Managing the Operator Ordering Problem in Parallel Databases

Harald Kosch

Institute of Information Technology, University Klagenfurt
A - 9020 Klagenfurt, Austria
email: harald.kosch@itec.uni-klu.ac.at

Abstract

This paper focuses on parallel query optimization. We consider the operator problem and introduce a new class of execution strategies called **Linear-Oriented Bushy Trees** (LBTs). Compared to the related approach of *Bushy Trees* (BTs) a significant complexity reduction of the operator ordering problem, while not loosing quality, can be derived theoretically and demonstrated later on experimentally. In addition, we applied LBTs to parallel query optimizers using an enumerative search technique, like the parallel query optimizer of Microsoft SQL Server [1]. To do this, we propose necessary modifications to the exploring strategy of the search technique. An effective memoizing structure is reused and its size for the space of LBTs is computed. In order to guarantee the efficiency of the optimization process, a duplicate-free transformation rule set for the generation of LBTs is introduced and its correctness is proven. Finally, we summarize selected performance results to give a flavor of the practical relevance.

Key words: Parallel Databases, Parallel Query Optimization, Search Space, Linear-Oriented Bushy Trees.

---

1 Introduction

Modern database applications, such as data mining and decision support pose several new challenges to query optimization and processing [2]. One of the main issues concerns the processing of complex queries (e.g., recent Teradata relational data warehouse queries involved more than 30 joins while the size of some relations reach several Gigabytes [3]) [1]. Parallelism is one of the key technologies to handle those challenges [5], but increases the complexity of

[1] In our comprehension a complex query involves more than ten binary query operators [4] ; this boundary plus minus two operators is mostly adopted.

query optimization [6]. The most crucial problem to be solved within query optimization for relational, object-relational as object-oriented databases [7] is the **operator ordering problem**. An operator ordering describes the dependencies between the different algebra operators of a query and determines at the same time the degree of *inter-operator parallelism*, i.e., how many operators are executed concurrently. The **operator ordering problem** is the search for an optimal ordering with respect to a cost function.

The search space spawn by all operator orderings in parallel databases, without any restrictions, is extremely large, e.g., for a 16 way-join relational query (16 participating relations), the number of possible orderings is low bounded by $3.18 \cdot 10^{11}$ [8]. In the context of this high complexity, most early works have restricted the search to orderings allowing no inter-operator parallelism. However, the exploitation of inter-operator parallelism has been shown to be very effective in the case of high performance parallel machines. Thus latter works managed the search complexity by applying randomized search or heuristics [9,10]. These methods work well if low cost partitions of the search space are accessed. However, as the navigation is more or less randomly, locally advantageous moves might be accepted allowing no access to global low cost strategies later on. This is not acceptable for complex database applications.

Rather than navigating 'blindly' through the search space we propose to consider a subspace of it, the so called **Linear-oriented Bushy Trees (LBTs)**. LBTs identify a regular structure in the search space including inter-operator parallelism and face a significantly smaller search space than the one spawn by Bushy Trees (BTs). We will validate the pertinence of the LBT processing in a serie of experiments with 1800 optimized queries. The main result is that using LBTs instead of BTs reduces the optimization time considerably: in mean by about a factor of 3 for query complexities > 10 (in the worst case the BT optimization time was nearly 2 minutes, as it was 35 seconds for the LBT). In addition, the costs of the generated plans differs slightly, i.e., LBT generated plans have in mean about 10% less quality than BT plans.

The paper is organized in nine Sections. Next Section 2 introduces to parallel query optimization and defines LBTs. Section 3 discusses the LBT processing in relation to previous works. Section 4 calculates the LBT search space size and compares it to that of the Bushy and Linear Trees. Section 5 introduces a memoizing structure (MEMO) to encode the search space of LBTs. In order to cope with LBTs, we present modifications to broadly used enumerative search strategies. Section 6 proposes a novel duplicate-free transformation rule set to encode the LBT space. Section 7 computes the size of the MEMO-structure encoding the space of LBT. Section 8 presents experimental results. Finally, Section 9 concludes this paper.

2 Parallel Query Optimization and Linear-Oriented Bushy Trees

Query optimization in an uni-processor DBMS can be roughly divided into three main tasks [11]: the rewriting of the query (e.g., push the selections as near as possible to the scan of the base relations/object sets), the operator ordering problem and the choice of the access methods (e.g., use of an index or not). In a parallel system, the optimization problem is more complicated with the new dimension introduced by parallelism, especially for the operator ordering and the now introduced parallel scheduling phase. The number of feasible execution strategies increases dramatically [12] as in addition to uni-processor query optimization, the degree of inter- (i.e., the number of concurrent operations or how bushy is the associated processing tree), and intra-operator parallelism (i.e., the number of sub-operations executed concurrently on partitions of the input relation/object sets), and possible pipeline parallelism has to be determined. Furthermore, proper scheduling and allocation of resources are critical to parallel databases. Two main meta-strategies have been developed to manage parallel query optimization. In the first one, the *two-phase* optimization strategy [13], the operator ordering generates one solution without taken the parallel resources into considerations. This solution is then scheduled for parallel execution in a second phase. In the second one, the *one-phase* optimization strategy [14], operator ordering and scheduling go hand in hand. For every generated ordering a physical scheduling is done.

In this paper we focus on the *operator ordering problem* which is common to both, the one-phase and two-phase strategy. It is the most important problem to be solved in parallel query optimization, as all other problems depend on it. Parallel scheduling is addressed elsewhere and good solutions have yet been proposed (see [15]). The estimated cost associated to one ordering and the determination of the degree of intra-operator parallelism will be concretized in the experimental context for a sample parallel machine and database schema. For the general methodologies we can make abstraction of it.

In the remainder we will concentrate on parallel relational databases and equi-join operators with single attribute join predicates [2], but the proposed methodologies apply to object-relational and object-oriented databases too. This is true because the best way to evaluate a path expression in an OO(-REL)DB query is to use pointer joins between the extents of the objects involved in the path expression [16]. Our approach remains general, because the join operators can be exchanged without further adaption of the proposed methodologies to other complex operators (as intersection, union and the object-oriented flatten operator). As usually performed by DBMS products, aggregation operators are supposed to be executed after the join.

[2] The join predicate has the form $R.attr1 = S.attr2$ for some relations R and S and relation attributes $attr1$ and $attr2$.

Problem formulation We concentrate on optimizing simple SQL queries of the form:

SELECT . . . FROM R_1, R_2, . . . ,R_n
        WHERE p_1 AND p_2 AND . . . AND p_m

This query involves n relations R_1, R_2, . . . ,R_n and m uncorrelated predicates p_1, p_2, . . . p_m. A predicate p_k is either a *local predicate* on some R_i, if it references only R_i, or a *join predicate* between two relations R_i and R_j. Each *local predicate* p_k on some R_i can be safely ignored, if we replace R_i by $\delta_{p_k}(R_i)$, where δ denotes the relational select operator. Thus, each query Q can be represented as a pair (R, P), where R is the set of participating relations $R = \{R_1, R_2, \ldots, R_n\}$ in Q and $P = \{p_1, p_2, \ldots p_m\}$ is the set of used join predicates.

We adopt the notion of a (valid) query processing tree to identify a member of the search space [17]:

Definition 1: A **query processing tree** for a query $Q = (R, P)$ over the relational algebra has the following syntax: $PT ::= R_i$ $(R_i \in R)$ or $PT ::= PT_1 \bowtie_{pred} PT_2$ where PT_1 and PT_2 are query processing trees and *pred* is either true (Cartesian Product) or it is a conjunction of join predicates that reference the relations in PR_1 and PR_2 exclusively.

Definition 2: A **valid query processing tree**, shortly processing tree, for Q is a query processing tree that uses each relation $R_i \in R$ exactly once and each join predicate in P at least once. We do not allow sharing of predicates or relations in the expression for the clarity of our explanation, and we are aware that though an expression with sharing may result in better performances.

Definition 3: The **operator ordering problem** is the search for the best ordering, thus the best processing tree for a given query Q with respect to some cost function C [17]. In order to solve the operator ordering problem, a query optimizer has to determine an appropriate search space, search strategy and cost function [18].

The shape of the processing tree is used to classify the operator orderings and, thus, the related parallel query processing strategies. Two major forms are actually distinguished [14], *linear*- and *bushy trees*. The definitions are the given below.

Definition 4: A **linear tree**, shortly LT, is a processing tree, where at least one input relation of each join must be a base relation. A **bushy tree**, shortly BT, is a processing tree where no restriction on both input relations are imposed.

LTs execute only one join operator at the same time. Three types of LTs are actually distinguished: *left-deep trees* bases on a data-parallel strategy (only intra-operator parallelism is exploited), *right-deep trees* bases on a more pipelined strategy and *zig-zag trees* allows a base relation to be either left or right operand for each join in the tree.

BTs allow the processing of two or more join operators lying on different paths of a query processing tree at the same time. Almost all query optimizers working on bushy trees have yet considered the complete spawn search space. Some interesting initial works considered *segmented right-deep trees* [19], which are composed of a set of right-deep trees, called segments. They allow the result relation of a segment to be either the left or the right operand of any of the subsequent segments. We will discuss this tree form in the next Section 3.

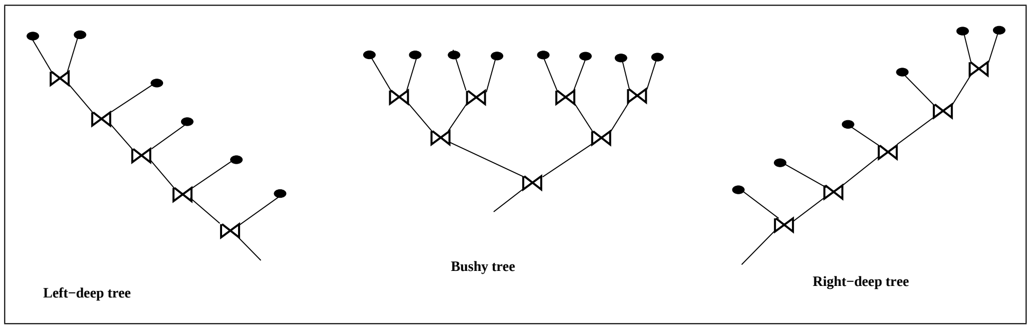

Fig. 1. *Example Processing trees. Example Processing Trees. From the Left: Left-deep Tree. Bushy Tree. Right-deep Tree.*

Fig. 1 shows on the left hand a left-deep tree, in the middle a bushy tree and on the right hand a right-deep tree. Supposing that a hash-based join is used and all hash-tables of the left-input relations are built for a right-deep tree, the tuples of the right input can be pipelined through the whole tree [20]. This pipelining can be implemented efficiently, if the entire hash table of the left-input relation fits in main memory. Otherwise, high I/O handling must be done. In such a situation, if the result relation of a highly selective join can fit in main memory, a left-deep subtree might be a better choice [21].

An important task of the query optimizer is to select the appropriate search space for a query. Many uni-processor query optimizers restrict themselves to linear trees [22,23], because of the more complicated memory management and the larger search space of BTs (refer to Section 4). However, as pointed out by several authors [24,25], if resource availability is high, bushy trees should be considered in order to achieve efficiency. For parallel databases two supplemental phenomenon are met. First, a too high degree of inter-operator parallelism can lead to load balancing problems (see [25–28]) degrading the performance. Second, right-deep segmentation of a BT is hard to do [20]. Experimental studies with the Volcano product showed that parallel pipeline chains in BTs might degrade performance [29]. In this scope, we propose a new search space, the so called *linear-oriented bushy trees* to contribute to the bridging of the gap. The definition is given below.

Definition 5: A *linear-oriented bushy tree*, shortly *LBT*, is a processing tree where for each join operator at least one of its two input relations must be either a base relation or a *single join* (a single join takes as its input two base relations).

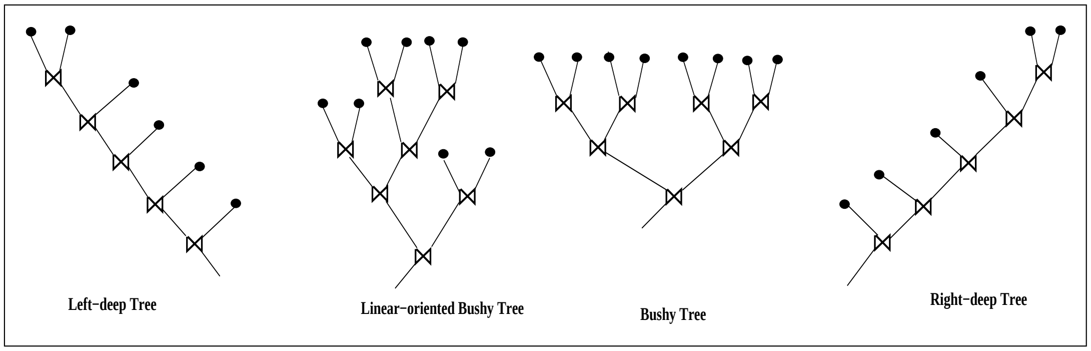

Fig. 2. *Example Processing Trees. From the Left: Left-deep Tree. Linear-oriented Bushy Tree. Bushy Tree. Right-deep Tree.*

Let us consider Fig. 2 which bases on Fig. 1. It displays an LBT on the second position from the left. To the right of the LBT, it shows a BT which is *not* an LBT (i.e., the left- and right input relation of the last processed join are neither a base relation nor a single join). Note that only for $n > 5$ (n number of relations participating in the query), the LBT space differs from the BT space.

LBTs unify advantages of both related LTs and BTs: on the one hand right-deep tree segmentation can be used, both in LTs and LBTs (but not BTs), because of the proximity of their definitions; on the other hand efficient inter-operator parallelism can be used, both in LBTs and in BTs (but not in LTs). However, LBTs exploit inter-operator parallelism more modestly than BTs and avoid, therewith, load imbalance problems.

3 Previous Works

Early related work in parallel query optimization (e.g., [21,30]) have concentrated on linear trees and *intra-operator parallelism* (e.g., [31,32]). These works have not yet considered operator orderings including *inter-operation parallelism* because of its high scheduling complexity and its difficult synchronization.

In the last years, several parallel DBMS products, as PARIS [33], Prisma [24] and the DB2 Parallel edition DBMS [34] have integrated inter-operator parallelism into the query execution machine. Performance evaluations demonstrate that, in the context of sufficient high resource availability (in terms of number of processors and memory size), inter-operator parallelism should be exploited for complex queries. In such cases, LT processing strategies risk to utilize the available resources inefficiently.

In order to manage the operator ordering problem for the BT space, randomized search or greedy heuristics are proposed. Randomized search has mainly been used in [9,14]. The performance of this method is hard to predicate, as the local transformations are chosen randomly out of the set of the possible ones. If the BT space has to be searched, it could not be guaranteed that, for the most cases, the search ends in a low cost partition of the search space. More recently, heuristics for the BT space have been proposed [10,35,36]. For example, [10] defines a greedy algorithm which constructs a complete processing tree, iteratively, from an initial set of trees consisting only of base relations. In each iteration, the algorithm obtains greedily a new (sub)tree that has the lowest costs from all possible combinations of joining two elements not yet considered in the set of subtrees. This greedy heuristic achieves, however, good quality orderings only for the special case of join graphs which are loosely inter-connected. Another example is the Sybase commercial optimizers [37] which is tuned especially for star queries.

Real-life applications can contain queries with very different types of join graphs, they may range from strings to highly inter-connected graphs. This is why several authors proposed to extend the Decision Support Benchmark Serie of the Transaction Processing Performance Council, to 'surprising real-life queries' [3] . An optimizer implementing heuristics for some query types risks to fail for an application introducing queries with many different join graphs. Thus, an important quality feature of a query optimizer is its capability to generate *good* solutions for a wide variety of input query types. Optimizers investigating always the complete BT space, if some of the already described conditions of resource availability are given, risk to get lost in the immense search space.

To the best of our knowledge, only two initial works, to build subspaces of the BT space, have yet been considered. First, the above mentioned *segmented right-deep tree space* [19]. The main drawback using this space is that an optimizer risks to have higher estimation errors for intermediate relation sizes, than with LBTs. The reason is that LBTs offer the advantage that at least one subtree is a single join or a base relation. Thus, we can expect rather fidel estimations about the intermediate relation sizes. Oppositely, in segmented

[3] For instance, consider the article of Richard Winter, entitled: "Summing Up: How the VLDB scene has changed", to be found at `http://www.wintercorp.com/rwintercolumns/vldb_summing.html`.

right-deep trees, the left-, and right input of a join may be a segment containing more than two relations. This leads, obviously, to less fidel estimations in the relation sizes, than using LBTs.

Second, a right-deep tree oriented subspace of the LBT space has been considered, for some performance tests, in the parallel database PRISMA [24] (project conducted by P. Apers). They have found out that a simple extension of right-deep trees with a supplement parallel join delivers better performances (using the Wisconsin benchmark executed on a 40 processors shared nothing system) than right-deep trees. However this work has never considered query optimization, nor the complete LBT space.

Let us finally remark that we recognized support for our original LBT strategy in the works of W. Hong for the XPRS shared memory prototype. He has claimed in [38] that running one CPU and one I/O bound tasks in parallel can use the available resources more efficiently. However, W. Hong has not considered a classification of the search space, nor any optimization issues.

4 Search Space Sizes

In this Section, we calculate *lower* and *upper* bounds for the LBT space and compare them to that of the LT and BT space. In order to compute the lower bounds, the submitted query must be a *string query*, and for the upper bounds a *clique query*. Fig. 3 shows both query types. In a clique query, the join graph is completely connected. In a string, the join graph is linear. The following identifiers denote the upper and lower bounds in dependency of n, the number of participating relations:

$\mathbf{v_n}$ upper bound and $\mathbf{w_n}$ lower bound for the LT space
$\mathbf{x_n}$ upper bound and $\mathbf{y_n}$ lower bound for the BT space
$\mathbf{r_n}$ upper bound and $\mathbf{s_n}$ lower bound for the LBT space.

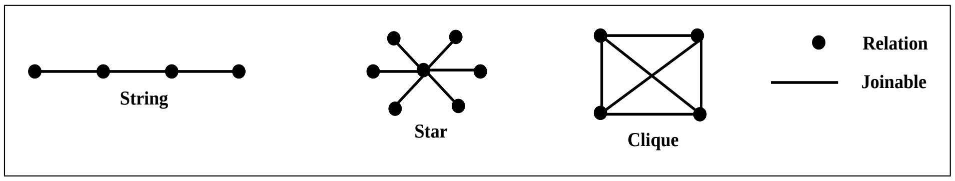

Fig. 3. *Query types: string, star and clique.*

In order to calculate the lower and upper bounds, the following representation must be introduced. Each *bushy tree* that identifies the order of evaluating the joins in a multi-way join query of n participating relations can be represented as a triple (T_k, r, T_{n-k}) where r is a distinguished node called the join result of the complete bushy tree T_n. T_k is the processing tree of k relations for some $1 \leq k \leq n-1$, called the *left bushy subtree* of T_n ; T_{n-k} the *right bushy subtree* of T_n.

Known upper and lower bounds
The number of possible BT orderings $\mathbf{x_n}$ for *clique queries* with n relations expresses by the recurrence relation [39] ($x_1 = 1$): $x_n = \sum_{k=1}^{n-1} \binom{n}{k} x_k x_{n-k}$. The solution of this recurrence is [8]:

$$x_n = \frac{(2n-2)!}{(n-1)!}.$$

The number of possible BT orderings, $\mathbf{y_n}$, for *string queries* expresses as the recurrence: $y_n = \sum_{k=1}^{n-1} 2 \cdot y_k y_{n-k}$. The expression explains as the string can be partitioned into $n-1$ places, but either partition may be left child. The relationship $y_n = 2^{n-1} \cdot x_n / n!$ holds, which may be verified by substitution in the recurrence. The solution of this recurrence is [8]:

$$y_n = \frac{2^{n-1}}{n} \cdot \binom{2n-2}{n-1}.$$

The number of possible LT orderings, $\mathbf{v_n}$, for *clique queries* computes as [8]:

$$v_n = n!\ 2^{n-2},$$

as there exists 2^{n-2} different tree forms modulo the join permutations.

The number of possible LT orderings, $\mathbf{w_n}$, for *string queries* computes, based on the number of possible string partitions, as [8]:

$$w_n = 2^{2n-3}.$$

Upper and lower bounds for linear-oriented bushy trees
We will calculate the upper and lower bounds for the space size of the LBTs based on the introduced identification of an LBT as a triple (T_k, r, T_{n-k}), where r is a distinguished node called the join result of the complete T_n.

The BT space and the LT one distinguishes only from $n \geq 6$. The first BT which is not an LBT can be constructed, when the last processed join has two input subtrees, each containing two joins. This is the case for $n \geq 6$ (refer to Section 2). The number of possible LBT orderings $\mathbf{r_n}$ for *clique queries* in the case $n < 6$ is, thus, identical to $x_n \Rightarrow r_1 = 1, r_2 = 2, r_3 = 12, r_4 = 120, r_5 = 1680$.

The number of possible **LBT** orderings for *clique queries*, $\mathbf{r_n}$ for $n \geq 6$, calculates as a recurrence:

$$r_n = \binom{n}{1} r_1 r_{n-1} + \binom{n}{2} r_2 r_{n-2} + \binom{n}{n-1} r_{n-1} r_1 + \binom{n}{n-2} r_{n-2} r_2$$

The first term $\binom{n}{1} r_1 r_{n-1}$ refers to a processing tree, where the last join has as left input a base relation, where in the third term $\binom{n}{n-1} r_{n-1} r_1$, it has as right input a base relation. For the second term $\binom{n}{2} r_2 r_{n-2}$, the last processed join has one join to the left and for the fourth term $\binom{n}{n-2} r_{n-2} r_2$ one to the right. The expression facilitates with $r_1 = 1$ and $r_2 = 2$ to:

$$r_n \;=\; 2nr_{n-1} + 2n(n-1)r_{n-2}$$

The closed formulae can be obtained by applying a well-known method, see e.g., in [40]:

$r_n \;=\; (n!/2^{n-1}) * (A * (2+2\sqrt{3})^n - B * (2-2\sqrt{3})^n)$ with $A = \frac{40d-224b}{ad-bc}$ and $B = \frac{40c-224a}{bc-da}$, where $a = (2+2\sqrt{3})^4$, $b = (2-2\sqrt{3})^4$, $c = (2+2\sqrt{3})^5$ and $d = (2-2\sqrt{3})^5$.

Approximately: $r_n \;=\; (n!/2^{n-1}) * (0.0458 * (2+2\sqrt{3})^n - 0.1708 * (2-2\sqrt{3})^n)$.

The number of possible **LBT** orderings for *string queries*, $\mathbf{s_n}$, computes as follows. There exists only two possible partitionings of the input relation set into two fully connected partitions of given sizes, therefore, $\mathbf{s_n}$ expresses as $(n \geq 6)$:

$$s_n \;=\; 2s_1 s_{n-1} + 2s_2 s_{n-2} + 2s_{n-1}s_1 + 2s_{n-2}s_2 = 4s_{n-1} + 8s_{n-2},$$

with the initial values: $s_1 = 1$,$s_2 = 2$,$s_3 = 8$,$s_4 = 40$,$s_5 = 224$.

The solution of this recurrence is $(n \geq 6)$:

$s_n \;=\; A * (2+2\sqrt{3})^n - B * (2-2\sqrt{3})^n$ with $A = \frac{40d-224b}{ad-bc}$ and $B = \frac{40c-224a}{bc-da}$, where $a = (2+2\sqrt{3})^4$, $b = (2-2\sqrt{3})^4$, $c = (2+2\sqrt{3})^5$ and $d = (2-2\sqrt{3})^5$.

Approximately: $s_n \;=\; 0.0458 * (2+2\sqrt{3})^n - 0.1708 * (2-2\sqrt{3})^n$.

Comparing the computed formulae for s_n, r_n on the one side, and for v_n, w_n, x_n, y_n on the other hand, one may derive the following Property:

Property 1: The following holds true for $n \geq 1$:

$$\frac{s_n}{r_n} = \frac{v_n}{w_n} = \frac{x_n}{y_n} = \frac{n!}{2^{n-1}}$$

Proof: Compute the respective ratios and compare them.

Summary

Table 1 summarizes the formulae for the different space size bounds $(n \geq 6)$ and shows respective orders of magnitudes for the lower bounds. Table 1 demonstrates clearly that the LBT search space size is in between that of the LT and the BT one, and that it contributes, therefore, to the bridging of the gap between these spaces.

	Lower Bound	Order for Lower Bound	Upper Bound
LT	2^{2n-3}	$O(4^n)$	$n!\ 2^{n-2}$
LBT	$0.0458 * (2+2\sqrt{3})^n$	$O((2+2\sqrt{3})^n)$	$(n!/2^{n-1}) * (0.0458 * (2+2\sqrt{3})^n$
	$-0.1708 * (2-2\sqrt{3})^n$	$\approx O((5.4641)^n)$	$-0.1708 * (2-2\sqrt{3})^n)$
BT	$\frac{2^{n-1}}{n} \cdot \binom{2n-2}{n-1}$	$O(8^n)$	$\frac{(2n-2)!}{(n-1)!}$

Table 1
Upper and lower bounds of the LBT and BT spaces.

5 Enumerative Search Technique

Enumerative search techniques are a broadly used strategy for complex parallel query optimization. For instance, they are employed in Microsoft SQL Server [1]. This technique explores a search space by using transformation rules, i.e., apply all possible rules on each alternative (a group of processing trees) and terminate when no new alternatives can be produced. A memoizing structure, the so-called *MEMO-structure*, has been proposed in [1] to improve the efficiency of the enumerative search technique [41,42]. The main idea of the MEMO-structure is to avoid replication of subtrees involving the same set of relations. The definition of the MEMO-structure is given below.

Definition 4: The **MEMO-structure** for a query Q is organized as a network of equivalence *groups* (shortly groups). Each group is a set of logical equivalent *multi-expressions* which generate the same intermediate results. A multi-expression is an operator whose two operands are groups. *Logical equivalence* of two multi-expressions is defined as logical equivalence of any processing tree corresponding to these multi-expressions. We assume that two processing trees are logical equivalent if they involve the same set of relations.

For convenience, the groups are labeled with the relations that are being joined. The simplest group involves only a base relation and its processing tree is the base relation itself. The process of generating all multi-expressions in a group is called the *group enumeration.*

Example: Throughout the remainder of the paper we use as example a multi-way join query over the TPC-D [4] benchmark schema (related to the Q5 Local Supplier Volume Query):

[4] `http://www.tcp.org/`. Let us note that for the time being of this habilitation, November 2001, the TPC-D is no longer supported by the Transaction Processing Performance Council. However, its successor, TPC-H, includes Query Q5 again.

```
SELECT N_Name, sum(L_extendedprice*(1-L_discount)) as revenue
FROM Customer C_,Order O_,Suppliers S_,Lineitem L_,Nation N_,Region R_
    WHERE O_.Custkey = C_.Custkey AND
        L_.Orderkey = O_.Orderkey AND L_.Suppkey = S_.Suppkey AND
        S_.Nationkey = N_.Nationkey AND N_.Regionkey=R_.Regionkey AND
        C_.Name='customer' AND R_.Name='region'
    GROUP BY N_Name
```

This query lists the revenue volume (sum(L_extendedprice*(1-L_discount)) as revenue) that resulted from lineitem transactions in which a certain customer orders parts and the part supplier fills them within a certain region. For further considerations, we discard the *group by* and the *selection* operations and concentrate on the *multi-way join* part. The corresponding join graph is cyclic with $S\_, L\_, O\_, C\_$ arranged in a square, and $R\_, N\_$ being connected as a string to the $S\_$. Table 2 displays the interesting parts of the MEMO-structure for n, the number of relations to be joined. We display the cases $n = 3, 4$ and 6.

n	**Enumerated Groups**	
3	$[S\_L\_O\_]=$	$\{[S\_L\_] \bowtie [O\_],\ [O\_] \bowtie [S\_L\_], [L\_O\_] \bowtie [S\_],\ [S\_] \bowtie [L\_O\_]\}$
	$[L\_O\_C\_]=$	$\{[L\_O\_] \bowtie [C\_],\ [C\_] \bowtie [L\_O\_], [O\_C\_] \bowtie [L\_],\ [L\_] \bowtie [O\_C\_]\}$
	$[L\_S\_C\_]=$	$\{[S\_L\_] \bowtie [C\_],\ [C\_] \bowtie [S\_L\_], [S\_C\_] \bowtie [L\_],\ [L\_] \bowtie [S\_C\_]\}$
	$[L\_S\_N\_]=$	$\{[S\_L\_] \bowtie [N\_],\ [N\_] \bowtie [S\_L\_], [S\_N\_] \bowtie [L\_],\ [L\_] \bowtie [S\_N\_]\}$
	$[R\_S\_N\_]=$	$\{[N\_R\_] \bowtie [S\_],\ [S\_] \bowtie [N\_R\_], [S\_N\_] \bowtie [R\_],\ [R\_] \bowtie [S\_N\_]\}$
	$[C\_S\_N\_]=$	$\{[S\_N\_] \bowtie [C\_],\ [C\_] \bowtie [S\_N\_], [S\_C\_] \bowtie [N\_],\ [N\_] \bowtie [S\_C\_]\}$
4	$[S\_L\_O\_C\_] =$	$\{[S\_L\_O\_] \bowtie [C\_],\ [C\_] \bowtie [S\_L\_O\_], [L\_O\_C\_] \bowtie [S\_],$ $[S\_] \bowtie [L\_O\_C\_], [L\_S\_C\_] \bowtie [O\_],\ [O\_] \bowtie [L\_S\_C\_],$ $[O\_C\_] \bowtie [S\_L\_],\ [S\_L\_] \bowtie [O\_C\_], [L\_O\_] \bowtie [S\_C\_], [S\_C\_] \bowtie [L\_O\_]\}$
	$[C\_S\_N\_R\_] =$	$\{[C\_S\_N\_] \bowtie [R\_],\ [R\_] \bowtie [C\_S\_N\_], [R\_N\_S\_] \bowtie [C\_],$ $[C\_] \bowtie [R\_N\_S\_], [N\_R\_] \bowtie [S\_C\_],\ [S\_C\_] \bowtie [N\_R\_]\}$
	$[L\_S\_N\_R\_] =$	$\{[L\_S\_N\_] \bowtie [R\_],\ [R\_] \bowtie [L\_S\_N\_], [R\_N\_S\_] \bowtie [L\_],$ $[L\_] \bowtie [R\_N\_S\_], [N\_R\_] \bowtie [S\_L\_],\ [S\_L\_] \bowtie [N\_R\_]\}$
5	$[S\_L\_O\_C\_N\_] =$	$\ldots$
	$[S\_C\_O\_R\_N\_] =$	$\ldots$
	$[S\_L\_O\_R\_N\_] =$	$\ldots$
6	$[S\_L\_O\_C\_N\_R\_] =$	$\{[S\_L\_O\_C\_N\_] \bowtie [R\_], [R\_] \bowtie [S\_L\_O\_C\_N\_], [S\_C\_O\_R\_N\_] \bowtie [L\_],$ $[L\_] \bowtie [S\_C\_O\_R\_N\_], [S\_L\_O\_R\_N\_] \bowtie [C\_], [C\_] \bowtie [S\_L\_O\_R\_N\_],$ $[S\_L\_O\_C\_] \bowtie [N\_R\_], [N\_R\_] \bowtie [S\_L\_O\_C\_], [C\_S\_N\_R\_] \bowtie [L\_O\_],$ $[L\_O\_] \bowtie [C\_S\_N\_R\_], [L\_S\_N\_R\_] \bowtie [C\_O\_], [C\_O\_] \bowtie [L\_S\_N\_R\_]\}$

Table 2
MEMO-structure for the example query ($n > 2$).

The MEMO-structure of the LBT space differs from the LT space starting with $n = 4$, e.g., the group $[S\_L\_O\_C\_]$ contains multi-expressions where both input groups involve two relations, such as $[S\_C\_] \bowtie [L\_O\_]$. For $n = 6$ the LBT and BT space differs the first time. The group $[S\_L\_O\_C\_N\_R\_]$ does not contain the multi-expressions $[L\_O\_C\_] \bowtie [S\_N\_R\_]$ and $[S\_N\_R\_] \bowtie [L\_O\_C\_]$, because their corresponding processing trees are not LBTs (the two input relations of the final join are neither a base relation nor a single join).

The MEMO-structure encoding a space is build by enumerating recursively the input groups of all multi-expressions of the group actually under consideration. Enumerating a group is done by applying transformation rules to generate all alternative multi-expressions starting from an initial one. We are aware of at least three parallel database systems which use this enumerative strategy: the commercial Microsoft SQL Server [1] and Tandem ServerWare [43] and the prototype MIDAS, developed at the Technical University Munich and University Stuttgart [13].

Fig. 4 displays the core strategy of the exploration algorithm. It is composed of a main program and a recursive procedure *FindOptimal()*. The MEMO-structure is a global structure to the main program and *FindOptimal()*.

```
/* PROCEDURE FINDOPTIMAL(GROUP) */
FindOptimal($group$)
if $group$ not already visited then
/* GENERATE ALTERNATIVES WITH A DUPLICATE-FREE TRANSFORMATION SET */
      foreach enumerated $mexpr$ of $group$
            $mexpr = [left\_group] \bowtie [right\_group]$
            if number of relations in $left\_group > 1$ then
                  $FindOptimal(left\_group)$ end(if)
            if number of relations in $right\_group > 1$ then
                  $FindOptimal(right\_group)$ end(if)
            end(foreach)
            write enumerated $group$ to the MEMO-structure;
end(if)
end(FindOptimal)

/* MAIN: ENUMERATE GROUPS INVOLVING ONE RELATION */
$Groups1 = \{[R_1], \ldots, [R_n]\}$ ;
Initialize the MEMO structure with $Groups1$;
/* CALL OF THE RECURSIVE FINDOPTIMAL */
$FindOptimal([R])$;
end
```

Fig. 4. *The core search strategy of the exploring algorithm.*

The principle task of the main program is to enumerate groups containing only a base relation. Afterwards, the main program calls $FindOptimal([R])$ where R denotes the set of participating relations. The procedure *FindOptimal(group)* exploits the input *group* by calling itself recursively for the input-groups of each generated multi-expression until the trivial case (group contains only a base relation) is reached. In order to avoid multiple visits of the same multi-expression, the group enumeration processes must be realized by a duplicate-free transformation-rule system. Such a rule system for LBTs will be developed in Section 6.

The major difference in the exploring processes for LBTs and BTs is the number of recursive calls of *FindOptimal(group)*. In the case of BTs, there are, in general, two recursive calls, one for the right operand and one for the left operand of the actually enumerated multi-expression. However, for LBTs, we have to perform at most *one* recursive call for the multi-expression actually under consideration. Let us see why.

In order to account for LBTs, the main program has to be extended to group enumeration not only involving base relations, but also single joins. The recursive procedure *FindOptimal(group)* is modified in a way that the recursive call for an actual considered group is only performed when its involved number of relations is greater than 2. As for LBTs, only one of the two input groups can contain more than two relations, consequently, only one recursive call is required.

Example: Let us consider the enumeration of $[P\_L\_O\_C\_]$ for LBTs. The main program first enumerates the groups containing one relation and then those containing single joins. With respect to the connectivity of the join graph, we first obtain $[P\_] = P\_$, $[L\_] = L\_$, $[O\_] = O\_$ and $[C\_] = C\_$, and second $[P\_L\_] = \{[P\_] \bowtie [L\_], [L\_] \bowtie [P\_]\}$, $[L\_O\_] = \{[L\_] \bowtie [O\_], [O\_] \bowtie [L\_]\}$ and $[O\_C\_] = \{[O\_] \bowtie [C\_], [C\_] \bowtie [O\_]\}$.

Let the initial processing tree be $((P\_ \bowtie L\_) \bowtie O\_) \bowtie C\_$. The main program calls $FindOptimal([P\_L\_O\_C\_])$. As the group has not been visited yet, we start by exploiting the multi-expression which represents the initial processing tree: $[P\_L\_O\_] \bowtie [C\_]$. The procedure $FindOptimal$ is recursively called on the input group $[P\_L\_O\_]$. This group has not yet been visited and we generate a first multi-expression $[P\_L] \bowtie [O\_]$. As both input groups contain only a base relation or a single join, the recursion terminates.

The next enumerated multi-expressions are $[O\_] \bowtie [P\_L\_]$, $[L\_O\_] \bowtie [P\_]$ and then $[P\_] \bowtie [L\_O\_]$ for which the recursion terminates immediately as their operands contain not more than two relations. Now all multi-expressions in $[P\_L\_O\_]$ are enumerated. The next multi-expression to be enumerated for the group $[P\_L\_O\_C\_]$ is $[C\_] \bowtie [P\_L\_O\_]$. Applying a join exchange leads to the following multi-expression $[L\_O\_C\_] \bowtie [P\_]$. Here, we should call recursively *FindOptimal* on the left-input group $[L\_O\_C\_]$. In this recursive call we will enumerate the multi-expressions: $\{[L\_O\_] \bowtie [C\_],\ [C\_] \bowtie [L\_O\_],\ [O\_C\_] \bowtie [L\_],\ [L\_] \bowtie [O\_C\_]\}$. Finally we consider the multi-expression $[P\_] \bowtie [L\_O\_C\_]$ for enumeration. $[L\_O\_C\_]$ has been already enumerated, thus the recursion terminates and $[P\_L\_O\_C\_]$ is completely enumerated.

The described control flow leads to the following call-hierarchy of $FindOptimal$.

Starting with $[P\_L\_O\_C\_] \rightarrow [P\_L\_O\_] \rightarrow [P\_L\_O\_](again) \rightarrow [L\_O\_C\_] \rightarrow [L\_O\_C\_](again)$. Thus, we have in all five calls, and we visit two groups twice.

6 Duplicate-Free Transformation Set for LBT Group Enumeration

In order to achieve efficiency, transformation-based query optimizers should implement a *duplicate-free* transformation rule set for the generation of all multi-expressions in a group, i.e., each multi-expression is visited only once during enumeration. Pellenkoft et al. [41] have shown that the application of the join associativity rule (for BTs and LTs) let the number of duplicates outgrow the number of unique multi-expressions already for $n > 4$. Thus, we have to develop a duplicate transformation set for the LBT group enumeration.

Transformation rules for group enumeration are described as left-hand side multi-expression (input) to right-hand side (output) with a rule condition when to fire the transformation. The transformation rule describes a mapping from an input multi-expression to a set of output multi-expressions by the following means. For each binding of the term variables which satisfies the input multi-expression, an output multi-expression is generated. The transformation-rule system for group enumeration performs as follows: pick out an initial valid multi-expression of the group actually under consideration and then apply all transformations where the left-hand side matches the initial multi-expression and the rule condition is true. Apply the transformation rules for all results of the former process until no more rules can be applied.

Example: Let G_1, G_2, G_3 be three enumerated groups and let the sample transformation rule be the left join associativity rule taken from the set of rules generating the BT space. The rule expresses as: $G_1 \bowtie (G_2 \bowtie G_3) \rightarrow (G_1 \bowtie G_2) \bowtie G_3$. Let $R = \{S\_, L\_, O\_, C\_\}$, i.e., a subset of the relations involved in our example query. The left-associativity rule applied to $[S\_L\_] \bowtie [O\_C\_]$ generates two multi-expressions $[S\_L\_C\_] \bowtie [O\_]$ and $[S\_L\_O\_] \bowtie [C\_]$, as each split of the right input group $[O\_, C\_]$, i.e., $[O\_] \bowtie [C\_]$ and $[C\_] \bowtie [O\_]$, is a valid binding.

Duplicate free rule sets:
Let the initial multi-expression have as left operand a group containing only one relation. Two rule sets have to be distinguished, one for $n < 6$ and one for $n \geq 6$. The LBT transformation rules for $n < 6$ are equal to the BT ones [41]:

Swap: $G_1 \bowtie_0 G_2 \rightarrow G_2 \bowtie_1 G_1$
with G_1, G_2 already enumerated groups and the rule condition: Disable *Swap* and *Left Join Associativity* on the new operator $\bowtie_1$.

Left Join Associativity: $G_1 \bowtie (G_2 \bowtie G_3) \rightarrow (G_1 \bowtie G_2) \bowtie G_3$
with G_1, G_2, G_3 already enumerated groups.

For $n \geq 6$, four different transformation rules are required:

Rule 1: Swap: $G_1 \bowtie_0 G_2 \rightarrow G_2 \bowtie_1 G_1$
with G_1, G_2 already enumerated groups. Disable Rules 1,2,3,4 on the new operator $\bowtie_1$.

Rule 2: Left Join Associativity1: $[R_i] \bowtie ([R_j] \bowtie G) \rightarrow ([R_i] \bowtie [R_j]) \bowtie G$
with R_i, R_j base relations and G an already enumerated group and the rule condition that $[R_i]$ and $[R_j]$ are joinable .

Rule 3: Left Join Associativity2: $[R_i] \bowtie (G \bowtie [R_j]) \rightarrow ([R_i] \bowtie G) \bowtie [R_j]$
with R_i, R_j base relations and G an already enumerated group and the rule condition that $[R_i]$ and G are joinable.

Rule 4: Left Join Associativity3: $[R_i] \bowtie (G \bowtie ([R_j] \bowtie [R_k])) \rightarrow ([R_i] \bowtie G) \bowtie ([R_j] \bowtie [R_k])$
with R_i, R_j, R_k base relations and G an already enumerated group and the rule condition that $[R_i]$ and G are joinable.

The specified rule conditions ensure that the complete LBT space is spawn and that no duplicates are generated. In the following, we will proof these properties for the case of clip queries. For acyclic queries, the correctness of the rule set can be shown in a similar way.

Lemma: The rule set generates for cliques only valid multi-expressions in the LBT space without duplicates.

Proof. For the case $n < 6$ we refer to Pellenkoft's works [41] (including the completeness proof). We consider hereafter $n \geq 6$.
For the given input multi-expression $[R_i] \bowtie G$, with R_i a base relation and G a group, all four rules are applicable. Rule 1 generates an output multi-expression where the right operand contains only one relation. All rules are disabled on the newly generated multi-expression. Rule 2 generates multi-expressions from the initial expression where the left operand involves a single join containing as left input the base relation R_i. None of Rules 2,3,4 can be applied to the output multi-expressions, as they require that the left operand contains a base relation. Now, by applying Rule 1 on these multi-expressions, the generation process stops for this case. Rule 3 generates multi-expressions where the right operand contains a base relation. This relation cannot be R_i. As the left operand contains at least a single join none of Rules 2,3,4 applies. Rule 1 generates the mirror images and the generation process stops for this case. No duplicates are generated, as the right input of the mirrors is not R_i. Finally, Rule 4 generates multi-expressions where the right operand contains a

single join. This single join does not contain the relation R_i. Rule 1 generates once again the mirror images and the generation process stops for this case. No duplicates are generated, as the right input of the mirrors does not contain the base relation R_i. ■

Theorem 1: The presented rule set applied to an initial multi-expression where the left operand contains only one relation, enumerates correctly groups for cliques.

Proof. In a fully enumerated group that references n relations, the number of multi-expressions is $n^2 + n$ (see below). With respect to the lemma above, it shall only be proven that our rule set generates exactly n^2+n multi-expressions including the initial multi-expression.

From the initial multi-expression, where the left operand contains only one relation, Rule 1 generates one mirror image. Rule 2 generates $n-1$ new multi-expressions (possible join combinations with R_i), counting the further generated mirrors we obtain $2(n-1)$. Rule 3 generates also $n-1$ new multi-expressions (the base relations excluding R_i), with the mirrors we count $2(n-1)$. Rule 3 generates $\binom{n-1}{2}$ new multi-expressions (possible join combinations excluding R_i), counting the mirrors and rewriting leads to $(n-1)(n-2)$. Summing up the number of newly created multi-expressions plus the initial multi-expression we obtain: $1+2(n-1)+2(n-1)+(n-1)(n-2)+1$. Rewriting shows that this is equal to n^2+n. ■

7 Size of the MEMO-Structure

In this Section, we compute a lower (string queries) and an upper (cyclic queries) bound for the size of the MEMO-structure encoding the LBT space, and then compare these sizes to respective sizes for BTs and LTs. These considerations are important, because the size of the MEMO-structure is an indicator for the computational complexity of the enumerative search technique. Exact computational models depend on the utilized data-structures to realize the MEMO-structure [44].

Theorem 2, known from related works [41], summarizes lower and upper bounds for the MEMO-structure size encoding the BT and LT space. Theorem 3 is a new contribution. It computes lower and upper bounds for the MEMO-structure size for LBTs. We will show that the MEMO-structure size for LBTs is in the same order of magnitude as for LTs and significantly smaller than for BTs. The experimental results confirm these theoretical results: the running time of our optimizer, when LBTs are considered, is close to the running time required for optimizing LTs.

Theorem 2: The size of the MEMO-structure considering BT for a clique is: $3^n - 2^{n+1} + n + 1$ ($O(3^n)$) and for a string is: $(n^3 - n)/3 + n$ ($O(n^3)$). The size of the MEMO-structure considering LT for a clique is: $n2^n - n$ ($O(2^n)$) and for a string is: $2n^2 - 6n - 4$ ($O(n^2)$).

Theorem 3: The size of the MEMO-structure considering LBT ($n > 6$) for a clique is: $2n2^{n-1} + n(n-1)2^{n-2} - n - 2n(n-1) - n(n-1)(n-2) - 1/4n(n-1)(n-2)(n-3)$ ($O(2^n)$) and for a string is: $4n^2 - 15n + 20$ ($O(n^2)$).

Proof. Let us start with the cliques and denote by $m(k)$ the number of multi-expressions enumerated for all groups containing k ($1 < k \leq n$) relations. Since each possible subset of R (the set of base relations) containing k relations will occur in a group, the number of generated groups is $\binom{n}{k}$.

How many multi-expressions are generated now in each of these groups ?

We must distinguish three cases, $k \geq 6$ (LBTs differs from BTs), $1 < k < 6$ and the special case $k = 1$.
For $k \geq 6$, $m(k)$ is equal to the number of partitions of all possible subsets of R containing k relations into a left-and right non-empty subset, where one of the subsets contains exactly one or two base relations. $m(k)$ evaluates as $\binom{n}{k}(k^2 + k)$.
For $1 < k < 6$, $m(k)$ is equal to the number of partitions of all possible subsets of R containing k relations into a left-and right non-empty subset. $m(k)$ evaluates as $\binom{n}{k}(2^k - 2)$.
$m(1)$ evaluates as n.
Summing up all the partial results, we obtain: $\sum_{k=2}^{n} m(k) + m(1)$. Rewriting results in $2n2^{n-1} + n(n-1)2^{n-2} - n - 2n(n-1) - n(n-1)(n-2) - 1/4n(n-1)(n-2)(n-3)$.

Let us now consider the case of strings. As for cliques, we first calculate the number of groups containing k relations ($1 < k \leq n$). Each string with n relations can have exactly $n - k + 1$ substrings containing k relations. Second, we compute the number of multi-expressions $m(k)$ enumerated for all groups containing k ($1 < k \leq n$) relations.

We must again distinguish three cases, $k \geq 6$ (LBTs differs from BTs), $1 < k < 6$ and the special case $k = 1$.
For $k \geq 6$, $m(k)$ computes as the number of partitions of all strings containing k relations in left- and right subsets, where one of the subsets contains exactly one or two base relations. $m(k)$ evaluates as $8(n - k + 1)$.

For $1 < k < 6$, there are $2(k-1)$ partitions of all the possible strings containing k relations in non-empty left- and right subsets (remember each relation except the terminal ones are connected to two relations in the string). Thus, $m(k)$ evaluates as $2(k-1)(n-k+1)$.
$m(1)$ evaluates again as n.
Summing up all the partial results, we obtain: $\sum_{k=2}^{n} m(k) + n$. Rewriting results in $4n^2 - 15n + 20$. ■

8 Experimental Results

This Section summarizes the significant part of the experiments we performed in order to evaluate the impact of the LBT space. We compare the quality of the generated execution plans and the running time of the optimizer (shortly optimization time) for LTs, LBTs and BTs. A prototype top-down optimizer for join enumeration, using a parallel cost function derived from [24], has been implemented. The optimization strategy bases on the enumerative exploring algorithm proposed in Section 5 and is, for instance, used in the Microsoft SQL Server [1].

In addition to the basic algorithm, each group is augmented with information about *optimality* with respect to the parallel cost function. For each group, *one* optimal multi-expression is retained (as it is done in similar projects [42]), as well as its cost and its corresponding processing tree. Therefore, all multi-expressions correspond to one unique processing tree. Hence, at the end of the optimization process, when the group containing all participating relations has been enumerated, only one processing tree, the best, has been selected. Additional and efficient pruning is achieved by maintaining an upper-bound on the multi-expression cost and discarding multi-expressions of the actually enumerated group which exceed this bound.

8.1 Experimental Settings

The test reported here are executed on a PC with an Intel III processor, 450 MHz, 128 MB memory, running Linux. The query optimizer is implemented in JAVA using jdk1.2 (JIT-compiler included).

Parameter Settings In all, *1800* different *queries* are run. The queries vary in two basic parameters: in the *number of relations (6,8,10,12,14 and 16)* and in the *shape of the join graph (string, one-cycle and two-cycle)*. The cyclic graphs are realized once as a one-cycle and once as a two-cycle graph. A graph of type one-cycle contains one cycle, and the rest of the graph is unspecified with the constraint that globally only a single cycle exits, respective definition for the two-cycle graph. Such query graphs shapes are representative for a wide range of applications [45]. For each parameter setting, we generate 100 different queries with different join selectivities (uniform distribution between 0 and 1) and different relation cardinalities (between 1 and 1000 MB), thus in all 6x3x100=1800 queries are run.

Cost Function The realized parallel cost function bases on the intra-operator computation model of the PRISMA parallel database [24]. The degree of intra-operator parallelism for a relation (R) is calculated as $\sqrt{\frac{|R|c}{a}}$, where c is an estimation of the processing time per tuple, a is an estimation of the initialization time per processor and $|R|$ denotes the cardinality of R, i.e., the number of tuples in R. The degree of inter-operator parallelism is given by the shape of the actual considered processing tree of the optimization algorithm. The parallel cost model assumes uniform relation partitions over processors and considers no pipeline parallelism. Furthermore we suppose a shared disk model. For simplicity of the cost function, we suppose that the intermediate relations are reloaded for processing from the shared disk. Therefore, no communication costs are encountered and well balanced partition sizes can be assumed.

Our parallel cost function is a simple, but commonly employed model [4,10]. The cost of a processing tree is computed recursively over the tree structure. Let $PT_1 \bowtie PT_2$ be the actually considered join operator with its left input subtree PT_1 and its right input subtree PT_2. Furthermore denote by $|PT_1 \bowtie PT_2|$ the cardinality of the intermediate relation generated by this join. The parallel cost of this join $cost_{par}(PT_1 \bowtie PT_2)$ is then computed as (with d the degree of intra-operator parallelism as computed above): $cost_{par}(PT_1 \bowtie PT_2) = |PT_1 \bowtie PT_2|/d + max(cost_{par}(PT_1), cost_{par}(PT_2))$. The trivial case is $cost_{par}(R) = 0$ for some base relation R.

8.2 Performance Evaluation

The performance evaluation of the LBT is given in two parts. In the first part, we will demonstrate that the quality of the generated LBT execution plan is almost as good as the quality of the generated BT plan and significantly better than the quality of the generated LT plan. In the second part, we will show that the mean optimization time for LBTs is close to the optimization time for LTs.

Comparison of the Execution Plan Costs Fig. 5 gives the execution cost ratio LBT/BT for all three query graph shapes and for $6 \leq n \leq 16$. Fig. 6 displays the execution cost ratio LT/LBT. Each point on the curve represents the average of 100 experiments.

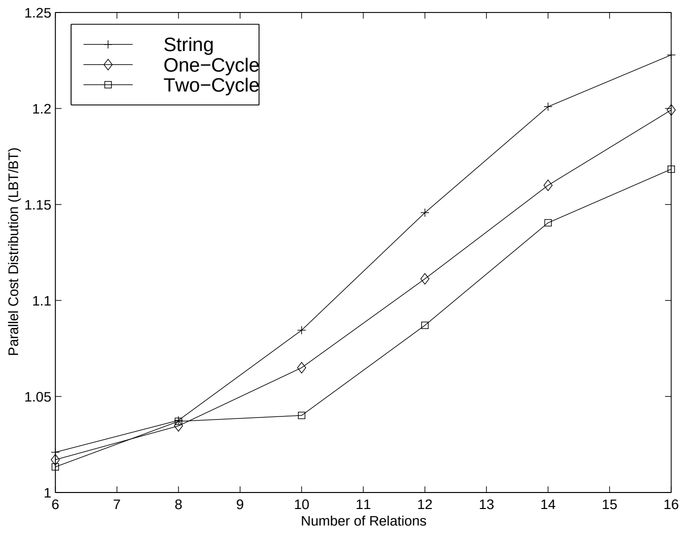

Fig. 5. *Execution Plan Cost Ratio LBT/BT.*

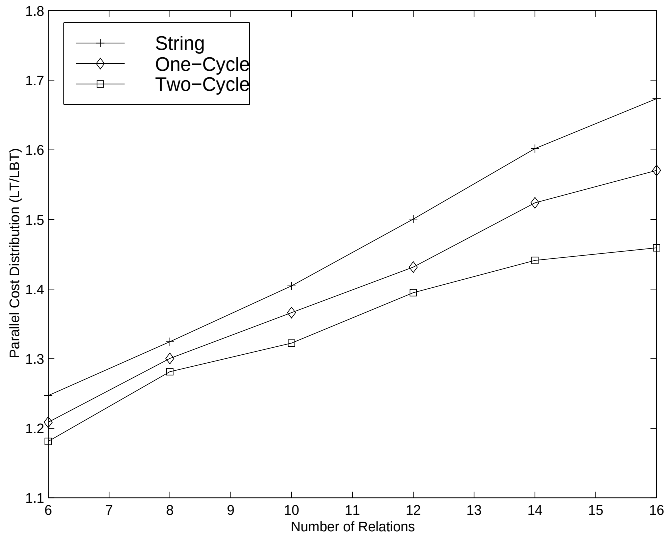

Fig. 6. *Execution Plan Cost Ratio LT/LBT.*

One sees clearly that the costs of the generated LBT execution plans are only slightly above the costs of the BT ones. The mean cost ratio LT/LBT for strings is 1.119, for one-cycles 1.098 and for two-cycles 1.081. The reason for the good quality of LBT execution plans compared to BT ones is the higher load imbalance introduced in BT. We measured that the sum of time differences in the input relations arrival for all joins in BT was on average 10% higher than in LBT.

If we compare LBT execution plan costs with those of LT plans (see Fig. 6), a significant higher cost difference, than between BT and LBT plans, can be remarked. The mean cost ratio LT/LBT for strings is 1.459, for one-cycles is 1.400 and for two-cycles is 1.347.

Comparison of the Optimization Times Let us compare now the optimization times for LTs, LBTs and BTs. Fig. 7 displays the mean optimization time for strings and Fig. 8 shows the mean optimization time for two-cycles (over 100 query runs). One-cycle queries show similar characteristics as the two-cycles ones and results are, therefore, omitted.

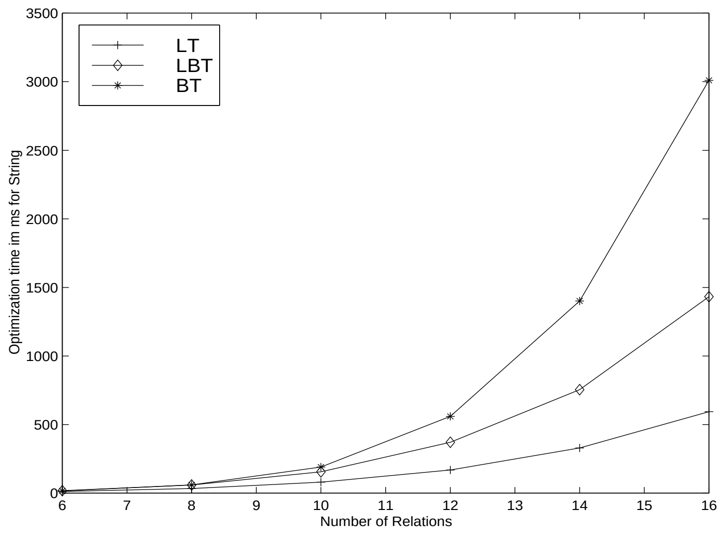

Fig. 7. *Optimization Times in Milliseconds for Strings.*

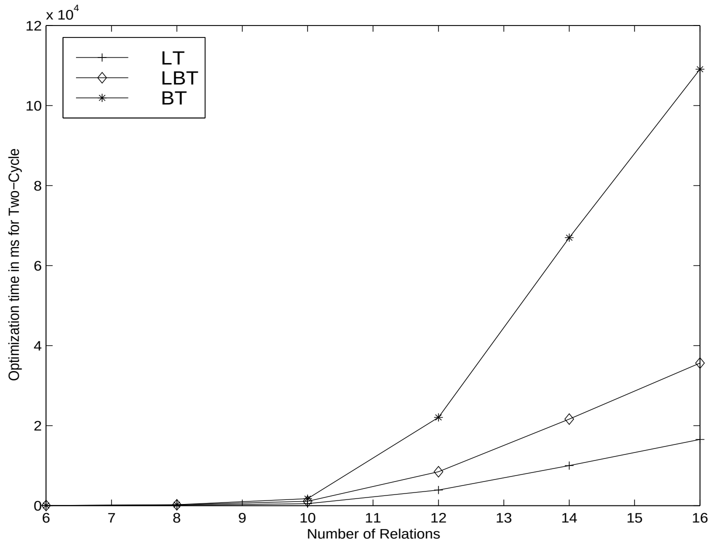

Fig. 8. *Optimization Times in Milliseconds for Two-cycles.*

The optimization times for LTs, LBTs and BTs remain small for query complexities $\leq$ 10 relations. For queries involving more than 10 relations, the distinction between the spaces becomes important:

- for 14 relations, the BT optimization time for strings remains small, i.e., under 1.5 seconds, but the optimization time for two-cycles is already above 1 minute and is clearly more than double the LBT optimization time. The optimization time ratio LBT/LT flavors at two and this for all query graph shapes ;

- for 16 relations, the BT optimization time shows a sharp increase, e.g., for cyclic queries the optimization time is more than 3 times higher than the LBT optimization time. The BT optimization time reaches for two-cycles almost 2 minutes. This expects even higher BT optimization times for more complex queries.

In summary, considering LBTs, approximately, double the optimization time compared to LTs, however they reduce the execution plan costs by about 40%. Considering LBTs instead of BTs let fall the costs of the generated plans slightly, in mean by about 10%, however let reduce the optimization time considerably, in mean by about a factor of 3 for query complexities > 10 (in the worst case the BT optimization time was nearly 2 minutes, as it was 35 seconds for LBTs).

9 Conclusion and Future Works

The operator ordering is one of the principle tasks of a query optimizer in parallel databases. This task is accomplished by defining an appropriate search space and a search strategy with respect to some parallel cost function. There has been a lot of work invested in search strategies for query optimization (among others enumerative, heuristic and randomized search). However, a structuring of the search space is half-heartened investigated. Research and commercial products rely mainly on two space classes, linear trees (more often left-deep linear trees) and bushy trees. This is astonishing, as the two classes are far away from each others in terms of search space size and search complexity.

In this context, we proposed a new search space, the so-called **linear-oriented bushy trees** (LBT) which contributes to the bridging of the gap. In order to validate the introduction of LBTs, we calculated the different spaces sizes, proposed modifications for an enumerative search strategy to consider LBTs, and finally developed and proved a duplicate-free transformation rule set for LBT group enumeration.

The experimental results, performed on more 1800 queries using an enumerative search strategy, showed that the running time of our optimizer, considering LBTs, is close to the running time of an optimizer that only considers LTs. Furthermore, we noted that the quality of the generated LBT execution plans was, in mean, almost as good as the quality of the best plan found in the space of BTs. These experimental results coincide well with the theoretical results obtained in the first part of this paper.

Future works concerns the integration of the new search space LBT in an available DBMS. We plan to participate at the PostgreSQL Open Source DBMS.

References

[1] Goetz Graefe. The Cascades Framework for Query Optimizatiom. *Bulletin of the IEEE Technical Committee on Data Engineering*, 18(3):19–29, September 1995.

[2] A. Silberschatz, M. Stonebraker, and J. Ullman. Database research: Achievements and opportunities. Into the 21st century. *SIGMOD Record*, 25(1):52–63, March 1996.

[3] Paul Krill. NCR boosts Teradata decision support database. Teradata News, April 1998.

[4] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proceedings of the ACM SIGMOD International Conference of Managment of Data*, pages 35–46, Montreal, Canada, June 1996.

[5] M. T. Özsu and P. Valduriez. *Distributed and Parallel Database Systems*, pages 1093–1111. CRC Press, 1997.

[6] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3):28–33, September 1996.

[7] D. Taniar and Y. Jiang. A high performance object-oriented distributed parallel database architecture. In *HPCN Conference 98*, pages 498–517. Springer Verlag, April 1998.

[8] K.-L. Tan and H. Lu. A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems. *SIGMOD Record*, 20(4):81–82, December 1991.

[9] M. Spiliopoulou, M. Hatzopoulos, and Y. Contronis. Parallel Optimization of Large Join Queries with Set Operators and Aggregates in a Parallel Environment Supporting Pipeline. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):429–445, June 1996.

[10] Leonindes Fegaras. A new heuristic for optimizing large queries. In *International Database and Expert Systems Applications Conference*, pages 726–735, Vienna, Austria, August 1998. Springer Verlag LNCS 1460.

[11] D.D. Straube and M.T. Ozsu. Query optimization and execution plan generation in object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.

[12] H. Lu and K.-L. Tan. Load-Balanced Join Processing in Shared-Nothing Systems. *Journal of Parallel and Distributed Computing*, 23:382–398, 1994.

[13] C. Nippl and B. Mitschang. TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer. In *Proceedings of the International Conference on Very Large Databases*, pages 251–262, New York City (NY), USA, August 1998.

[14] M. Zaït, D. Florescu, and P. Valduriez. Benchmarking the DBS3 Parallel Query Optimizer. *IEEE Parallel and Distributed Technology: Systems and Applications*, 4(2):26–40, 1996.

[15] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *Proceedings of the ACM SIGMOD International Conference of Managment of Data*, pages 365–376, Montral, Canada, June 1996.

[16] N. Kabra and D.J. DeWitt. OPT++ : An Object-Oriented Implementation for Extensible Database Query Optimization. *The VLDB Journal*, 8(1):55–78, 1999.

[17] Yannis E. Ioannidis. *Handbook of Computer Science and Engineering*, chapter Query Optimization, pages 1038–1057. CRC Press, 1997.

[18] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, pages 9–18, San Diego, California, USA, 1992.

[19] M.-S. Chen, M.-L Lo, P.S. Yu, and Y.C. Young. Using Segmented Right-Deep Trees for the Execution of pipelined Hash Joins. In *Proceedings of the International Conference on Very Large Databases*, pages 15–26, Vancouver, BC, Canada, August 1992.

[20] D. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multi-processor database machines. In *Proceedings of the International Conference on Very Large Databases*, pages 469–490, Melbourne, Australia, August 1990.

[21] M. Ziane, M. Zaït, and P. Borla Salamet. Parallel Query Processing with ZigZag Trees. *Very Large Databases Journal*, 2(3):277–301, March 1993.

[22] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proceedings of the International Conference on Database Theory*, Prague, Czech Republic, January 1995. Springer Verlag, LNCS 893.

[23] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *ACM Symposium on Principles of Database Systems*, pages 238–248, Tucson, Arizona, USA, May 1998. ACM Press.

[24] A.N. Wilschut, J. Flokstra, and P.M.G. Apers. Parallel Evolution of Multi-Join Queries. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, pages 115–126, San Jose, California, USA, May 1995.

[25] L. Bouganim, D. Florescu, and P. Valduriez. Load balancing for parallel query execution on NUMA multiprocessors. *Distributed and Parallel Databases*, 7(1):99–121, 1999.

[26] K.A. Hua, M. Lo, and H.C. Young. Considering Data Skew Factor in Multi-Way Join Query Optimization for Parallel Execution. *The VLDB Journal*, 2(6):303–330, March 1993.

[27] H. Kosch, L.Brunie, and W. Wohner. From the modeling of parallel relational query processing to query optimization and simulation. *Parallel Processing Letters*, 8(1):2–14, March 1998.

[28] S. Bonneau and A. Hameurlain. Hybrid simultaneous scheduling and mapping in SQL multi-query parallelization. In LNCS 1677 Springer Verlag, editor, *International Conference on Database and Expert Systems Applications (DEXA)*, pages 88–99, Florence, Italy, August-September 1999.

[29] Goetz Graefe. Iterators, schedulers, and distributed-memory parallelism. *Software - Practice and Experience(SPE)*, 26(4):427–452, April 1996.

[30] M.-S. Cheng and P.S. Yu. Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. In *Proceedings of the International Conference on Data Engineering*, pages 58–67, February 1992.

[31] M. Mehta and D.J. DeWitt. Managing Intra-operator Parallelism in Parallel Databases. In *Proceedings of the International Conference on Very Large Databases*, pages 382–394, Zurich, Switzerland, September 1995.

[32] E. Rahm and R.Marek. Dynamic Multi Resource Load Balancing in Parallel Database Systems. In *Proceedings of the International Conference on Very Large Databases*, pages 395–406, Zurich, Switzerland, September 1995.

[33] A. Hameurlain and F. Morvan. Scheduling and Mapping for Parallel Execution of Extended SQL Queries. In *ACM CIKM 95*, pages 197–204, Baltimore, MD, USA, November 1995.

[34] C.K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G.P. Copeland, and W.G. Wilson. DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–323, 1995.

[35] H. Lu, B.-C. Ooi, and K.-L. Tan, editors. *Query Processing in Parallel Relational Database Systems*, chapter Parallel Query Otimization. IEEE Computer Society Press, 1994.

[36] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.

[37] E. Ding, L.A. Diminio, G. Gopal, and T.K. Rengarajan. Parallel processing capabilities of Sybase Adaptive Server Enterprise 11.5. *Data Engineering Bulletin*, 20(2):35–43, 1997.

[38] Wei Hong. Exploiting Inter-Operation Parallelism in XPRS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 19–28, USA, June 1992.

[39] R.S.G. Lanzelotte, P. Valduriez, M. Zaït, and M. Ziane. Invited project review: Industrial-strength parallel query optimization: issues and lessons. *Information Systems*, 19(4):311–330, 1994.

[40] G. Myerson and A.J. van der Poorten. Some problems concerning recurrence sequences. *The American Mathematical Monthly*, 102(8):698–705, October 1995.

[41] A. Pellenkoft, C.A. Galindo-Legaria, and M.L. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proceedings of the International Conference on Very Large Databases*, pages 306–315, Athens, Greece, September 1997.

[42] L. Shapiro and D. Maier. Pruning in the Columbia query optimizer. Information about the Columbia Query Optimization Project at:http://www.cs.pdx.edu/.

[43] Pedro Celis. The query optimizer in Tandem's Serverware SQL Product. In *Proceedings of the International Conference on Very Large Databases*, page 512, Bombay, India, September 1996.

[44] W. Scheufele and G. Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *International Conference on Extending Database Technology*, pages 201–215, Valencia, Spain, March 1998. Springer Verlag, LNCS 1377.

[45] Florian Waas. Handling non-deterministic data-availability in parallel query execution. In *Proceedings of the International Workshop on Database and Expert Systems Applications*, pages 61–65, Florence, Italy, September 1999. IEEE CS Press.