ISSN 0944-5900

Uwe J. Langer

Join optimization in distributed database systems based on a complete search space

PREPRINTS
CS - 04 - 95

UNIVERSITÄT ROSTOCK

Preprints aus dem Fachbereich Informatik

HERAUSGEBER: Fachbereich Informatik der Universität Rostock

LEKTORAT: Autorenkorrektur

Die Preprint-Reihe des FB Informatik dient der Vorveröffentlichung der aktuellen fachlichen Ergebnisse der jeweiligen Autoren. Die fachliche Verantwortung für den Inhalt des Heftes liegt ausschließlich beim Autor.

ZITAT KURZTITEL:
Uwe J. Langer:
Join optimization in distributed database systems
based on a complete search space - Rostock : Universität Rostock, 1995. -
(Preprints aus dem Fachbereich Informatik; 17)

ISSN 0944-5900

BEZUGSMÖGLICHKEITEN: Universität Rostock, Presse- und Informationsstelle, Wissenschaftspublizistik
18051 Rostock, ☎ (0381) 4 98 10 36; FAX (0381) 4 98 10 70

Universität Rostock, Fachbereich Informatik, 18051 Rostock

DRUCK: Drucktechnische Zentralstelle der Universität Rostock 925/95

Join optimization in distributed database systems based on a complete search space

Uwe J. Langer *

University of Rostock
Computer Science Department
Database Research Group
18051 Rostock
Germany
ul@informatik.uni-rostock.de
http://wwwdb.informatik.uni-rostock.de/~ul/

ABSTRACT

In the context of the project HEAD* (*H*eterogeneous *E*xtensible *a*nd *D*istributed DBMS) we present a technology to distribute join queries over distributed relational DBMSs in a heterogeneous workstation network. Special aspects are the optimization of multi-way joins and the optimal parallel execution based on data flow control involving CPU, I/O, memory allocation and network load.

We focus our attention on spanning a complete search space without invalid elements to optimize multi-way joins with help of an approximative search algorithm and a complex cost function.

1. Introduction

Join operations have a great importance for the performance of algebraic query execution due to their high costs. Therefore the optimization of complex join expressions is an important part of the optimization in relational database systems.

There are many approaches to optimize complex join trees. Besides conventional optimization strategies exploited for local DBMS there are approaches for multiprocessors, homogeneous and heterogeneous distributed DBMS based on a shared memory, shared disk or shared nothing architecture.

The base of the most published articles are optimization approaches from [SAC+79] and [KrBZ86] (KBZ-algorithm).

* The research was supported by the DFG (German Research Association) under contract: Me 1346/1-2

Most relational database query optimizers concentrate on linear query evolution plans [SwGu88, BIL+89, DGS+90]. This holds in the context of sequential execution or pipeline parallelism only.

Since multi-processor machines and workstation nets with powerful communication media has been available, horizontal parallelism supported by bushy trees became attractive, too [DeKT90, LuST91]. Extreme tree shapes like linear and balanced bushy trees play an important role if we have homogeneous processor nodes [HoSt93].

The cost evaluation was often limited to special resources like the communication media. If communication costs are dominating we have to focus our attention on avoiding Cartesian products and minimizing the size of intermediate relations. Examples are shown for instance in [LaVZ93, YCWT93, Swam89]. Heuristic strategies like greedy algorithms and the algorithm for Complete and Feasible sets of cuts reduce the computing complexity and the transmission costs depending on relation size and selectivity. [ChYu94, LuST91]

In [ScDe90] the application of right deep trees is analyzed in the case of main storage deficiency in a shared nothing multiprocessor environment. A similar approach is shown in [ZiZB93], besides linear trees and bushy trees so called zig zag trees are introduced. These zig zag trees permit a better adaptation of query execution to the available memory.

In [DuKS92] proposals made by [SAC+79] and [KrBZ86] are applied to a heterogeneous environment. As a result linear and special bushy trees are generated.

Extreme tree shapes are bushy, left deep and right deep tree. The others reflect several transition states (Figure 1).

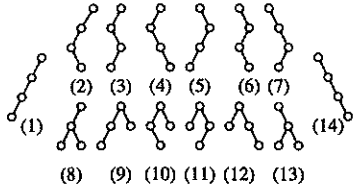


Figure 1: Possible shapes for a join tree with four vertices

Without join dependencies all combinations of join arrangement have to be considered. In figure 1 there are $14 * 4! = 336$ arrangements. This variety can be limited by join dependencies.

In a heterogeneous workstation environment like HEAD where load from other non database applications is considered, it is ingeniously to permit transition shapes between linear and balanced bushy trees to get a finer granularity for a better utilization of main storage, load, hardware power and the parallelism of available computer nodes to achieve a maximal throughput.

Including such transition shapes into the search space was done in [Kang91, GaHK92].

A very hard problem is the spanning of the complete search space. The step by step optimization of query plans as in [Fong86], heuristic transformations as in [LVZC93] or rule based transformations of the query plan [PiHH92] deliver not necessarily the optimal plan.

The presentation of a complete search space by usual strategies partially results in tree shapes including Cartesian products as in [GrMc93, LePP91].

In the past treeshapes were subdivided into linear, bushy and the so called zig zag trees and very often the relationship between tree shapes and hardware resources was examined. It is based on the fact, that until now the optimization approaches of multi-way joins *do not solve the problem of spanning a complete search space without Cartesian products.*

In this article we present an approach for mapping the search space without Cartesian products. In general the solution space $\mathcal{L}$ is a subset from the search space $\mathcal{S}$, i.e. $\mathcal{L} \subset \mathcal{S}$, but in the following approach the search space and the solution space are identically, i.e. $\mathcal{L} \equiv \mathcal{S}$.

1.1. Multi-way joins

In the project HEAD only binary and unary operations are considered. Therefore multi-way joins are transformed to sequences of binary joins. The results are join trees, i.e. $\overset{n}{\underset{i=1}{\bowtie}} R_i$ is substituted by an expression like this:

$$(((..(R_1 \bowtie R_2) \bowtie R_3)..) \bowtie R_n)$$

A join tree does not contain any other algebraic operation then joins.

Equivalence rules are used to isolate join trees from the given query tree to get join trees without any other algebraic operations. The leaves of a join tree could be complex sub-trees as the root node of the join tree can be a sub-tree of a complex expression too. They are abstracted to simple relations here.

Semi-join programs as used in [ChLi85] are not considered in this article, since transmission costs are not the dominating factor in our environment and so we don't benefit from including extra operations into the query plan.

2. Abstract tree shapes

2.1. Input data stream arrangement

The arrangement of input operations concerning join operations is significantly. In hash join cascades it influences the arrangement of hash tables. Without loss of generality the hash join is considered as an example for the other join implementations in the following.

In this article the right sub-relation is considered as the inner relation of a join operation used for the hash table (build phase) and the left sub-relation as the outer relation is used for comparison with the hash table (probe phase) in opposition to [ZiZQ93, AdBh93].

The arrangement of input data streams influences the query execution, the memory need and the system throughput in general. The input data streams can be produced by simple relations and complex sub-trees. The data-flow can be controlled by the input stream arrangement.

The arrangement depends on three criteria. These criteria are partly contrary (Figure 2). We assume that paging between intern and extern memory decreases the system throughput hardly. Therefore an optimization criterion is to prevent paging.

More details about input data stream arrangement are described in [FlLM95].

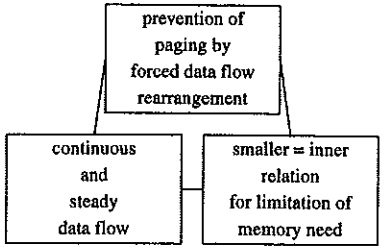


Figure 2: Contrary criteria for efficient joins

2.2. Selected tree shapes

In the following selected tree shapes are examined to show the relationship between concrete tree shapes and resource allocation. The hash join is used as a representative again.

Join dependencies are not considered in the following, i.e. it is assumed, that transformations do not implicate Cartesian products on the basis of join dependencies. Concrete values like size of relations, hash tables and so on are not taken into account.

Parameters are:

- The count of *hash tables* used at the same time.
- Needed *Phases* to execute sub-trees. A phase is the duration to load one or more hash tables into the main memory or to execute a steady and continuous data-flow. This approach is in contrast to [ZiZB93], where only executed data flows are called phases. In the following the time to load a hash table is called one phase, too.
- All necessary *begin* and *end cycles* to execute a query tree. These cycles represent the duration of some algebraic operations waiting for data in a cascade of join operations or the termination of the last operation caused by the data flow control. End cycles have a low importance since bound resources can be released after the last data block was processed. Begin cycles are of a larger importance because the initialization of operations have to be completed before the first block of data arrives at any operation.
- The transmission of a complete data stream between two operations is called *communication unit* (CU). In the following example the locations of R_1 - R_8 are assumed to be on the node performing the join operation. The results are local communications with very low transmission costs. On the other side there are no interactions between local communication and global resources. Moreover the network communication is assumed to be a communication in a network without subnets.

2.2.1. Left deep and bushy trees

In the first part the left deep tree *(LDT)*, bushy trees *(BT)* and the balanced bushy tree *(BBT)* are considered only (Table 1 and 2). In the column "Hash tables" the first item represents the maximal count of long living and coexisting hash tables, the second one represents temporal existing hash tables with smaller importance.

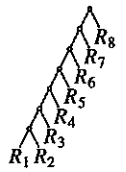
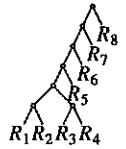
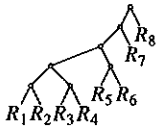
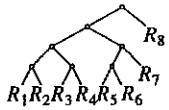
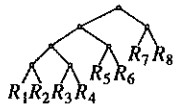
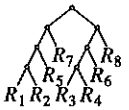
0	1	2
		
3	**4**	**5**
		
6		
		

Table 1: Examples for left deep and bushy trees

2.2.2. Evaluation of left deep and selected bushy trees

Linear trees (example *index*#0) prefered in [SAC+79, SwGu88] are executable in only two phases. But in comparison to bushy trees the concurrent resource allocation is very high. In the example in table 1 there are seven coexisting long living hash tables. On the other hand there is the maximal count of begin and end cycles. A further disadvantage is the concurrent transmission of many data blocks across the whole network.

In the case of large relations the usage of this tree shape is restricted due to memory allocation and communication load. On the other side it is usefully for on multiprocessor machines with a large shared memory and fast local communication mechanisms.

In distributed systems with a shared nothing architecture the left deep tree shape (*index*#0) is only applicable for small relations.

#	Typ	Hashtables	Phases	Begin cycles/ end cycles	CU
0	LDT	7	2	7+7	6
1	BT	6+1	3	6+6	5
2	BT	5+2	3	5+5	4
3	BT	5+2	3	4+4	3
4	BT	4+3	3	4+4	3
5	BT	6+1	3	4+4	3
6	BBT	4+2	4	3+3	2

Table 2: Evaluation of linear and bushy trees

The other extreme case is the balanced bushy tree (*index*#6) with a smaller resource need at once. It has fewer begin and end cycles but more phases to execute a query plan. All other tree representatives are steps between these two extreme forms.

Notice the bushy tree with *index*#4. It is a favorable compromise on the advantages and disadvantages of extreme tree shapes. Without loss on efficiency its regular structure is linear extensible through further join operations.

2.2.3. Right deep and zig zag trees

In the following zig zag trees *ZZT* and the right deep tree *RDT* are investigated with the equal count of joins and relations as in the example above (see Table 3 and 4).

#	Typ	Hash tables	Phases	Begin cycles/ end cycles	CU
7	RDT	1+1	8	1+1	1
8	ZZT	2+1	7	2+1	2
9	ZZT	3+1	6	3+1	3
10	ZZT	4+1	5	4+1	4
11	ZZT	5+1	4	5+1	5
12	ZZT	6+1	3	6+1	6

Table 4: Evaluation of right and zig zag trees

2.2.4. Evaluation of right deep and zig zag trees

The evaluation of zig zag trees shows that the resource demand of these trees can be influenced and controlled in an easy way. An interesting fact is the inverse proportional relation ship between execution speed and resource need, mainly main storage and communication media. Therefore it is possible to find an ideal mapping between the just available resources and a zig zag tree.

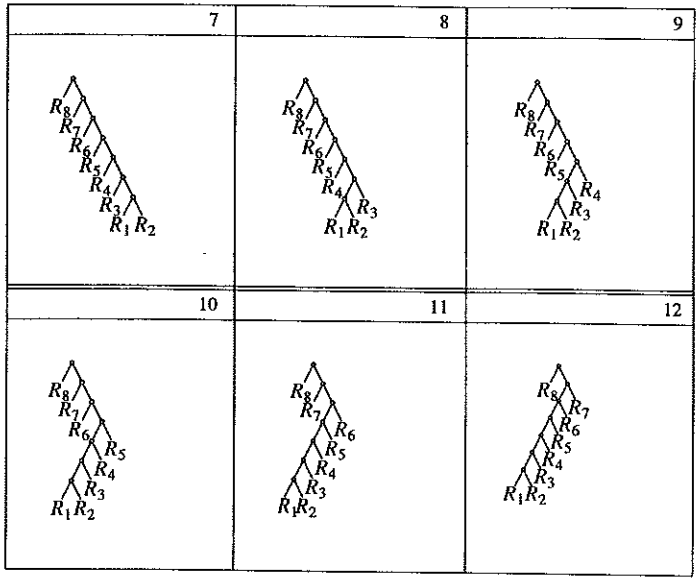


Table 3: Right deep and zig zag trees

2.2.5. Comparison between bushy and zig zag trees

Bushy trees permit in contrast to zig zag trees a limited mapping between available resources and possible tree shapes. They are very restrictively in case of resource deficiency. Zig zag trees allow a controlled resource allocation till a minimum of resources.

A comparison of efficiency between nearly adequate representatives (join tree with *index*#4 vs. join tree with *index*#12 or join tree with *index*#6 vs. join tree with *index*#11) shows the higher speed of bushy trees based on the same memory need but less phases. The load of the communication media is smaller, too.

The following conclusions are derivable:

- Bushy trees are to prefer to handle smaller or medium resource load in the global system.
- A heavy load in a global system forces zig zag trees due to their easier adaptability and the better support in a case of the heavy loaded system.
- A minimum of available resources forces right deep trees.
- Only the existence of voluminous free resources justify the usage of left deep structures.

3. A complete search space

A join tree can be transformed in such a way that the result is a correct join tree. The tree shape is changed but the resulting relation produced from the input relations has to be identically to the output relation based on the original tree shape.

There are two questions:

- Is it usefully to consider all possible alternative join trees?
- How can selected join trees be mapped in an efficient way?

Especially complex queries justify the consideration (or the ability of consideration) of all alternative join trees because the answering times may differ extremely.

3.1. Equivalent sets

To every join tree there is a finite number of equivalent join trees. All of them can be generated by a finite number of steps based on equivalence rules. These trees differ in their costs. We are only interested in trees with the smallest costs. These can ever be found through exhaustive search.

Let Ω be a set of join trees derivable from a given join tree t_0 and there is a subset T without any Cartesian products. We are only interested in the subset $T \subset \Omega$ excluding Cartesian product from the set of all graphs Ω with $|\Omega| = k!$ including the combinatorial variety derived from the original join tree t_0. k is the count of binary join operations of the join tree.

To optimize join trees with any approximative search algorithm the complete search space has to be spanned. A problem is the mapping with respect to the equivalence set of join trees and a kind of representation suitable for approximative search algorithms.

3.2. Adaptation to approximative search algorithms

Approximative search algorithms need the size of the search space as well as the lower and upper bound of a range as input parameters. It generates solutions evaluated with the help of a cost function. Dependent on the search strategy the approximative algorithm systematically approximates to the global optimum without searching exhaustively.

Solution proposals of any approximative algorithm are only numeric values. That means, the optimization of complex structures needs a mapping to a more abstract representation.

To be open to any serial and parallel algorithm an integer approach follows mapping join trees to a set of representatives. Therefore the term set of representatives is defined here.

Definition 1:

A set of representatives $\Upsilon = \{u_i\}$ with $u_i \in \mathbb{N}$ is a bijective mapping between the set of equivalent join trees and a closed interval $[0,..,n-1]$ of natural numbers (internal representation).

This set of representatives is used by the approximative algorithm as the search space. It is assumed, that the count of representatives is correctly calculable.

In the best case the search space can be an integer space only. There should be only a few abrupt changes in the space of values caused by small steps in the definition space. Otherwise the advantage of an approximative algorithm to find the global optimum with statistical warranty in a finite number of steps is lost [InRo89]. As an effect the final value could be the best of all considered values, but it can extremely differ from a local or the global optimum.

For an efficient use of an approximative algorithm assumptions are needed, which are enumerated in the following.

Like [PaKe86] special criteria have to be fulfilled.

3.3. Criteria for an efficient internal representation of a query graph

The internal representation of the query graph has to fulfill the following criteria to get the warranty of applicability and the necessary efficiency.

- $\forall (t_i \in T)\ \exists\ (u_i \in \Upsilon)$ with $T = \{t_i\}$ and $\Upsilon = \{u_i\}$, i.e. all query graphs can be figured.
- All query graphs have to be represented in a unique and fair kind to guarantee a unique management.
- The solution space should be convex. Only trees and nothing else has to be generated to get the warranty that every new derived graph lies in the search space.
- The predecessor and the successor have to be produced with a low expense.
- $pred(t_i) = t_{i-1} \Leftrightarrow pred(u_i) = u_{i-1}$ and $succ(t_i) = t_{i+1} \Leftrightarrow succ(u_i) = u_{i+1}$, i.e. small changes in the query graph have to implicate only small changes in the internal representation and small changes in the internal representation have to implicate small changes in the query graph, too.
- $|T| = |\Upsilon|$, i.e. the cardinality of the set of all equivalent query graphs has to be equal to the cardinality of the set of representatives.

3.4. Transformation of join trees

To optimize acyclic graphs these prerequisites are easily to fulfill. Therefore in the first step we assume sub-queries with acyclic join dependencies.

3.4.1. Creation of equivalent join trees based on acyclic join dependencies

Let t_0 be a given join tree extracted from a complete query tree. Equivalent join trees are generated based on algebraic equivalence rules.

Let T be a set of equivalent trees to t_0 with $T = \{t_i\}$, $i = 0..n-1$, $n \in \mathbb{N}$, where $n = |T|$ is the count of all equivalent trees.

A join graph (Figure 3(b)) can be derived from a given join tree (Figure 3(a)).

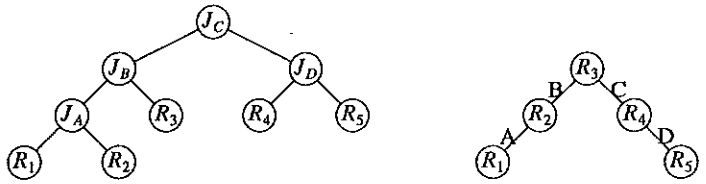


Figure 3: Join tree and join graph of acyclic join queries

To use any approximative search algorithm for join optimization it is necessary to have the possibility to map the variety of sequences and arrangements to the search space.

It is assumed that there are no multiple edges. Multiple join dependencies between relations or sub-relations are concatenated to one join dependency.

To determine all the possible equivalent sub-trees the following special graph reduction approach is chosen:

Select any edge from the join graph, fuse the two disjunct vertices together and change the label, i.e. $R_i \bowtie R_{i+1} \rightarrow R_{(i,i+1)}$. Repeat this until one vertice is left only. Multiple edges are concatenated to a simple one. The index of the remaining vertice describes a correct join tree. In this case valid join trees without Cartesian products are generated.

The search space of an acyclic join graph is equal to the product of all alternatives to select an edge in every step. It means n relations span a search space of $(n-1)!$ equivalent join trees.

The commutativity of joins has no effect to the spanning of the solution space. The order of the sub-relations as input streams is not considered here.

3.4.2. Creation of equivalent join trees based on cyclic join dependencies

To span the search space for cyclic join graphs satisfying the demands of the space of representatives is much more difficult than to span the search space for acyclic join graphs.

With the assumption of fuseing multiple edges and the concatenation of the join dependencies there is the possibility of a discontinuous decreasing of the edge count. Therefore the count of the equivalent join trees is difficult to determine.

The mapping from the set of n equivalent join trees to the closed interval of $(n-1)!$ integer representatives in the same way as the mapping of acyclic join graphs involves many invalid representatives. A part of the evaluation of the representatives has to be the validity examination to avoid Cartesian products. Invalid representatives can not be mapped to the solution space. The solution space would become concave.

Therefore in the following an approach is shown to achieve a correct mapping between join trees based on cyclic join graphs and the space of representatives satisfying the prerequisites.

The mapping from the set of equivalent join trees T to the set of representatives Υ and vice versa can not be handled in a unique way because of the structure complexity and the differences in structures. On the other hand only the mapping from $\Upsilon \rightarrow$ T and equal cardinalities of the sets Υ and T are necessary for join optimization. For simplification the demand of a bijective mapping between the set of representatives and the set of equivalent join trees (T$\leftrightarrow\Upsilon$) is reduced to injective mapping from the set of representatives to the set of equivalent join trees ($\Upsilon \rightarrow$ T) with the additional assumption of the equivalence of the cardinalities ($|T| = |\Upsilon|$) (This is shown in figure 4).

3.4.3. Coefficient tree

The following definitions are needed for the presentation of the mapping strategy between join representatives and concrete join trees.

The presented mechanism based on the graph reduction approach described above uses a so called coefficient vector tree to map any representative $u_i \in \Upsilon$ with $u_i \in \mathbb{N}$ to a concrete join tree t_i. This coefficient vector tree consists of so called coefficient vectors built from coefficients. The fuseing of disjunct vertices with eliminating multiple edges by concatenation of join attributes means the reduction of more than one edge at a moment dependent on the graph

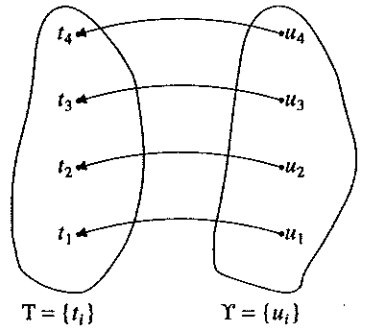


Figure 4: Injective mapping from Υ to T

structure. For instance in a cycle of three vertices and three edges (3-cycle) there remains only one edge after fuseing two vertices to one.

An edge can be part of many 3-cycles. Removing such an edge and the following melting of multiple edges implicates that many edges are removed at one time. This means a stronger reduction of the graph through special edges during the edge elimination. The term *edge value* is defined to describe nominally the reduction strength of an edge:

Definition 2:
The edge value p of an edge e in an undirected graph $G(V, E)$ with a set of vertices V and a set edges E and $e \in E$ is the grade of reduction of the graph $G(V, E)$ to $G'(V', E')$ with $p = |E| - |E'|$ in the case of elimination of e.

Based on edge values there is an unique injective mapping between a given join tree and a so called coefficient vector. Dependent on the edges of a join graph there exist different rule sets R_n used for the graph reduction taking into account the behavior of edges with higher edge values ($p > 2$).

Definition 3:
An edge coefficient m_i is the count of all edges of the same edge value p in a join subgraph $G(V, E)$ without edges which are part of cycles with more than three edges.

Definition 4:
A rule set R_n is the set of rules describing the behavior of a concrete base structure of join graphs and the dependencies of edges during the graph reduction. n is the index of the greatest edge value in the sub-graph.

Definition 5:

$$\vec{c_n} = \begin{bmatrix} m_1 \\ .. \\ m_n \\ - \\ (id, s) \end{bmatrix}_{R_n}$$

A coefficient vector $\vec{c_n}$ of a join graph $G(V, E)$ is a sorted sequence of edge coefficients

(with reference to the edge values) m_i with $i = 1..n$, where n is the maximal index belonging to a rule set R_n. (id, s) are optional tuples describing cycles with $s > 3$ edges and id is a unique cycle identifier. The extension of coefficient vectors by tuples (id, s) is not necessarily.

Reduction behavior

The reduction behavior is not only determined by the edge values p. It is also strongly influenced by the graph construction. On the other side the graph construction influences the edge values, too. The edge value of a single edge can be calculated in the following way: $p = |c_3| + 1$ with $|c_3|$ is the count of adjacent 3-cycles.

Rule sets

For different graph schemas different rule sets are necessary to apply the graph reduction mechanism. The following rules decribes the behavior of coefficients during the reduction of one edge. The schemas in table 5 belong to the rules. If for instance one edge with an edge value of i is reduced, any edge coefficients are changed. In the case of $i = 1$ e.g. only the coefficient m_1 is changed.

Rule set 1:

- $m_1 = m_1 - 1$
- $m_2 = m_2 - 3 \wedge m_1 = m_1 + 1$

Rule set 2:

- $m_1 = m_1 - 1$
- $m_2 = m_2 - 2 \wedge m_3 = m_3 - 1$
- $m_3 = m_3 - 1 \wedge m_2 = m_2 - 4 \wedge m_1 = m_1 + 2$

Rule set 3:

- $m_1 = m_1 - 1$
- $m_2 = m_2 - 2 \wedge m_4 = m_4 - 1 \wedge m_3 = m_3 + 1$
- $m_4 = m_4 - 1 \wedge m_2 = m_2 - 6 \wedge m_1 = m_1 + 3$

Rule set 4:

- $m_1 = m_1 - 1$
- $m_2 = m_2 - 2 \wedge m_5 = m_5 - 1 \wedge m_4 = m_4 + 1$
- $m_5 = m_5 - 1 \wedge m_2 = m_2 - 8 \wedge m_1 = m_1 + 4$

etc.

3.4.4. Spanning the solution space

The following approach can be used to calculate the count of all equivalent join trees |T| and to get an injective mapping from one integer representative to the equivalent join tree:

It is possible to create a join graph G_{JD} (Figure 5 b) from a given join tree G (Figure 5 a). This join graph is identically for all join trees from a set T. From the join graph in Figure 5

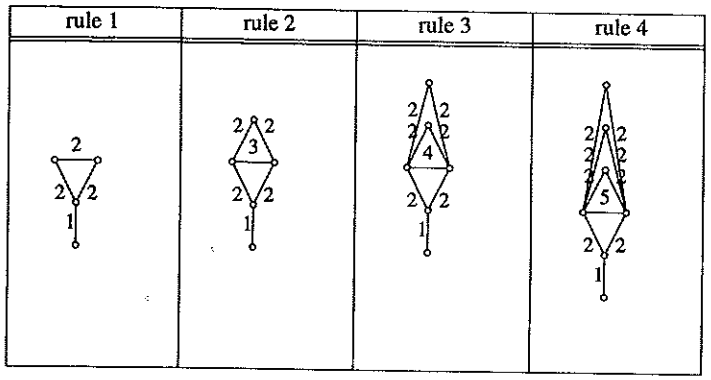


Table 5: Examples for rule sets 1 to 4

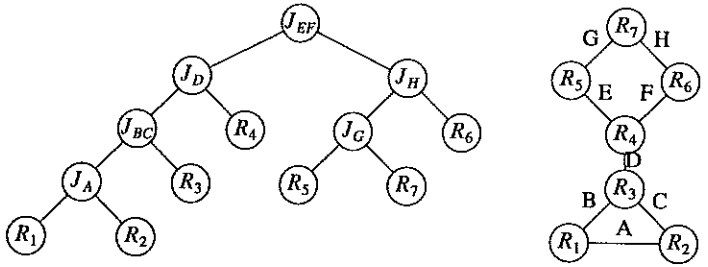


Figure 5: Join tree and join graph of a cyclic join query

the following coefficient vector can be generated:

$$\vec{c_1} = \begin{bmatrix} m_1 = 1 \\ m_2 = 3 \\ \dots \\ (id = 1, s = 4) \end{bmatrix}_{R_1}$$

The following coefficient vector tree is created by the complete derivation based on the application of rule set 1 and the repetition over the coefficient vector (Figure 6). The numbered arrows represent the accumulated number of equivalent sub-graphs. The coefficient vector tree is created one time during the join graph analysis and allows to calculate the exact search space size and to derive a concrete join tree $t \in T$ from a given representative $u \in \Upsilon$, which is shown in the following. To calculate the search space size all alternative counts for further derivations are stored temporarily for each arrow in a top down traversal. All of them are multiplied with the predecessor values bottom up. The values from alternative

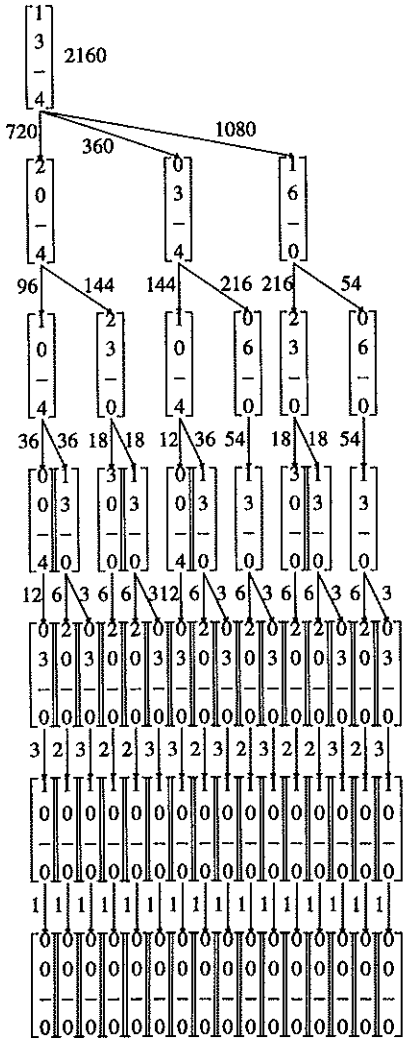


Figure 6: Coefficient tree of the example graph

arrows are added. As a result we get the exact count of join trees of the set T.

The mapping from a member of the representatives to a deterministic join tree based on the coefficient vector tree is presented as pseudo code (Figure 7). We assume that edges in the join tree have a fixed access order independent of removed edges.


```
x .. size of the search space
tree .. coefficient vector tree
treestruct {
   int index;
      /* coefficient index of  */
      /* parent node */
   int m[idx][coefficient_cnt
         + cnt_of large cycles];
      /* coefficients and large cycles  */
      /* are mixed together, normal  */
      /* coefficients have idx = 0 */
   int cnt;
      /* count of alternatives */
   tree * next;
      /* pointer to the next in */
      /* the same row */
   tree * subtree;
      /* pointer to first child */
};

treestruct *tree, *subtree;
while (tree->subtree)
{
   subtree = tree->subtree);
   while (x - subtree->cnt > 0) {
      x = x - subtree->cnt;
      subtree = subtree->next;
   }
      /* index of edge coefficient */
      /* for reduction
   edge_coefficient = index(subtree);

   real_subtree_size = subtree->cnt
        / tree->m[edge_coefficient][0];
   edge_idx = 0;
   while((x-real_subtree_size) > 0) {
         x = x-real_subtree_size;
         ++edge_idx;
         GRAPH_RED_APPR(edge_idx);
   }
} while (tree);

GRAPH_RED_APPR(edge_idx)
{
      /* create a new vertice from the
         adjacent vertices of the edge
         with index edge_idx */
      /* modify the label of the new
         vertice */
      /* eliminate multiple edges */
}
```

Figure 7: Mapping algorithm

Each selected sub-tree representing uniquely one join may be added to the actual join plan.

If there are base elements with different rule sets the parts can be concatenated in the following way:

$$\vec{c}_{1,2} = \begin{bmatrix} 0 \\ 3 \end{bmatrix}_{R_1} \overline{\begin{bmatrix} 0 \\ 4 \\ 1 \end{bmatrix}_{R_2}}$$

This represents the join graph in figure 8. The notation includes two base elements of two different rule sets without cycles of a size $p > 3$.

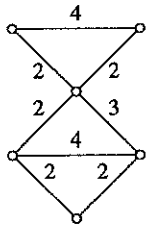


Figure 8: Join graph made from base elements

3.4.5. Ambiguous join graphs

There is a low probability for join graphs with multiple edges of an edge value $p > 2$ implicating different rule sets. But these graphs are not divisible into the basic join graph components described above. In this case a coefficient vector tree is not uniquely derivable to coefficient vectors (Figure 9).

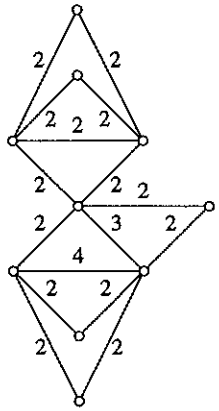


Figure 9: Ambiguous join graph

If such case is detected during the join graph analysis, a conventional optimization strategy (like the greedy algorithm) has to be used.

4. Summary and conclusions

In this article research results are presented optimizing join trees in a distributed, heterogeneous, relational DBMS based on data flow control. It was shown, that there is no universal tree shape for multi-way joins with optimal results independent from load distribution and hardware architecture. The attention was focused to the spanning of a complete search space for join optimization without any invalid elements. This is a prerequisite to apply

any approximative search algorithms in an efficient way and to find the global optimum for complex queries in a distributed DBMS.

The aim of the future work is to solve the ambiguous problem for complex join graphs and make it more universally.

References

[AdBh93] R. A. Adam and K. B. Bhargava, "Advanced Database Systems" in *Parallel Query Processing*, Springer, Berlin (1993).

[BIL+89] G. v. Bültzingsloewen, C. Iochpe, R. P. Liedtke, R. Kramer, M. Schryro, K. R. Dittrich, and P. C. Lockemann, "Design and Implementation of KARDAMOM – a Set-oriented Data Flow Database Machine" in *6th Int. Workshop on Database Machines*, Deauville (1989).

[ChLi85] A. L. P. Chen and V. O. K. Li, "An Optimal Algorithm for Processing Distributed Star Queries," *IEEE ToSE*, SE-11, 10 (Oct. 1985).

[ChYu94] M. S. Chen and S. Yu, "A Graph Theoretical Approach to Determine a Join Reducer Sequence in Distributed Query Processing," *IEEE Transactions on Knowledge and Data Engineering*, 6, 1 (February 1994).

[DeKT90] S. M. Deen, M. C. Kannangara, and M. C. Taylor, "Multi-Join on Parallel Processors," *IEEE*, pp. 92-102, University of Keele, Dept. of CS, Keele (1990).

[DGS+90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The Gamma Database Machine Project," TR 921, CSs Dept., University of Wisconsin-Madison (March 1990).

[DuKS92] W. Du, R. Krishnamurthy, and M-C. Shan, *Query Optimization in Heterogeneous DBMS* (1992).

[FlLM95] G. Flach, U. J. Langer, and H. Meyer, *HEAD - State of the art in june 1995*, Dept. of CS, Rostock Univ. (June 1995).

[Fong86] Z. Fong, "The Design and Implementation of the Postgres Query Optimizer," *Masters Report*, CS Division, DEECS, University of California, Berkeley, California (Aug 1986).

[GaHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution," *SIGMOD Record*, 12 (June 1992).

[GrMc93] G. Graefe and B. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search," *IEEE Conf. on Data Eng.*, p. 209, Vienna (Apr. 1993).

[HoSt93] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS," *Distributed and Parallel Databases*, 1, pp. 9-32 (1993).

[InRo89] L. Ingber and B. Rosen, "Very Fast Simulated Re-annealing," *Mathematical Computer Modelling*, 12, 8, pp. 967-973 (1989).

[Kang91] Younkyung Cha Kang, "Randomized Algorithms for Query Optimization," TR 1053, CSs Dept., University of Wisconsin-Madison (October 1991).

[KrBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of Nonrecursive Queries," *Proc. of the 12th Int. Conf. on VLDB*, pp. 128-13 (1986).

[LVZC93] R. Lanzelotte, P. Valduriez, M. Ziane, and J.-P. Cheiney, *Optimization of Nonrecursive Queries in OODBs*, INRIA, Rocquencourt, France (1993).

[LaVZ93] R. S. G. Lanzelotte, P. Valduriez, and M. Zait, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces," *Proc. of the 19th VLDB Conf.*, Dublin (1993).

[LePP91] P. Legato, G. Paletta, and L. Palopoli, "Optimization of Join Strategies in Distributed Databases," *Information Systems*, 16, 4, pp. 363-374 (1991).

[LuST91] H. Lu, M.-C. Shan, and K.-L. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution" in *Proc. of the 17th VLDB Conf.*, Barcelona, Spain (1991).

[PaKe86] C. C. Palmer and A. Kershenbaum, *Representing Trees in Genetic Algorithms*, IBM T. J. Watson Research Center, Yorktown Heights, NY (1993).

[PiHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst," *Proc. ACM SIGMOD*, pp. 39-48 (1992).

[ScDe90] D. A. Schneider and D. J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines" in *Proc. the 16th VLDB Conf.*, pp. 469-480, Brisbane, Australia (1990).

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *ACM* (1979).

[SwGu88] A. Swami and A. Gupta, *Optimization of Large Join Queries*, Dept. of CS., Stanford University, Stanford (1988).

[Swam89] Arun Swami, "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques," *ACM*, Dept. of CS, Stanford University, Stanford (1989).

[YCWT93] P. S. Yu, M.-S. Chen, J. L. Wolf, and J. Turek, "Parallel Query Processing" in *Advanced Database Systems*, Springer (1993).

[ZiZB93] M. Ziane, M. Zait, and P. Borla-Salamet, *Parallel Query Processing in DBS3*, INRIA, Rocquencourt (1993).

[ZiZQ93] M. Ziane, M. Zait, and H. H. Quang, "The Impact of Parallelism on Query Optimization" in *Proc. of Fifth Workshop on Foundations of Models and Languages for Data and Objects*, pp. 127-138 (Sept. 1993).