ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

ModParOpt: a Modular Query Optimizer for Multi-Query Parallel Databases

Lionel Brunie and Harald Kosch

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76227-2

ModParOpt: a Modular Query Optimizer for Multi-query Parallel Databases

Lionel Brunie
LIP-lab,
ENS Lyon
Lyon, France

Harald Kosch
Dep. of Information Technology,
University Klagenfurt
Klagenfurt, Austria

Abstract

This paper presents MODPAROPT, MODular PARallel Query OPTimizer) a parallel optimizer for complex relational queries in a multi-query environment, which meets perfectly the requirements of modern database applications (e.g. decision support and data mining). The optimizers architecture was developed in a very strict modular way being therefore highly extensible. It integrates an intelligent resource allocation module coupled with a randomized search module in order to seek for the best parallelization strategy when resource availability is heterogeneous and probably limited. Experiments performed on a 100 relation database with 432 randomly chosen queries show the effectiveness of MODPAROPT.

Keywords: Parallel databases, decision support systems, parallel query optimization

1 Introduction

Modern database applications, such as data mining and decision support on a data warehouse pose several new challenges to parallel relational query optimization and processing [1]. The complexity of queries against the data base size rises significantly compared to traditional systems (i.e. typically the number of joins and the relation size are the dominant factor in such complex queries, which can grow up for reasonable applications to 30 joins for relation size of some Gigabytes [2]). Furthermore, most of the data warehouse are built for multiple occupancy. Thus, the parallel DBMS must provide concurrent processing of several queries (further referred as a *multi-query* mode). However, the number of concurrent queries is lower than in OLTP processing, but even with few users on the system the management of resource contentions becomes a crisis problem. Finally, the queries are read-only, thus no locking mechanism must be involved.

In front of this problems, the fast development of high performance parallel machines provided with efficient multi-tasking opens the possibility to exploit massive parallelism for each complex query execution and higher throughput of concurrent complex query execution [3]. Optimizing complex queries for a parallel multi-query environment introduces three main problems. At first, the *inhomogeneous resource availability* must be incorporated in the optimization process. At second, *resource contentions* must be managed in order to maintain an acceptable response time. Finally, *additional costs* due to *sharing the resources* between concurrent operations of one query and between queries must be taken into account.

Most of the related work assume a single-query optimization and are hardly applicable to the multi-context. In this context, we use a three-step approach to parallel join-query optimization. Firstly, the *query execution frame* is determined, i.e. the resources attributed to the complex query, depending on the actual and predicated query processing workload. Then, a *scheduling* and *parallelization technique* is developed which explicitly attributes time-shared resources, such as the CPU and the disk bandwidth and non-preemptable resources, such as the memory. Furthermore, the *resource contentions* are managed, by *serializing* parallel join operations. Finally, the proposed mechanism is completed by a third step, proposed in our related work [4], which provided a run-time *control mechanism* in order to *reoptimize the query* in case of drastically changing availability of the resources.

Figure 1: The modules of MODPAROPT

Next section 2 analysis related works. Section 3 gives an overview of the software architecture of MODPAROPT. Section 4 describes the transformation manager (transformation rules, interface between the resource allocation and the utilized randomized search strategies). Section 5 presents the resource allocation module. Section 6 describes the match optimizer, in charge of determining the best implementation strategies. Section 7 reports selected results of our experiments, showing the pertinence of the methodology. Finally, section 8 proposes a conclusion and points out future works.

2 Related work

A parallel join-query optimizer has first to determine the queries join ordering [1]. This implies the fixing of the *degree of inter-operation parallelism*, i.e. the number of joins to be executed concurrently. The number of possible orderings faces a combinatorial explosion, i.e. up-to 17643225600 for 10 joins and 4299578163927654889881600000 ($4.3 \cdot 10^{27}$) for 20 joins [5]. Furthermore for each participating join the optimizer must compute the degree of *intra-operator parallelism*, i.e. the number of processors to execute one join operation, the *join implementation method* e.g. a hash-join or a sort-merge technique and if *pipeline parallelism* should be enabled between two successive join operations. Finally the *resource allocation*, i.e. processor and memory allocations must be done.

The traditional technique of dynamic programming [6], utilized in sequential query optimization, performs almost an exhaustive search over the search space. Even, if special implementation techniques are used to reject obvious costly plans, this technique becomes soon intractable in the parallel context, as pointed out by several authors [7, 8]. Therefore some works propose heuristic based optimization methods [9, 10]. However, those heuristics rely on the cost function that is not always satisfied. For instance, most techniques incrementally construct an optimal ordering by making an optimal decision at each step. Possible redistribution extra-costs due to data not located to the same processors or advantages of sorting in the remaining of the optimization can not be predicated. Thus optimality may be lost. Randomized search strategies offer a way around those

problems; they perform local transformations on already complete execution, until a low-cost scenario can be found. The use of randomized search strategies in parallel query optimization was first examined by Lanzelotte et al. [8] based on experiences in sequential optimization [11]. The presented framework only considered intra-operator parallelism. Spilipoulou et al. [12] then studied the effectiveness of those strategies for inter-operation parallelism. However, this approach suffers from the fact that it does not consider resource limitations and thus generates parallelism without load balance. Therefore only a subset of effective inter-operation parallelism is considered.

During query execution multiple resources must be time-shared, e.g. the CPU and disk bandwidth, others must be partitioned e.g. the memory, in order to enable concurrent running operations on one processor. Previous works [13, 14, 15] summarize the work done on timed shared resources in a weighted sum (typically called 'work'). However, these approaches render it impossible to take into account the multi-dimensionality of the resources. Furthermore, the memory consumption is seldom encountered [16, 13].

In our framework, MODPAROPT is introduced to deal with all kinds of parallelism in a multi-query environment. Furthermore it intelligently choose the appropriate degree of parallelism due to the resource availability and possible resource contentions. The proposed execution scenario differs so less from real processing situations and renders the optimization process more effective. Although the cost models and the resource modules are more complex, experimental results in section 8 will show that the optimization time remains very acceptable.

3 Overview of MODPAROPT

MODPAROPT is built up in the strictest possible modular way (see fig. 1). Search strategy, resource allocation, cost model and the choice of the best implementation strategy are put into different modules. Each module is optimized independently. Extensions and modifications are easily achievable.

Three static modules are distinguished, the *Catalog Manager* contains the static informations of the relations (e.g. the tuple size, relation cardinality), the *Distribution Manager* describes the way the relations are distributed over the disks. Finally the *Cost Manager* includes the cost functions for the optimization process. We implemented a first-last page cost model, including communication costs. Latency delays of pipeline parallelism and overlapping of communications and computations can be easily expressed with this formalism. Furthermore, special functions were added in order to calculate the CPU and disk access costs in dependency of the resource utilization rate. Those functions penalize cost overheads due to over-utilization of time-shared resources.

For the dynamic modules, the *Transformation Manager* implements the search strategy, i.e. it chooses the transformation to be applied to the current scenario and thus the degree of inter-operation parallelism. It communicates with the *Resource Allocation* module, in order to determine the optimal intra-operation parallelism and the processors allocation. Based on the result of the latter, the *Transformation Manager* computes the new scenario costs and decides if the transformation can be accepted. The *Resource Allocation* module depends on the *Match Optimizer*. For each join operation the *Match Optimizer* fixes the best implementation method. A supplement module, the *Initial plan generator* produces a randomized initial scenario for the optimization process. To finish, the *Temporary Manager* includes information about the intermediate relations and its partitioning.

4 Transformation manager

Classically, for join processing [17], six transformations rules are distinguished to exploit the optimizers search space (bushy space):

1. *Method choice*-transformation: $R_1 \bowtie_{method1} R_2 \rightarrow R_2 \bowtie_{method2} R_1$
2. *Swap*-transformation: $R_1 \bowtie R_2 \rightarrow R_2 \bowtie R_1$
3. *LeftJoin exchange*-transformation: $(R_1 \bowtie R_2) \bowtie R_3 \rightarrow (R_1 \bowtie R_3) \bowtie R_2$

Figure 2: Left and right parent-child relationship.

Figure 3: Different subtrees may provide the join attribute of J_2 .

4. *RightJoin exchange*-transformation: $R_1 \bowtie (R_2 \bowtie R_3) \rightarrow R_2 \bowtie (R_1 \bowtie R_3)$

5. *Join associativity*-transformation: $(R_1 \bowtie R_2) \bowtie R_3 \rightarrow R_1 \bowtie (R_2 \bowtie R_3)$

In our objective to implement a fast parallel query optimizer we optimized the number of applied transformations. At first the transformation rules applied to sequences of three joins (3. 4. and the 5. rule) are regrouped to an unique rule.

The common hypothesis is joined to exclude cartesian products from the parallel search space [17, 18, 12]. Indeed the parallel bushy search space is complex enough. Furthermore our work does not intend to study if the processing strategy including cartesian products will bring much improvement to the problem of finding optimal execution scenarios¹. They are likely to have lesser impact to our problem of optimizing complex queries on parallel machines disposing multi-tasking capacities. The reason for this is that the size of intermediate results in cartesian products are in general much higher than for joins operations, augmenting the number of resource contention conflicts.

The regroup of the 3., 4. and 5. transformation rule to a single transformation **JoinExchange()** has been motivated by the observation, that not all applicable transformation rules lead to a valid processing tree (in a processing tree, the nodes represent the relational operators and the arcs represent the data flow) since they may create cartesian products. For instance let us consider fig. 2, left scheme, where the join J_1 is to be exchanged with the join J_2 . In the given left parent-child relationship between J_1 and J_2 , the 4. and 5. rule can be applied. Let us furthermore assume that the joins are exercised on different join attributes, thus only one of the two subtrees A or B, let us say B, can contain the join attribute of J_2 . Therefore only the application of the 4. rule generates no cartesian products.

Given an arbitrary query processing tree structure q two basic starting points for the *JoinExchange()* can be distinguished (see fig. 2). A, B and C are three subtrees of q while J_1 and J_2 are the two joins that are subject to the *JoinExchange()*. As these two configurations differ only in the parent - child relationship between J_1 and J_2 it is sufficient to regard the different cases that may evolve when applying a *JoinExchange()* to the case of a left parent-child relationship as shown in fig. 2, left scheme.

At first, let us assume that only one of the two subtrees of A and B provides relations containing the join attribute of join J_2 . Then two cases, as shown in fig. 3 can be distinguished. The subtrees providing the join attribute of join J_2 are marked.

Taking a closer look at the case where the resulting relations of A and C carry the join attribute. It becomes clear that J_2 must take A and C as its parent tree. Four different implementations are then possible : either A

¹ Some authors (e.g. [7]) have proposed recently search algorithm to investigate bushy tree space for sequential query optimization. Even if supplement search effort pays off in finding better sequential plans, the authors state that the cost model were too simple to allow general conclusions.

Figure 4: Subtree A is providing the join attribute.

Figure 5: Subtree B is providing the join attribute.

or C is the outer relation of J_2 and for each of these possibilities, J_2 can be the inner or the outer relation of J_1 . Our *JoinExchange()* implements the case where the result of J_2 is the inner relation of the result of J_1 and C is the outer relation of J_2 (see fig. 4). The three other alternatives can be attained by the chosen *JoinExchange()* implementation by applying at most two supplement *Swap()* transformations.

Fig. 5 illustrated the transformation management for the case when the join attribute being provided by subtrees B and C. Here, the existing left parent-child relationship between J_1 and J_2 is changed into a right parent-child relationship. Once again, all other alternatives can be attained by supplement *Swap()* transformations.

At second, considering the case when both subtrees of A and B provides the join attribute of J_2 (this case is in reality rare). An heuristic approach is chosen to handle the larger set of successor processing trees. First the transformation is performed as if only the subtree A provides the join attribute. the costs of the resulting processing tree is compared with the cost of the processing tree obtained by a transformation, which considered that only the subtree B provides the join attribute. The lower cost processing tree is retained.

Figure 6: The transformation application has side effects on the query remainder.

4.1 The resource allocation and the application of transformations rules

The application of the transformation rule to a join (or two joins) implies a reallocation of the resources, possible serialization and a new choice of the implementation strategies for those joins, i.e. the transformation of the *processing tree* to a *scenario* (we refer to scenario for a physical representation of the query).

For example, consider the example query tree of fig. 6. Performing a *JoinExchange*(J_3, J_4), changes the producer of the outer relation for J_5 from J_3 to J_4 . Consequently, the arriving time of the data pages of J_4 for pipeline consummation changes, thus the whole memory availability changes. Therefore, resource allocation for join J_5 has to be reconsidered.

Obviously, the resource allocation must be done for all joins presented in the processing trees when applying a local transformation². This allocation is achieved by ordering the joins according to their position in the processing tree. The furthest left join receives the lowest position 0. Walking to the right until the end of one level or switching to the next level increases the position by one. For example, in fig. 6, the join J_5 has the position 4, as the join J_2 holds the position 1.

After the ordering of the joins is complete, we are now left with the task of detecting which of the joins are parallel. However this can only be done for the joins having a smaller position (only for these joins the new resource allocation has already been computed). Therefore we take the risk of not detecting some of the parallel joins. To detect all the parallel joins would require us to go over the processing tree at least once again. This technique is not cost effective.

Our chosen strategy can also be imagined in terms of priority. The joins with the smaller positions have the higher priority to use the resources as memory, CPU and disk. The join operations with the higher positions must use the resources which are not yet exploited. That does not mean that they have in general smaller resources at their disposition as the joins with a smaller position. A special factor (described in section 5.3) regulates the memory attributions. It is a pre-defined function of the user requirements as the system specifications and guarantee the fairness of the resource allocation.

4.2 Randomized search strategies

For managing inter-operation parallelism in complex queries, randomized search strategies have been suggested as one alternative to restrict the search space. Randomized algorithms are well known combinatorial optimization [19], and two basic variants of these algorithms are implemented in the *transformation manager* : the Simulated Annealing (SA) and the Iterative Improvement (II) [8]. Furthermore a extension to the two basic methods is utilized : the Two-Phase Optimization (TPO).

The general idea is as follows. The randomized algorithms start from several random states³ in the search space. For each random initial state, the algorithms walk through the search space, evaluating the cost of each state and stopping either when they estimate that they have found the optimum state or when a predefined optimization time expires. The walking between the states is controlled by transformation rules and a global strategy. The II technique accepts a move from one step to another only if the cost of the destination state is lower than the cost of the source state. The SA technique, on the other hand, allows a move to a higher-cost state with a certain probability that diminishes as optimization time moves along. The TPO technique is a combination algorithms for the II and SA. In the first phase an II is run for a few local optimizations. The output of this phase, which is the best local minimum found, is the initial state to the next phase, where an SA is run. This algorithm tries to limit the number of initial random states to II, by applying the SA technique to come up with too high cost local minimum.

Taking a very close look at the proposed search algorithms, one notices that the runs, i.e. the local optimizations for each initial state, in an II and TPO are independent from each others and could so be executed in parallel. So why not utilize the capacities of the parallel machine for optimizing the queries. Thus, the original *parallized II* and *parallized TPO* are proposed. In the *parallized TPO* as many SA runs are made as there have been II runs within the first phase. Experiments (see section 8) will demonstrate that this parallized TPO is

²which is not the case for sequential optimization [11].

³The generated solution *states*, according to the usual convention in randomized optimization algorithms [19].

the most cost-effective technique. In order to generate with 95% a good execution scenario⁴, until a 25-way joins, only 6 processors are required to independently run one II+SA per processor.

5 Resource allocation module

In this section, at first the resource usage model, parallel cost model, the memory management are presented. Then the novel four level resource allocation heuristics is introduced, which computes the degree of intra-operator parallelism and the processors allocation, with regard to the available main memory size, CPU power and disk bandwidth of the processors. The hierarchy of the heuristics is based on previous experiences in parallel databases and on an original serialization technique.

5.1 Resource usage model

The work done on the resources in a multi-query parallel database is measured by most related works (see section 2) as the sum of the effective time where an operator uses the resources. Such an approach makes it difficult to compute the additional cost of time-sharing preemptable resources like the CPU and the disk bandwidth. Rather than computing the sum of the elapsed time spent on the resources we are more interested in the question whether or not a resource is over-used at any moment of the query execution.

For instance, for the preemptive resources, the CPU and the disk bandwidth, we introduce the term : *number of concurrent users* in order to measure the work done at any moment on those resources. A maximal number had to be fixed a priori by a system administrator, depending on the standard query complexity and the resource availability. Time sharing of the CPU and the disk bandwidth causes additional costs of context switch. Even if a multi-threaded machine now provides faster switching than a multi-process machine, such costs are not negligible. We compute these costs according to the works of Mehta et al. [20] made in the context of parallel single-join queries in multi-query databases. Once an operator is scheduled on a processor already occupied by some other operators, the operator execution time is increased by 10%.

Moreover, with a certain number of concurrent processes, the system performance deteriorates. Hence, the maximal number of concurrent users must be chosen in a way that on each processor, no deterioration can occur for normal workload estimations. For computing the work done on non-preemptable resources, like the memory, and for determining their allocation, special features must be introduced. A regulation factor is presented, which guarantees a fair utilization of the memory. The higher the value of this factor is chosen, the more memory is granted to the first join, which consequently will leave less memory to the other joins running in parallel to it. This factor is also set by the database administrator in dependency of the initial memory availability.

5.2 The parallel cost model

The cost of a query executed on a parallel machine must contain two features, the response time of the query and its resource consumption [13, 16]. Computing the query response time means making an estimation of the execution time, as the query resource consumption must detail the work done on individual resources.

In this context, we present the cost model the response time is composed of a **local** and a **global time**. The *local time* represents the elapsed time to execute one operator of the execution scenario in parallel. It does not include costs issuing from the dependency between operators e.g. latency delays in pipeline parallelism. The *global time* contains the local times due to the kind of data dependency (pipelined or sequential). The resource consumption will be properly addressed in the next subsection.

In order to account for pipeline parallelism, we have chosen to let the global and local response time have two parts (the first and last page time). The **first page time** is the time when the first page is output by the operator (for the local time) or an execution scenario (for the global time). The **last page time** is the time, when the operator terminates its processing or the execution of the scenario has been completed. This kind of

⁴Good means in the experiments, not more than 10% over the lowest local minimum cost found in the experiments

time model will be furthermore called the **first and last page response time model**. This model has first been introduced by Ganguly et al. [13]. The original definition does not yet introduce all dependency combinations, e.g. pipelined or sequential consumption of the input relations. In the following we introduce briefly the complete computation of local and global response time.

The *local response time* is computed using architectural parameters and database statics by querying the *catalog manager*, the *distribution manager* and the *temporary Manager*. It is based on typical formulae used by the gamma shared-nothing systems [21], which have been experimentally validated. The *local response time* is computed by the *resource allocation* module and the *match optimizer*.

The *global response time* represents the elapsed time to execute a scenario and is computed recursively over its structure. Several computation cases must be considered. At first the scenario is composed of an input scenario and one operator. Then, two scenarios input to one operator. For each of the two cases it must be distinguished if the data dependency of the input scenario to the operator are pipelined or sequential. The *global response time* is computed by the *transformation manager*, relying on the local response time of each operator.

In detail, let us first assume that only one scenario inputs to an operator. Then, let (t_f, t_l) be the local response time of the operator, where t_f notes the response time when the first page is output and t_l , the response time when the last page is output. Furthermore the input scenario has the global response time (i_f, i_l) .

The global time of the resulting scenario (o_f, o_l) computes in the case of sequential dependency as $(i_l + t_f, i_l + t_l)$ and in the case of pipelined dependency as $(i_f + t_f, i_f + t_f + \max(i_l - i_f, t_l - t_f))$. The latter formula reflects the fact that the operator can start as soon as the scenario has output the first page. The remainder of the processing the input scenario then runs in parallel with the remainder of the operator processing. Note, that pipeline delay is correctly modeled, as follows. Delay caused by communication of pages is encapsulated in a communication operator which must be placed at the end of the input scenario. Delay issuing from slow producer and fast consumer (or visa versa) is hidden in the $\max(., .)$ expression of the latter formula.

Secondly, let us suppose that two scenarios input to an operator. Let $(i1_f, i1_l)$ be the local response time of the first input scenario and $(i2_f, i2_l)$ those of the second input scenario. The (t_f, t_l) notes again the local response time of the operator.

Here three cases of data dependency must be considered. Firstly, the input scenarios output the pages in sequential dependency to the operator. The global time of the resulting scenario (o_f, o_l) expresses then as :

$$(\max(i1_l, i2_l) + t_f, \max(i1_l, i2_l) + t_l)$$

Secondly, for the case that the two dependencies from the input scenarios are pipelined, we obtain the following expression for (o_f, o_l) :

$$(\max(i1_f, i2_f) + t_f, \max(\max(i1_f, i2_f) + t_l, i1_l + t_f, i2_l + t_f))$$

For this case, if we assume to have no buffers available, the remainder of the operator runs in parallel with the remainder of the first and second input scenario.

Finally, when only one data dependency is pipelined, assuming without loss of generality that the first input is pipelined, we can rely on the latter formulae and obtain for (o_f, o_l) :

$$(\max(i1_f, i2_l) + t_f, \max(i1_l + t_f, t_l + \max(i1_f, i2_l)))$$

5.3 Memory management

The initial free memory On a multi-tasked machine several processes must share the processors main memory. The commonly used technique to manage the memory attribution is the introduction of priorities. In the context of a multi-query database, the different processes can implement parts of simple or complex queries. For simple queries working directly on base relations, the highest priority must be assigned. For more complex queries the priority attribution depends on the characteristics of the query and the user requirements. For instance, queries from real-time application have to be executed with a higher priority. Other queries, with no special user requirement and real-time constraints can be run in batch (i.e. with a very small priority).

Attributing such priorities is a complex problem. We do not address this problem, as it does not affect our resource allocation. Therefore it is assumed that a database instance has attributed a priority to the submitted

query. Furthermore, this instance computes the processors, their memory size and number of available CPU processes which *could* be used by the submitted query, depending on the attributed priority, the number of operations in the query and the sum of the size of the participating relations. For the memory, reserved for the query execution on each processor we will furthermore refer to as the initial free memory : $mem_{initial}(P_i)$ on processor P_i

Finally, it must be remarked, that the low priority of complex queries render their processing performance very sensitive to changing workloads on the machine. For example, supposing a suddenly increasing workload. The resources must then be taken away from the lower priority complex query to support a reasonable throughput of all running queries. In such a case a reoptimization of the complex query must be made immediately, in order to guarantee still good performance. This point will be properly addressed in our related work [4]

Requested memory The memory capacity for hash-join processing is supposed to be equal to the size $\|R\|$ of the inner relation R for the join $J = R \bowtie S$, as the memory resident hash table is built on it.

Furthermore, it is assumed that $\|R\|$ is being equally distributed over the np join processors. This determines the amount of requested memory $mem_{req}(P_i)$ per processor P_i . The mem_{req} describes then as : $mem_{req}(P_i) = \frac{\|R\|}{np}$. Note, that this hypothesis is common to all compile-time query optimization approaches [17, 16]. Redistribution skew effects are run-time dependent and can not exactly be determined, even if they occur at all.

Memory allocation In order to allocate the memory, for each P_i of the system, the value of $mem_{req}(P_i)$ must be compared to the amount of available memory $mem_{avail}(P_i)$ on this processor P_i . $mem_{avail}(P_i)$, is taken from the share of free memory, on P_i ($mem_{free}(P_i)$).

The latter free memory size $mem_{free}(P_i)$ depends on the number of processes c , involved in the processing of J and accessing the shared memory on the processor P_i . If no process is *already* working on P_i ($c = 0$), the $mem_{free}(P_i)$ is the initial memory ($mem_{initial}(P_i)$) dedicated to the query. Its size is calculated, when submitting the query (see the previous paragraph). On the contrary, $mem_{free}(P_i)$ computes as the difference of the initial memory minus the total used memory shape of the c processes. The available memory is computed as follows :

$$mem_{avail}(P_i) = mem_{free}(P_i) * k(c)$$

where c is the number of concurrent processes on the processor P_i and the factor $k(c)$ ($0 \leq k \leq 1$) is the so called *regulation factor*.

It is obvious that $k(c)$ is a powerful means to *regulate* the joins memory distribution. The higher the value of $k(c)$ chosen, the more memory is granted to a join, which consequently will leave less memory to the other joins running in parallel to it (i.e we favorite a *bushy processing strategy*). On the contrary a lower value of $k(c)$, is more likely to generate a *linear processing strategy*.

In the implementation of MODPAROPT, we simply $k(c)$ chosen as a constant C for $C > \frac{1}{2}$ and for $C < \frac{1}{2}$ as :

$$k(c) = \begin{cases} C & : c = 1 \\ 1 - C & : c > 1 \end{cases}$$

5.4 Hierarchical allocation heuristic

In the first level, it is examined if the data locality of the relations can be retained. In such a case, the join is performed on the processors/disks where the relations are stored and therefore avoiding supplement communication and memory utilization. This first allocation strategy appeared to us the most important, as all shared resources are used less. The same significance was attributed by many other authors, for instance see the algorithm developed by Hasan et al. [22] or Hameurlain et al. [23] for optimizing the communication costs. Even if communication network speed has increased dramatically over the last years (e.g. faster networks as the optical one come up quickly), additional costs on the sender and receiver machine could not be neglected. For instance, on both sides communication buffer must be provided in memory to store the data to be sent and received, causing memory space problems and computing additional costs in the context of huge relation

sizes [12]. Of course, data locality can not be conserved when the concerned processors do not hold enough resources to process the join, or when the relations are not distributed on the join attribute, which is in general the case for intermediate results.

So the second level heuristic tries to distribute the relations in such a way that they fit in the processors main memory. This avoids cost intensive temporary I/O access. In detail, if a processors set exists which puts all inner relation partitions in the processors memory, the minimum and maximum degree of intra-operator parallelism ($diop_{min}$ and $diop_{max}$) related with the smallest and largest possible set of processors allocation ($allocation_{min}$ and $allocation_{max}$) is determined. If at least one degree of intra-operator parallelism is found, the *load balancer procedure* is called, which chooses the final degree $diop$ and the final processors allocation $allocation$. On the contrary, the third level must be addressed, which includes bucket join processing. We put this strategy at the second level, motivated by the works done on parallel in-memory databases showing excellent performances e.g. [24].

The third level heuristic splits the relation into partitions to be hold in the main memory. As the second level heuristic, it searches the minimum and maximum degree of intra-operator parallelism ($diop_{min}$ and $diop_{max}$) and allocation ($allocation_{min}$ and $allocation_{max}$), for which a reasonable bucket processing is possible, is returned and the *load balancer procedure* is applied. Otherwise, if the processor allocation fails once again, the fourth level heuristic is applied.

In this fourth level, the join operations running in parallel, are serialized. Here, the resource contentions affect directly the computation of the degree of inter-operator parallelism, i.e. the degree must decrease to enable a correct processor allocation by serializing parallel joins. This maintains the original join ordering while adapting more precisely to the machines workload as do earlier approaches. After the serialization, the resource allocation heuristics are reapplied (starting from level 1). If, even with this new constraint, the allocation is not possible, the transformation must be rejected.

If the four levels find a possible processor allocation, it is adjusted in order to optimize the stream of pipeline parallelism; the latency of a pipeline chain should be the least possible (*load balancer procedure*). That means the appropriate $diop$ is chosen out of ($diop_{min}$ and $diop_{max}$) which optimizes this stream of pipeline. If no pipeline parallelism is utilized, the degree of intra-operator parallelism can be chosen freely into the interval $[diop_{min}, diop_{max}]$. Again an heuristic approach must be chosen. We began by examining if the inner relation is small enough to fit in the memory of the processor disposing the highest available memory. Allocating the join to only one processor eliminates the problem of data skew. Then, we must ensure that the degree of intra-operator parallelism remains under a certain threshold degree (above this threshold, intra-operator load balance becomes critical). The existence of such threshold has been demonstrated by Hua et al. in [25]. This threshold must be set by the database administration, based on statistics on previous query executions. For all other cases, a tradeoff factor, set by the system, is introduced. It decides whether higher *intra-operator parallelism* should be applied or not. At last, if the work of two parallel join operations differs significantly no adjustment would be found to equally distribute the work. Therefore the joins will be serialized and the resource allocation heuristics is reapplied (starting from level 1).

6 Match optimizer

The *match optimizer* of MODPAROPT is in charge of evaluating the best implementation strategy for each join and its related costs, based on the processors allocation for the joins, provided by the *resource allocation* module. In comparison to previous works, see the overview in [26], the best strategy is quickly determined by searching in a pre-defined table (referred as a *decision table*) of typical execution cases.

The construction of such a *decision table* is possible, as the join implementation strategies have been largely investigated in literature. In our related work [27] an comparative study of the three basic classes of join algorithms (nested loop, sort-merge and hash-join) in a shared-nothing environment based on a literature study and an analytical model was introduced. The result of this comparative study was put in the decision table available for the *match optimizer*.

Figure 7: Average hit-rate of the II and TPO.

7 Experimental results

In this section, an overview of the experiments done with MODPAROPT on a decision support database schema with 100 relations and 432 queries is given. After tuning the parameter of the randomized search algorithms (the reader is referred to [28] for the tuning), the quality of the execution scenario found by the different search algorithms were studied. Furthermore, the optimization effort compared to the estimated processing time was examined.

7.1 Experimental testbed

An relation scheme of 100 relations was designed. The corresponding join graph has been designed in a way that adapts it the well for decision support (DSS) and data mining queries. Firstly, three stars (subgraphs with one special node which is connected by only one edge to several others) were introduced in the scheme graph. This corresponds typically to commercial databases where a lot of small relations are connected to one large one. Queries for such connectivity are called *star queries*. These star queries have traditionally been regarded as one of the representative DSS query types [7, 17]. Secondly, several long strings are presented in the query (a string is a subgraph where the nodes only have one incoming and one outgoing edge) in order to implement the so-called *string queries*. These string queries, along with the star queries, are considered very likely to appear in DSS [7, 17]. For instance, the 9 relations in the TPC-D decision support benchmark [29] are arranged along a string of 6 relations. Thirdly, subgraphs in which the relations are highly inter-connected were realized. Queries where at least 50 % of the participating relations belong to such subgraphs will hereafter called *high connectivity queries*. Although *star* and *string* queries are considered DSS representatives, the *high connectivity queries* are not unlikely to occur.

Figure 8: Percent of optimization time versus processing time of the TPO technique.

Different types of relation cardinality, tuple size, number of attributes and the number of unique values in the join attribute (controls the join selectivity) were chosen for the 100 relations. The relation cardinality is chosen between 10 and 50,000,000 tuples, the tuple size ranging from 1 to 1,000 bytes, the number of attributes from 2 to 26 and the ratio of the $(\text{number of unique values}) / (\text{relation cardinality})$ from 0 to 1. The latter ratio determines the join selectivity.

The proposed characteristics are quite close to the optimizer benchmark proposed by Ioannidis and later Spilipoulou [11, 12]. In addition to those works, the relation cardinality range here more widely, in order to adapt to the characteristics of the decision support benchmark TPC-D [29].

The considered system architecture is *shared-nothing* for 8, 32 or 64 processors. The experiments for different resource availabilities cases were run : the *full resource availability* case, where each processor disposed of 25 MBytes dedicated to the query and the maximal number of concurrent join processes on each processor is assumed to be 3 (a multi-query environment is supposed, thus only 25 MBytes on each processor dedicated to a query is chosen) ; the *memory contention* and *full CPU power* case, where each processor disposed only of 5 MBytes dedicated to the query ; the *full memory availability* and *CPU contention* case, where only one join process is allowed on each processor and the *memory and CPU contention* case, with 5 MByte processors memory and one join process on the processors.

Randomly designed were 18 query-types (size ranging from 6 to 25 joins) to be optimized under 24 criteria; i.e. varying number of available processors (3 cases), different resource availability (4 cases) and the two regulation factors (2 cases), which presented 432 individual optimizations. The 18 different query-types were repartitioned for their complexity : 5 queries for 6, 11 and 15 number of joins and 3 queries for 25 number of joins. For the each query complexity one star, one high connectivity, one string and two random queries (not 25 joins) were implemented.

7.2 Analyze of the results

To get an idea for the characteristics of the local minima and its cost distribution, an *Iterative Improvement (II)* was run 200 times for 6, 11 and 15 joins and 50 times for 25 joins. Furthermore, the search space was randomly sampled. First of all the average degree of inter-operator parallelism of the lowest minima costs was examined.

In the presence of CPU contentions and for star-queries, the optimizer chosen clearly lesser inter-operation parallelism than for all the other cases. But, even in this case the average degree of inter-operation parallelism was high in comparison to the query complexity (not under 1/5-th of the number of joins). The reason is that in such a case inter-operation parallelism can be efficiently exploited by allocating the joins to disjointed subsets of processors. Note that these results are attributed to the experiences of the IBM DB2 Parallel Edition optimizer [3]. Furthermore, the more complex the query is, the more inter-operation parallelism should be exploited. When concerning the most appropriate degree of inter-operator parallelism for a query, the whole available degree should not be chosen, because it generates too much cost-inefficient resource contentions, which are not serializable.

The average local minima cost was more than 1000 times lower than the highest randomly sampled cost. Moreover the average local minima cost was more than 30% above the lowest one, stating that it is hard to generate a good scenario. This observation was deepened by examining the percentage of local minima whose costs are inferior to the optimal costs found plus 10% was examined (hereafter called *simple hit-rate*). Based on that *simple hit-rate* an average number of runs is calculated for which the parallelized II finds a local minimum cost below the optimal costs found plus 10% with a probability more than 95% (hereafter called *II hit-rate*).

Figure 7 shows the average II hit-rate for every query type as a solid line (curve marked II). Just as for the 6-way joins, the average II hit-rate is 33%; for the 11-way joins, it decreases significantly to 8% and for the 15-way joins to 6%. Astonishing, however is, that for the 25-way join, the average II hit-rate increases slightly to 12%. This could be explained by the smaller sample size of the experiments compared to the lower complex queries.

The number of runs required to achieve a II hit-rate of 95% is relatively high, i.e. 36 for the 11-way joins and 48 for the 15-way joins. This high number render the II technique costly.

Then, we examined the hit-rate for the parallelized Two Phase Optimization (TPO) technique, i.e. for every local minima generated by the II, an Simulated Annealing is applied whose optimization effort is limited to those of the II.

Figure 7 shows the average TPO hit-rate for every query type as a dotted line (curve marked TPO). The hit-rate for the TPO technique increases compared to the II hit-rate on average by a factor of 3.8. The number of runs required to achieve a TPO hit-rate of 95% is not higher than 6. Compared to the II technique, the TPO requires double the number of applied transformations, but still reduces the number of runs by significantly more than the half. Thus, the TPO technique turns out to be the most effective search algorithms.

In a new series of experiments the optimization effort for the TPO technique was studied. The time to process an optimization for query to be executed on 32 processors varies in average from 2.5 seconds (6 joins) on four SUN Sparc 20 workstations, to 6.3 seconds (11 joins), to 21.7 seconds (15 joins) and to 67.7 seconds (25 joins) on six SUN Sparc 20 workstations.

Indeed, measuring only optimization times disrupts correct complexity analysis, as long-running queries can effort more time-intensive optimization. Thus for each query complexity, the average optimization time running the TPO was compared to the average estimated query processing time for the example of a 32 processors system which was calculated by the optimizer. Figure 8 illustrates that very acceptable optimization times were achieved, even for 25 joins. However, the results for the 25 joins shows the tendency that proposed optimization techniques will become intractable for super complex queries. The use of such queries is very limited and they are unlikely to become more popular in the future.

8 Conclusion and future work

This paper introduces ModParOpt, a parallel query optimizer especially adapted for optimizing complex and data-intensive queries (typical for DSS and data mining applications) for a multi-query parallel database. Our

optimizer manages heterogeneous resource availabilities and resource contentions, and handles additional cost of time-shared resources.

Experiments performed on a 100 relation database with 324 randomly chosen queries show the effectiveness of the proposed search strategies and the allocation algorithms.

Object-oriented databases attract more and more attention, when complex data-modeling is required. Future works will concern the integration of the proposed methodology in the object-oriented query optimization.

References

- [1] W. Hasan D. Florescu and P. Valduriez. Open issues in parallel query optimization. *ACM Sigmod Records*, 25(3), September 1996.
- [2] Jim Gray. Parallel Database Systems Survey. In *Tutorial Handouts of the 21st International VLDB Conf.*, Zurich, Switzerland, September 1995.
- [3] 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), 1995.
- [4] L. Brunie and H. Kosch. Control strategies for complex relational query processing in shared nothing systems. *ACM Sigmod Records*, 25(3), September 1996.
- [5] K.-L. Tan and H. Lu. A note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems. *ACM Sigmod Records*, 20(4), December 1991.
- [6] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2), June 1984.
- [7] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. In *ACM SIGMOD Conf.*, Montreal, Canada, June 1996.
- [8] R.S.G. Lanzelotte P. Valduriez and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *International VLDB Conf.*, Dublin, Ireland, August 1993.
- [9] E.J. Shekita K.L. Tan K.-L. and H.C. Young. Multi-Join Optimization for Symmetric Multiprocessors. In *International VLDB Conf.*, Dublin, Ireland, August 1993.
- [10] J. Srivastava and G. Elssesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *International PDIS Conf.*, Los Alamitos, California, USA, December 1993. IEEE Computer Society Press.
- [11] Y.E. Ioannidis and Y. Cha Kang. Randomized Algorithms for Optimizing large join queries. In *ACM SIGMOD Conf.*, Atlantic City, New York, USA, 1990.
- [12] 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 Trans. on Know. and Data Eng.*, 8(3), June 1996.
- [13] S. Ganguly W. Hasan and R. Krishnamurthy. Query Optimization for Parallel Execution. In *ACM SIGMOD Conf.*, San Diego, California, USA, 1992.
- [14] 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.
- [15] M. Zaïait, D. Florescu, and P. Valduriez. Benchmarking the DBS3 Parallel Query Optimizer. *IEEE Parallel and Distr. Techn.: Systems and Appli.*, 4(2):26–40, Summer 1996.
- [16] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *ACM SIGMOD Conf.*, Montral, Canada, June 1996.

- [17] R. S. G. Lancelotte 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.
- [18] H. Lu K.-L. Tan and S. Dao. The fittest survives: An adaptive Approach to Query Optimization. In *Proceedings of the International Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
- [19] M. Groetschel and L. Lovasz. *Combinatorial Optimization*, chapter 28. Handbook of Combinatorics. Elsevier Science B.V., 1995.
- [20] M. Mehta and D. J. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal*, 6(1):53–72, January 1997.
- [21] D.J. DeWitt J. Naughton and J. Burger. Nested Loops revisited. In *Proceedings of the Parallel and Distributed Information Systems*, San Diego, California, USA, January 1993.
- [22] C. Chekuri W. Hasan and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *ACM PODS Conf.*, San Jose, California, USA, May 1995.
- [23] 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.
- [24] A. N. Wilschut J. Flokstra and P. M. G. Apers. Parallel Evolution of Multi-Join Queries. In *ACM SIGMOD Conf.*, San Jose, California, USA, May 1995.
- [25] K.A. Hua M. Lo and H.C. Young. Considering Data Skew Factor in Multi-Way Join Query Optimization for Parallel Execution. *Very Large Databases Journal*, 2(6), March 1993.
- [26] H. Lu B.-C. Ooi and K.-L. Tan, editors. *Query Processing in Parallel Relational Database Systems*, chapter Parallel Query Optimization. IEEE Computer Society Press, 1994.
- [27] N. Biscondi A. Flory L. Brunie and H. Kosch. Encapsulation of intra-operator parallelism in a parallel match operator. In LNCS 1127 Springer, editor, *ACPC 96*, September 1996.
- [28] Harald Kosch. *Exploiting parallel and serialized bushy tree space for parallel query optimization*. PhD thesis, Ecole Normale Sup rieure de Lyon, Lyon, France, 1997. to appear.
- [29] Trans. Processing Performance Council. TPC BENCHMARK D Standard Specification. Technical report, TPC, December 1995. Available at the WWW address: "http://www.tpc.org/".