use. Finally, the executor iteratively steps through the received plan tree from top to bottom, execute the described plans and returns the result rows. In the following parts each section is described in more detail and applied to our matrix multiplication example.

2 Parser

Like already described in the introduction, the Parser checks if an entered query has a correct syntax and then it builds a query tree. Before it can start with that, the input query needs to be split up in tokens, which the parser can recognize. This step is done by the lexer. For every keyword and identifier of an object (e.g. relation, attribute) the lexer finds in the given query, a token is generated and handed in to the parser. The parser itself consists of a set of grammar rules with a corresponding action for every rule. A rule is a list of consecutive tokens and its action is activated if the list of tokens appears in the token list received from the lexer. For every subsequence of tokens from the token list delivered by the lexer, there need to be a rule in the parser, which is fired, when the corresponding sequence of tokens is recognized. If a specific sequence of the input tokens is not matched with any rule of the parser, there is a syntax error in the query. When a rule is triggered, the corresponding action creates a parse node, a data structure which stores the necessary information for use later in the process of evaluating the given query. Every node can have other nodes as children. Because of this a query tree is generated out of the created parse nodes and is delivered to the optimizer. Later in the optimizer this tree can be handled iteratively starting from the root node, which calls its child nodes and so on. The control whether the required relations exist or not, is inspected in a semantic check right after the Parser. We skip a detailed description of this analyzing phase, because we don't really need to do something with it. To better understand the concept of the parser, Listing 1, Listing 2 and Figure 1 show the individual steps for a simple example.

Listing 1: Entered query

```sql
SELECT r, c FROM Input1 WHERE v > 10
```

Listing 2: Generated token list (Lexer output)

```
[SELECT, r, ',', c, FROM, Input1, WHERE, v, >, 10]
```

$$\begin{tikzcd}
\pi_{r,c} \arrow[d, dash] \\
\sigma_{r>10} \arrow[d, dash] \\
\text{Input1}
\end{tikzcd}$$

Figure 1: Genrated query tree (Parser output)

Hence to let PostgreSQL know the syntax of the matrix-multiplication operation we need to extend the parser and add some new rules and keywords for it. The matrix-multiplication can be compared to a join, because it somehow takes two input relations, joins them, and return a new output relation. Out of this reason we don't need to build a completely new parse node and we can just extend the join parse nodes to store information for a matrix multiplication. The C datatype that represents a join in a query tree is `JoinExpr`. In the `JoinExpr` all the necessary information of a join are stored. In Listing 3 the most important attributes of `JoinExpr` are shown. The type of the join is stored in an enum called `jointype`. Possible values of it are `JOIN_INNER`, `JOIN_LEFT`, etc. The relations, which are joined, are stored in the fields `larg` and `rarg`. Optional join conditions are stored in the `quals` node. If an alias clause exists, it is stored in the `alias` node. This is the basis that forms a `JoinExpr`.

Listing 3: JoinExpr

```
typedef struct JoinExpr
{
   JoinType jointype;          /* type of join */
   Node *larg;                 /* left subtree */
   Node *rarg;                 /* right subtree */
   Node *quals;                /* qualifiers on join, if any */
   Alias *alias;               /* user-written alias clause, if any */
   (some other members)
} JoinExpr;
```

Now we need to extend this `JoinExpr`, so that it can also store the necessary information for matrix multiplication operation. At the first part we just need to store in `JoinExpr` the information that a matrix-multiplication is required by the query. This is first done by adding a new rule in the parser, which is activated when the matrix multiplication keyword `MMUL` appears and assigning `JOIN_MMUL` as `jointype` to the `JoinExpr`. To make this legitimate, we also need to add the type `JOIN_MMUL` to `JoinType` enum. Also we need to specify what information are needed to execute the matrix-multiplication. In the first hand it seems that no more information are needed to perform the join, but if we look a bit forward to the optimizer and the executor, it gets obvious that we also need to know the dimensions of the input tables in order to calculate the costs and to execute the algorithm later. The dimensions represent the number of rows and columns of the two input tables. It would have been also possible to calculate the dimensions through another `SELECT` statement, but that would have been out of scope, and probably also not that efficient. The dimensions are stored in `JoinExpr` as integer fields. So the new `JoinExpr` node with the added information is shown in Listing 4.

Listing 4: JoinExpr with matrix multiplication members

```
typedef struct JoinExpr
{
   JoinType jointype;          /* MMul-join is assigned here */
   Node *larg;                 /* left subtree */
   Node *rarg;                 /* right subtree */
   Node *quals;                /* qualifiers on join, null in our example */
   Alias *alias;               /* user-written alias clause */
   (some other members)
    // our added dimensions
   int rowLeft;                /* number of rows left subtree */
   int colLeft;                /* number of columns left subtree */
   int  rowRight;              /* number of rows right subtree */
   int colRight;               /* number of columns right subtree */
} JoinExpr;
```

Since we have defined how a matrix multiplication operator is stored in a query tree, we need to define the new parser rule to recognize this operator. To build the rule we first need to know how the final query looks like. Our query takes 2 tables as input and joins them with the keyword `MMUL`. The values of the dimensions are placed right after the relations and they are introduced with the keyword `DIMENSIONS`, both for the left and the right relation. At the end the alias clause is placed. In Listing 5 an example of such a query is shown.

Listing 5: Example input query for a matrix multiplication operation

```
SELECT * FROM (Input1 MMUL Input2) DIMENSIONS 3 2 DIMENSIONS 2 2 AS Output;
```

Out of the query we can build our new rules. The rule which is added for the matrix-multiplication is shown in Listing 6. The rule matches the `FROM` part of the query, where the `MMUL` keyword appears. It handles all the tokens after the `FROM` keyword and before the `AS` keyword. In our rule a `JoinExpr` node is built and the necessary values are assigned to it. To access tokens of the rule body the '$' sign is used. With $i we can access the i^{th} token of the rule. For example with the statement (`n->larg = $2`), the second token (the first `table_ref`) is assigned to the left argument of the `JoinExpr`. The head of the rule, that is finally returned is assigned to `$$`.

Listing 6: Parse rule for matrix multiplication

```
mmul_join:
'(' table_ref MMUL table_ref ')' DIMENSIONS Iconst Iconst DIMENSIONS Iconst
    Iconst
{
   /* letting join_type reduce to empty doesn't work */
   JoinExpr *n = makeNode(JoinExpr);
   (some other initializations)
   n->jointype = JOIN_MMUL;
   n->larg = $2;
   n->rarg = $4;
   n->quals = NULL;
   n->rowLeft = $7;
   n->colLeft = $8;
   n->rowRight = $10;
   n->colRight = $11;
   $$ = n;
 }
```

Summarized we extended the `JoinExpr` node with a new `joinype` that performs matrix multiplication and 4 dimension attributes. Also we added a new rule for the matrix multiplication join, to make the PostgreSQL parser accept our new query type along with the dimensions of input matrixes. Also we added an alias for the output at the end, just to give the whole query a nicer flair.
After applying these changes to PostgreSQL source code the Parser accepts inputs like Listing 5, but Postgresql throws an error during the optimization process. To prohibit this we also need to adapt the optimizer.

3 Optimizer

The main goal of the optimizer is to create an optimal execution plan for a given query. The execution plan is constructed in a plan tree, which is composed of plan nodes. A plan node describes the specific algorithm that can be used to evaluate a corresponding SQL operator. A plan node returns a number of tuples, thus a table, when it's executed. Therefore when we have the resulting plan tree, the root plan node is called and it is resolved iteratively, getting resulting tables from its children as input tables.
For a given query, there can exist more than one way to execute it, each with the same set of results. The optimizer task is to find the way which is the cheapest to evaluate the query. To do so it builds a set with all possible path trees, which are delivering the correct results. A path tree is similar to a plan tree, but compared to a plan tree it takes only the necessary information to calculate the costs and therefore to find the cheapest path, it omits the information that is only need for the execution process. To better compare

a `Plan` and a `Path` , the basic `Plan` node is shown in Listing 7 and the basic `Path` node is shown in Listing 8. Both can be applied to simple plan types like the sequential scan. We don't want to go deep in detail, but we can see, that the `Path` keeps less information than the `Plan`. The `Path` more or less only cares about the type of itself (in this example a sequential scan) and its cost to execute. It doesn't even care about the input relation, which is scanned. At the other side the `Plan` needs the information like the childtrees (`lefttree`, `righttree` = `NULL` in a scan) and the `qual` conditions, in which the selection condition are stored, to execute the query.

Listing 7: Plan node

```
typedef struct Plan
{
   NodeTag    type;

   Cost    startup_cost;      /* cost expended before fetching any tuples */
   Cost    total_cost;        /* total cost (assuming all tuples fetched) */

   double     plan_rows;      /* number of rows plan is expected to emit */
   int        plan_width;     /* average row width in bytes */

   List    *targetlist;       /* target list to be computed at this node */
   List    *qual;             /* implicitly-ANDed qual conditions */
   struct Plan *lefttree;     /* input plan tree(s) */
   struct Plan *righttree;
   List    *initPlan;         /* Init Plan nodes (un-correlated expr *
       subselects) */

   Bitmapset *extParam;
   Bitmapset *allParam;
} Plan;
```

Listing 8: Path node

```
typedef struct Path
{
   NodeTag    type;
   NodeTag    pathtype;          /* tag identifying scan/join method */
   RelOptInfo *parent;           /* the relation this path can build */
   ParamPathInfo *param_info;    /* parameterization info, or NULL if none */

   double     rows;              /* estimated number of result tuples */
   Cost    startup_cost;         /* cost expended before fetching any tuples */
   Cost    total_cost;           /* total cost (assuming all tuples fetched) */

   List    *pathkeys;            /* sort ordering of path's output */
} Path;
```

The paths can be differed for example in the algorithm of scan or join operations that are executed. For example if a sequential or an index scan can be chosen, and both delivers the same result. Out of the cheapest path the plan tree is built by adding the necessary information for the executor again. Two possible path trees for our initial example in Figure 1 are shown in Figure 2. The IndexScan is probably the better path in this simple example, because it doesn't need to scan every tuple once.

Since there is only one implemented algorithm to evaluate matrix multiplication, we only

Figure 2: Two possible paths for our initial example

$\pi_{r,c}$
SeqScan/$\sigma_{r>10}$
Input1
$\pi_{r,c}$
IndexScan/$\sigma_{r>10}$
Input1

(a) Path with Sequential Scan (b) Path with Index Scan

consider the paths, which include this algorithm. Our algorithm is implemented in a way similar to the Nested Loop algorithm, only two different paths are possible, one with the left relation as the outer relation and one with the right relation as the outer relation. For both options we calculate the I/O costs and take the one, which has the smaller I/0 costs. I/O costs are the number of tuples which are read from a table or written to a table. To calculate these costs we first need to know how the algorithm works, out of this reason we move the cost calculations after the part about the executor.
So far PostgreSQL generates a plan for matrix multiplication operation. The next step is to extend the executor, so that we add some behavior to the new function.

4 Executor

Yet we have not discussed some algorithm and what exactly our new function should do. For this part the Executor is responsible. Like already mentioned the optimizer delivers a plan tree with all the necessary information to execute each node. At the beginning, a second tree with identical structure like the plan tree is built. This tree contains executor state nodes, where every state node belongs to a plan node from the plan tree and also points to it. A state node stores data, that is used during the execution of the corresponding plan node. In this way the information in the plan nodes are never changed and the plan tree is allowed to be read-only: all data that is modified during execution is in the state tree.
The plan tree is resolved iteratively as a demand-pull pipeline, that each node calls (pulls information) from its child nodes. At every call of a node one tuple is returned. If no more tuples are available, the called node returns `NULL` to its parent. For example if the root node is called, it calls its children and gets from each child a tuple. With this child tuples it creates a new tuple and returns it. After that, it calls its children again and does the same process. This is done until the root tuple returns `NULL`, which means that no more tuples are available and that we're done.
The implementation of a plan node consists of three functions: The `ExecInit`, the `Exec` and the `ExecEnd` function. The `ExecInit` function is called at the beginning of the execution process and initializes the state node belonging to the plan node.
After the initialization, the `Exec` function is called. This is the main part of the executor and the output is actually built by this function. For every output tuple the `Exec` function is called once and delivers a tuple. Like already mentioned we use the state nodes to save information between the different calls of the `Exec` function.
At the end, if the `Exec` function returns `NULL` the `ExecEnd` function is called. In the `ExecEnd` function everything which was initialized gets freed. A sample algorithm of this process is shown in a generalized way in Listing 9.

Listing 9: Main process in Executor

```
Main(){
  StateNode state = ExecInitNode(RootPlanNode);
  do{
    TupleSlot* tuple = ExecNode(state);
    output(tuple);
  }while(tuple != NULL);
  ExecEndNode(state)
}
```

While a simplified example of the iteratively called `ExecNode` function is described in Listing 10.

Listing 10: Generalized ExecNode algorithm

```
ExecNode(node){
     leftTuple = ExecNode(node->leftChild);
     rightTuple = ExecNode(node->rightChild);
     outputTuple = ProduceOutputTuple(leftTuple, rightTuple);
     //content of Planstate node may change
     return outputTuple;
}
```

To understand the next steps, a brief description of our algorithm would be helpful. In the following examples, we make the assumption, that the left is the outer relation. Our algorithm reads one row from the outer relation and then reads one column after the other from the inner relation. We assume that both relations are sorted, the left relation in (row, column) order and the right relation in (column, row) order. So we can read a row of the left relation or a column of the right relation with a sequential scan. For every (row, colum)-pair it outputs a tuple. If the inner relation has no more tuples to read, the function reads a new row from the outer relation and rescans the inner relation from the start and again reads column by column. An output tuple is constructed by a row i from the outer relation and a column j from the inner relation, which have the same length. To make the calculation examples clearlier, we call this length `med`. The row and column value of the output tuple is also i and j, while the value from it need to be calculated like the following (Assuming that R is the left relation and S the right):

$$value_{i,j} = \sum_{K=0}^{med} R.row_K * S.col_K$$

row_K is the K^{th} tuple in the current row. The resulting output table is sorted in the same way like the outer relation.
In every call of the function `Exec`, we need to know the dimensions of the tables to know how much tuples we need to read. During the execution of the algorithm we need to know, if we have scanned the whole inner relation for the current row of the outer, thus if we need a new row from the outer relation. This information is stored in a boolean called `mj_NeedNewOuter`. In another boolean (`leftIsOuter`) we store the information which path was picked by the optimizer, so if the left or the right relation is the outer. To store the current row from the outer relation, which we need to calculate the output tuples, we also have a float array in our state node. As a last element we have a `Matrix-Element`, whose type definition is shown in Listing 11. It contains two integers for the row and column and one float for the value, hence it 's similar to an output tuple. In this `MatrixElement` we store the values (row, col, value), which will be returned next.

Listing 11: Typedef of MatrixElement

```
typedef struct {
  int r, c;
  float v;
} MatrixElement;
```

All this combined with the standard attributes of a Join State result in the `MMulJoinState` node, which is shown in Listing 12.

Listing 12: Typedef of MMulJoinState

```
typedef struct MMulJoinState
{
  (normal join stuff)
  bool      mj_NeedNewOuter;
  int       rowLeft;
  int          colLeft;
  int       rowRight;
  int       colRight;
  float     *outerSafe;
  bool      leftIsOuter;
  MatrixElement rme;
} MMulJoinState;
```

This `MMulJoinState` is built and initialized in the `ExecInitMMulJoin`. Most of the values like the dimensions can just be assigned from the `MMulJoin` node to the `MMulJoinState` node. The boolean mj_NeedNewOuter need to be true at the beginning, since we need to read a new row from the outer relation, because we haven't read one yet. The first output row should have the values (0, 0, float). Because the row integer in the `MatrixElement` increases by 1 when a new row from the outer relation is read, the `MatrixElement` is initialized with the values -1 for row and 0 for column. The `ExecInitMMulJoin` is shown in Listing 13.

Listing 13: ExecInitMMulJoin Function

```
ExecInitMMulJoin(MMulJoin *node){
   MMulJoinState *mjstate;
   mjstate = makeNode(MMulJoinState);

   mjstate->rowLeft = node->rowLeft;
   mjstate->colLeft = node->colLeft;
   mjstate->rowRight = node->rowRight;
   mjstate->colRight = node->colRight;

   mjstate->rme.r = -1;
   mjstate->rme.c = 0;
   mjstate->rme.v = 0;

   mjstate->outerSafe = palloc(sizeof(float)*node->colLeft);

   mjstate->mj_NeedNewOuter = true;
}
```

After we initialized the `MMulJoinState` in the `ExecInitMMulJoin` function, the actual work is done in the `ExecMMulJoin` function. At the start of the `ExecMMulJoin` a `Matrix-Element` is initialized, in which we temporarily store each tuple we read from a relation.

The value of `node->rme.v` is set to 0 again, that we can store the intermediary result of our calculations in it. After the initialization we enter a while loop, which is running until we read a column from the inner relation and `running` is set to false. Normally the loop is processed only once. Just in the cases, in which we have read the full inner relation and getting `NULL` as result from reading the next inner tuple, we need to go back to the start of the loop and first read a new outer relation and rescan the inner. Inside the while loop first we check if we need a new row from the outer relation. If it's true, the outer safe is filled with a new row from the outer relation. For example, this is done in the first time the method is called. Like already mentioned the row integer increases by 1 when a new row is read. Also the column integer is set to 0 again and the inner relation is rescanned, because we have a new row. After this section we scan a column from the inner relation. If we get `NULL` as result we need to scan a new row from the outer relation and we go back to the top of the while loop. Otherwise we have a row from the outer relation and a column from the inner relation. Thus we can return a new tuple by multiplying the values from the outer row with the values from the inner columns. But before we return the tuple we increase the column integer of our `MatrixElement` by 1. This function is called until the scanning from the outer relation returns `NULL`, then the `ExecMMulJoin` Function is finished. In a simplified way the algorithm is shown in Listing 14. The functions `readSlot` and `readMatrixElement` read a Slot/MatrixElement(first parameter) and assign the values to another construct(second parameter).

Listing 14: ExecMMulJoin Function

```
ExecMMulJoin(MMulJoinState *node){
   // first some standard initalization
   MatrixElement temp = allocateTempMatrixElement();
   rme->v = 0;
   bool running = true;
   while(running){
          if(node->needNewOuter){
            //get new row of outerPlan
            for(int j = 0; j<colLeft; j++){
              outerTubleSlot = ExecProcNode(outerPlan);
              if(TupIsNull(outerTubleSlot){
                return NULL;
              }
              readSlot(outerTubleSlot, temp);
              node->outerSafe[j] = temp->v;
              //outerSafe is an array storing values of 1 row of the outerPlan
              node->rme.r += 1;
              node->rme.c = 0;
            }
           node->needNewOuter = false;
           ExecReScan(innerPlan);
         }

         for(int i= 0; i<colLeft; i++){
           //get new column of innerPlan
           innerTubleSlot = ExecProcNode(innerPlan);
           if(TupIsNull(innerTubleSlot){
             //need to scan a new outer first
             node ->needNewOuter = true;
             break;
           }else{
             //inner and outer tuple is available -> new tuple can be generated
             running = false;
           }
           readSlot(innerTubleSlot, temp);
           rme->v += node->outerSafe[i] * temp->v;
         }
       }
       pfree(temp);
       readMatrixElement(node->rme, resSlot);
       node->rme.c += 1;
       return resSlot;
 }
```

At the end in the `ExecEndMMulJoin` all the structures, which were initialized in the `ExecInitMMulJoin`, gets freed and the process is finished.

5 Cost Calculations

Now that we know the exact algortihm we can calculate the I/O cost to execute it. We know that the outer relation is scanned only once, and the inner relation is scanned `rowLeft` times if the left is the outer and `colRight` times if the right is the outer. If the left is the outer we need to read or write tuples like the following term:

$$rowLeft * colLeft + rowLeft * rowRight * colRight + rowLeft * colRight$$

If the right is the outer we need to read or write tuples like the following term:

$$rowRight * colRight + colRight * rowLeft * colLeft + rowLeft * colRight$$

Compared both I/O cost with each other we get the following result. We take the left as the outer if:

$$rowLeft * colLeft + rowLeft * rowRight * colRight + rowLeft * colRight <$$

$$rowRight * colRight + colRight * rowLeft * colLeft + rowLeft * colRight$$

Since we know that `colLeft` and `rowRight` are the same and both sides have the section $+rowLeft * colRight$ in it, we can reduce the calculation to the following(taking `med` for `colLeft` and `rowRight`):

$$rowLeft * med + rowLeft * med * colRight < med * colRight + colRight * rowLeft * med$$

Subtracting $rowLeft * med * colRight$ at both sides leads to the following:

$$rowLeft * med < med * colRight$$

Dividing both sides with `med` leads to the following:

$$rowLeft < colRight$$

Now we see that if the number of rows from the left relation is smaller than the number of columns from the right relation, it is cheaper to take the left relation as the outer and if it is bigger, it would be better to take the right as the outer relation.
The difference between the costs can be proved by some tests, where we compare the time to calculate the same query once with the "good" relation as the outer relation and once with the "bad" relation as the outer. Table 7 shows the time needed to process the query for both variants along with the 4 dimensions.

Table 7: Tests

Example	rowLeft	colLeft	rowRight	colRight	"good"	"bad"
1	10	10	10	1	1.077ms	1.157ms
2	10	10	10	100	7.636ms	8.245ms
3	1000	10	10	100	514.603ms	524.425ms
4	1000	10	10	1	8.504ms	13.727ms
5	1000000	10	10	10	73982.231ms	80542.414ms
6	10000	100	100	1	323.835ms	635.441ms
7	1000	100	100	2	649.677ms	970.235ms
8	1000	100	100	3	963.935ms	1293.760ms
9	1000	100	100	4	1296.738ms	1552.843ms
10	10000	100	100	5	1575.867ms	1853.592ms
11	10000	100	100	10	3113.068ms	3428.076ms
12	10000	100	100	100	31502.871ms	32000.282ms

The results from the tests show that the difference between the two options isn't that huge in the most cases, but the bad option takes nearly 2x longer when one of rowLeft and colRight is 1 and the other isn't (examples 4 and 6). In example 1 the time spent to calculate is so small and with a huge deviation, that this result isn't really significant. This difference can be calculated. We take the 2 cost calculations from above and divide them with each other to see the cost difference factor. To simplify the calculation we abbreviate the terms (`rowLeft` = rl, `colRight` = cr, `med` = m).

$$\frac{rl * d + rl * d * cr}{cr * d + rl * d * cr}$$

This term can be simplified by removing d to the following.

$$\frac{rl * (1 + cr)}{cr * (1 + rl)}$$

This formula confirms our tests, because we can see that the factor is at the most 2, if one of rl or cr is 1 and it sinks when the smaller of rl or cr increases. Approximated the elapsed time for the bad and the good plans are in the following relation:

$$bad = (1 + \frac{1}{\min(rowLeft, colRight)}) * good$$

Our test results are close to this relation. We did just 10 trials per test case and took the average. The numbers from examples 1-4 are really small, and because of this, there is a big deviation, but in the example with bigger input relations (examples 5-12), it is really close to this formula.

6 Conclusions

As a short final statement we want to assess our algorithm in efficiency and scalability.
Our algorithm has complexity O(rowLeft * med * colRight) which is a high complexity, but we cannot avoid it since for every output tuple we need to compute the inner product of a row and a column. In terms of scalability if one row contains a number of tuples and the memory can't store them all at once, the process breaks.
What can be done better is to reduce the I/O cost considering storing more than one row of the outer relation in memory. Of course the outer relation still need to be scanned once, but the inner relation doesn't need to be scanned `rowLeft` times, because we could calculate more than one output tuple with one scanned column from the inner relation.

The algorithm would stay more or less the same, but instead of computing only one output tuple with a inner column, we could compute many tuples. When we could store i rows from the outer relation, we need to scan the inner relation only $\frac{rowLeft}{i}$ times. The I/O cost would sink to the following:

$$rowLeft * colLeft + \frac{rowLeft}{i} * rowRight * colRight + rowLeft * colRight$$

It should be noted, that in this case the outer relation isn't sorted anymore.
Another beneficial thing about the algorithm is, that it could be parallelized easily. This benefit comes from the one after the other row handling. Every row from the outer relation is handled separately one after the other, and in this structure, we can handle every row individual and so we can also handle every row in parallel.
As a last point we want to consider the I/O cost when the input relations are increased. Starting from the I/O cost function:

$$rowLeft * med + rowLeft * med * colRight + rowLeft * colRight$$

With the I/O cost function, we can see that the I/O costs are increased approximately linear when one of `rowLeft`, `colRight` or `med` is increased. This means if one of the dimensions is doubled, also the I/O cost doubles. When `rowLeft` is doubled we need to scan the inner relation twice as much. When `colRight` is doubled we need to scan twice as many columns per row as before. In this two cases just one input relation has the double size, but when we double `med` both input relations get doubled and still just doubled I/O cost. When `med` is doubled, we just need to read twice as many tuples and so the I/O cost also just get doubled. In Table 7 in examples 6-12 only one dimension is changed, and the cost are changed in the same factor in the good plan. In examples 1-4 there is also just one dimension changed, but like already mentioned the numbers are to small to have really significant data. So if only one dimension is increased the cost increases linearly. But if more than one dimension is increased, then the cost increases polinomially.