TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer

Clara Nippl
nippl@in.tum.de

Bernhard Mitschang
mitsch@in.tum.de

Technische Universität München, Institut für Informatik, D - 80290 Munich, Germany

Abstract

Currently the key problems of query optimization are extensibility imposed by object-relational technology, as well as query complexity caused by forthcoming applications, such as OLAP. We propose a generic approach to parallelization, called TOPAZ. Different forms of parallelism are exploited to obtain maximum speedup combined with lowest resource consumption. The necessary abstractions w.r.t. operator characteristics and system architecture are provided by rules that are used by a cost-based, top-down search engine. A multi-phase pruning based on a global analysis of the plan efficiently guides the search process, thus considerably reducing complexity and achieving optimization performance. Since TOPAZ solely relies on the widespread concepts of iterators and data rivers common to (parallel) execution models, it fits as an enabling technology into most state-of-the-art (object-) relational systems.

1 Introduction

The MIDAS project [BJ+96] concentrates on optimization, parallelization and execution aspects of queries coming from areas such as OLAP, DSS, and document management systems [CJ+97]. Generally, the responsibility for query parallelization is taken over by the so-called parallelizer. Its task is to come up, without incurring too much overhead, with a parallel query execution plan (PQEP) that exploits various forms of parallelism, thereby achieving maximum speedup combined with lowest resource consumption. Among the most important requirements to be fulfilled by modern parallelizer technology are the following ones:

- *Extensibility:* This requirement is stressed by forthcoming application scenarios. Necessary SQL extensions are dealt with using concepts like user-defined functions [JM98] or designated (internal) operators [NJM97]. Similar issues are treated in the context of (parallel) object-relational database systems [SM96].
- *Performance:* To conquer the complexity of the parallel search space [GHK92], [LVZ93], accurate pruning techniques are necessary.
- *Granularity of Parallelism:* A QEP consists of operators showing significantly dissimilar processing costs. Currently, cost models as well as parallelization strategies only deal with high-cost operators (also called *coarse-grain* operators [GGS96] , [GI97]), as the performance speedup by means of parallelization is most profitable for this kind of operators. However, low-cost operators, if not treated the right way, can deteriorate query execution performance considerably. Hence, a flexible granularity of parallelism that provides a comprehensive treatment of low-cost and high-cost operators is necessary.
- *Economical Resource Allocation:* The maximum speedup obtainable is delimited by the critical path of the best execution schedule for the PQEP. Hence, to limit resource contention, no resources should be allocated to subplans that cannot reduce this measure. This demand is particularly important in the case of queries that run over longer periods of time (e.g. DSS queries) and in multi-user environments.
- *Comprehensiveness:* In order to generate PQEPs of acceptable quality for all query types it is necessary to take into account all forms of intra-query parallelism.
- *Adaptability*: Hybrid (or hierarchical) system architectures are gaining popularity. However, the development of optimization techniques to exploit their full potential is still an open problem [Gr95], [BFV96].

Obviously, numerous techniques have to be devised and combined in order to meet all of the above listed requirements. As rule-driven optimizers [Lo88], [Gra95] have already proved to be extensible and efficient, we rely on that technology also for parallelization and propose a solution based on a top-down search strategy, called TOPAZ

Proceedings of the 24th VLDB Conference
New York, USA, 1998

(**TO**p-down **PA**ralleli**Z**er). TOPAZ relies on a fully cost-based decision making. To reduce complexity, it splits parallelization into subsequent phases, each phase concentrating on particular aspects of parallel query execution. Moreover, by means of a global pre-analysis of the sequential plan first cost measures are derived that are used to guide and restrict the search process. TOPAZ combines high-cost as well as low-cost operators into so-called blocks that are subject to coarse-grain parallelism. This cost-based block building achieves economical and efficient resource utilization. The rule-driven approach provides for the necessary abstraction to support different architecture types, including hybrid ones. Since TOPAZ solely relies on the widespread concepts of iterators and data rivers common to (parallel) execution models, it fits as an enabling technology into most state-of-the-art (object-) relational systems. This is additionally supported by the fact that TOPAZ builds upon the Cascades Optimizer Framework [Gra95], which is also the basis of some commercial optimizers, like Microsoft's SQL Server [Gra96] and Tandem's Serverware SQL [Ce96].

In this paper, we analyze the possibilities of integrating parallelization into established query optimization search engines and point out limitations of the heuristics used in state-of-the art parallelizers. A discussion of related work is given in Section 2. Section 3 describes the architectural embedding of TOPAZ into our testbed parallel database prototype MIDAS and a sample query taken from the TPC-D benchmark is introduced as a running example. The design of TOPAZ in terms of basic parallelization strategies as well as internal optimization and control measures is detailed in Section 4. In Section 5 the main phases of the parallelization task are described. An analysis and evaluation of the technology incorporated into TOPAZ is provided in Section 6. Finally, in Section 7 a conclusion and an outlook to future work completes the paper.

2 Review of Parallel Query Optimization Techniques

Query optimizers use continuously improved techniques to simplify the task of extending functionality, making search strategies more flexible, and increasing efficiency [HK+97], [PGK97], [ON+95]. In a parallel context, the search space becomes even more complex as resource utilization has to be considered as well. One way to tackle this problem is to develop specialized solutions for the exploration of the parallel search space. The other approach is to reuse existing sequential optimization techniques and to extend them by special heuristics.

2.1 Specialized Parallelization Techniques

The ***two-phase*** approach uses a traditional search strategy for optimization and a specialized one for parallelization of queries. The parallelization task is based in many cases on heuristics [HS93] or it is restricted to join orderings in combination with restricted forms of parallelism [Ha95]. As the parallelizer is a separate system, easy reuse of infrastructure or technology improvements as mentioned above is prohibited. In addition, extensibility, as required by object-relational extensions, is limited.

Other approaches propose an integration of the optimization and parallelization, but are still highly specialized for specific architectures or certain operator types. Some research concentrated on scheduling of joins that maximizes the effect of specific forms of parallelism. Thus concepts as ***right-deep*** [SD90], ***segmented right-deep*** [LC+93], ***zig-zag*** [ZZS93] and ***bushy*** trees [WFA95] have been elaborated. Although these strategies have achieved good performance for specific scenarios [STY93], they rely on the hash-join execution model and thus cannot be applied in a straightforward way to complex queries holding any kind of operators.

2.2 Parallelization Using Traditional Sequential Optimization Techniques

If the optimizer and the parallelizer are integrated, they both use the same search strategy. However, they differ in exploration of the search space and the size of the portion explored. Due to its exponential complexity, ***exhaustive*** search is only feasible for very simple queries and is implemented in few research DBMSs, mainly for performance comparison purposes. ***Randomized*** algorithms have been mainly tested for join orderings in both the parallel and sequential context [LVZ93], [KD96], showing efficiency only for a high number of base tables. The best-known ***polynomial*** algorithm, the greedy paradigm [LST91], explores only a small portion of the search space, often ignoring some forms of parallelism. Thus, it is likely to fail the regions of good physical operator trees.

The ***bottom-up (dynamic programming)*** algorithm works iteratively over the number of base tables, constructing an optimal physical operator tree based on the expansion of optimal subtrees involving a smaller number of tables. To handle exponential complexity, a pruning function is introduced to realize a branch and bound strategy, i.e. it reduces the number of trees to be expanded in the remaining of the search algorithm. Pruning is achieved by comparing all subtrees which join the same set of relations with respect to an equivalence criteria and then discarding all trees of non-optimal cost. In order to tackle the even higher complexity of a parallel context, many PDBMSs use beside dynamic programming pruning also other simplifying heuristics. Thus, in DB2 Parallel Edition [BF+95], [JMP97], the degree of parallelism (DOP) of the operators is mainly determined by the partitioning of the inputs. In Teradata and Oracle's Parallel Query Option [BF97], [Or98] it even remains the same for the whole plan.

In ***top-down*** optimizers, such as Volcano [Gra94], Cascades [Gra95] and Columbia [SM+98], the complexity explosion is also controlled by pruning. These optimizers compute costs for high-level plans before some lower-level plans are examined. These cost limits are then passed down to optimization of incrementally constructed QEPs and can thus prune plans whose predicted total costs exceed this limit. Some investigations [KD96] have yielded poor performance for top-down optimizers. How-

ever, these results referred to the Volcano search strategy, that meanwhile got improved in the new generation of top-down optimizers, especially concerning pruning techniques [Gra95], [SM+98].
W.r.t. parallelization, in Volcano the best sequential plan found is divided into several segments bracketed by so-called *Exchange* operators. Please note that in this way parallelization, i.e. the computation of the degree of intra-operator parallelism or the determination of the segment boundaries, is done with another search strategy than that of the Volcano optimizer. Although top-down optimizers are used in other PDBMSs [Ce96] as well, we do not know of any publicly available report on how to decide on parallelization using this type of search engine.
To overcome the shortcomings of each optimization strategy in combination with certain query types, also ***hybrid*** optimizers have been proposed [ON+95], [MB+96].
As optimizers based on bottom-up [Zou97], [HK+97], [JMP97] and top-down [Ce96], [Gra96] search strategies are both extensible [Lo88], [Gra95] and in addition the most frequently used in commercial DBMSs, we have concentrated our research on the suitability of these two techniques for parallel query optimization.
Generally, some crucial decisions in the parallel context refer to *physical* properties, as e.g. partitioning, degrees of parallelism and usage of resources, that have to be chosen in a way to guarantee overall efficiency and to minimize resource contention. Given the above, an advantage of the top-down search strategy is that it comes up very early with physical tree solutions, whose cost estimates can be used to perform a global plan analysis and to guide further parallel search space exploration. More details on this topic can be found in [NM98]. Thus, beside the quality of the plans also the performance of the parallelization task itself can be improved considerably. For the MIDAS project, it was important to first concentrate on the strategies needed to achieve these goals. Hence, for a first version of TOPAZ we decided to have as input a complete sequential plan that is generated by a top-down sequential optimizer. Thus, optimizer and parallelizer use the same (top-down) search strategy, i.e. Cascades, but explore different search space regions with different rule sets. In the following, we assume that the top-down optimization technique is well understood and thus concentrate only on the parallelization effort.

3 Architectural Embedding of TOPAZ into MIDAS

In this section, we shortly present some of the underlying concepts of parallel query execution in MIDAS as far as it is necessary for further understanding.

3.1 System Environment

MIDAS [BJ+96] is a prototype of a parallel database system running on a hybrid architecture comprising several SMP nodes combined in a shared-disk manner. To provide the necessary abstraction, it is important to decouple optimization from some scheduling and load balancing aspects. This is achieved in MIDAS through parameters. Thus, the goal of TOPAZ is to come up with a parameterized PQEP. Each parameter keeps track of particular plan properties whose final adjustment has to be made according to the run-time system state, e.g. degrees of parallelism and resource consumption. Since TOPAZ relies on a cost-based approach, it comes up with reasonable lower and upper bounds for these parameters. This allows the run-time component, called Query Execution Control (QEC), to derive individual PQEPs having fixed parameter values. Thus, QEC comprises the run-time responsibilities of load balancing, scheduling, and resource allocation. It performs a fine-tuning of the PQEP and schedules different portions of the PQEP to different *execution units*.

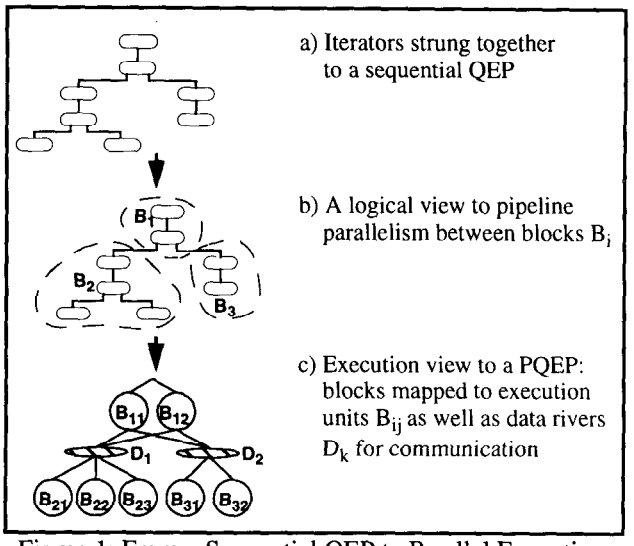


Figure 1: From a Sequential QEP to Parallel Execution Using Blocks and Data Rivers

The MIDAS operators are self-contained software objects designed according to the *iterator* (or *Open-Next-Close*) processing model [GB+96], [Gra94]. In this model queries are structured by bundling the appropriate operators (iterators) into a QEP (Figure 1a). In a sequential DBMS, each QEP is processed by a single execution unit. In the course of parallelization, this QEP is broken down into several subplans or *blocks* [TD93] (Figure 1b) that define the granularity or unit of parallelism [HS93]. A block can be considered as a single operator whose processing cost is the sum of the processing costs of the constituting operators. In order to obtain optimal speedup, TOPAZ performs a cost-based analysis to identify the optimal granularity for parallelism, i.e. number of blocks for a given query, number of operators within a block, and degree of parallelism assigned to a block. The corresponding strategies are presented in Section 4.
As determined by TOPAZ and at run-time adjusted by the QEC, a block is assigned to one or several execution units, according to its degree of parallelism. The flow of tuples among the various execution units is organized by the concept of *data rivers* [Gr95] (see Figure 1c). If multiple producers add records to the same river that is consumed by consumers, the river consists of several data streams. In

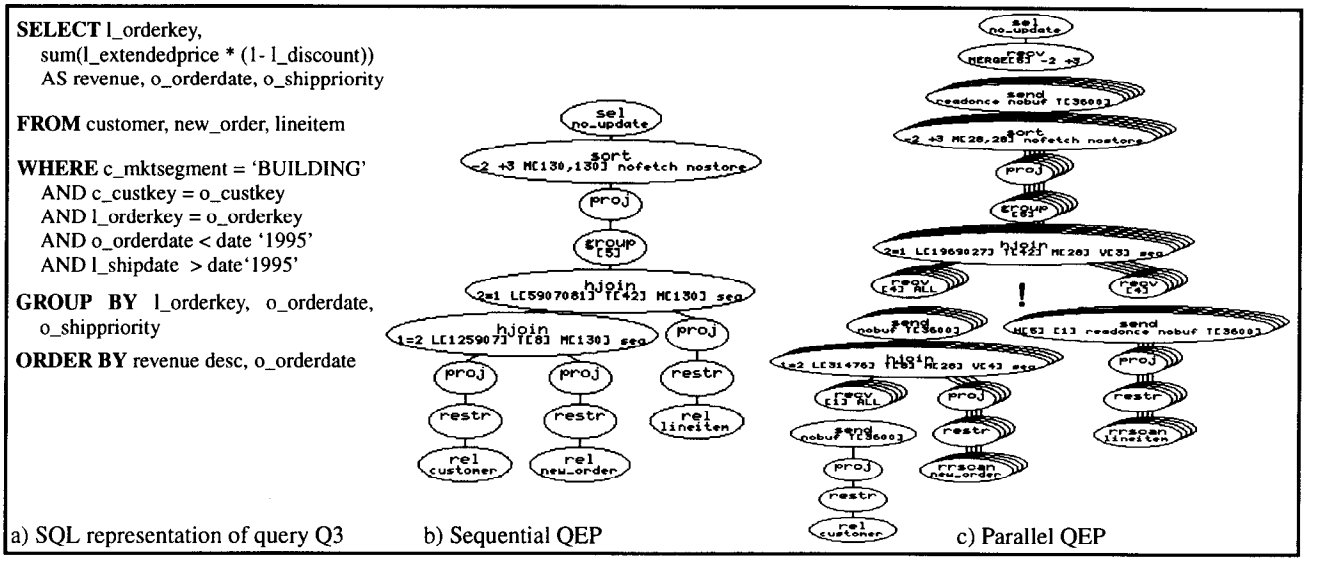


Figure 2: Parallelization of Query Q3 of the TPC-D Benchmark

this way, parallelism is transparent for operators applying the iterator processing model, as they still operate sequentially on these data streams that constitute the data river. In MIDAS, the data river paradigm is realized by means of two new communication operators: *send* and *receive*. These follow the same iterator concept as all the other operators and are implemented to write respectively read from a data stream. Thus *send* and *receive* realize all communication patterns necessary for intra-query parallelism: pipelining, replication, partitioning (a single flow of tuples into several data streams), merging (several data streams into a single flow of tuples) and repartitioning. Hence the execution model of TOPAZ only relies on the two basic concepts: ***data rivers*** and ***iterators***. Thus, the approach is applicable to all engines that refer to the same model. To our knowledge, the iterator protocol is used in many (object-) relational DBMSs [GB+96], [Zou97]. Our communication operators implementing the data river concept are similar to other approaches as well, e.g. the *send/ receive* operators in DB2 PE [JMP97], the *split/merge* operators in Gamma [De+90], or the *Exchange* operator in Informix Online XPS and Volcano [Gra94], [Zou97].

3.2 Running Example

We will exemplify the parallelization using query Q3 from the TPC-D benchmark [TPC95], whose SQL representation is given in Figure 2a. The sequential execution plan of this query that serves as input for TOPAZ is depicted in Figure 2b. Essentially, it consists of a 3-way join (performed by 2 hash joins on the tables CUSTOMER, NEW_ORDER, and LINEITEM) followed by a complex aggregation. Some operators, e.g. the *sort* and the *send*, show certain parameters defining memory allocation, buffer management etc. These parameters are set by the parallelizer according to the cost model, but can be adjusted by QEC at run-time according to the system state. In this scenario we further assume that the tables are physically partitioned across 4 disks in a round-robin manner. The resulting PQEP can be found in Figure 2c. Please note that in this representation, the operators bracketed by *send* and *receive* (*recv*) nodes are bundled together to a block. Between *send* and *recv* nodes there are data rivers consisting of several data streams that are for simplification reasons not depicted in the PQEP visualization.

4 TOPAZ Strategies

In the following, we describe in detail some of the core strategies of TOPAZ. The PQEP in Figure 2c already shows some of the intrinsic characteristics resulting from our parallelizer that are quite different to the ones known from other approaches already mentioned in Section 2:

- usage of all possible communication patterns to realize efficient intra-query parallelism
- cost-related degrees of parallelism and adjusted block sizes, saving scarce resources
- parameters allowing a fine-tuning of the execution plan to different application scenarios.

4.1 Control of the Search Space

Exponential complexity [OL90] has forced optimizers to use different techniques to restrict the search space and to improve performance. One of these techniques is to prune expressions that cannot participate in the final, best plan. However, traditional optimization metrics are not sufficient for parallel search spaces [GHK92], because, contrary to sequential optimization, physical resources, partitioning strategies, and scheduling play a vital role. A pruning strategy that doesn't take into account these aspects risks to miss the best parallel plan. Heuristic solutions, as extending the traditional pruning criteria by "interesting partitionings" are also insufficient, as shown in [NM98]. The solutions proposed in [GHK92] and [LVZ93] refer to extensions of the optimization metric that account also for resource utilization. Thus, the costs for a

single QEP fill up a vector, and a multidimensional "less-than" is needed to prune the search space. The problem with these approaches is that dynamic programming pruning techniques become generally ineffective and optimization effort explodes in terms of time and memory consumption, as it becomes comparable to exhaustive search. Recent work [GGS96], [GI97] propose a more relaxed cost metric that is based on approximations taking into account some global parameters as critical path length or average work per processing site. To our knowledge, there exists no published work on how to incorporate these cost metrics into existing search engines.

Our solution to these problems comprise the following extensions to top-down optimization:

1. Cost Model The strategies proposed in [GHK92], [LVZ93] are known to assure correct pruning. Based on these results, our cost model comprises besides CPU-costs also communication costs, memory usage, disk accesses, and blocking boundaries[1]. In addition, rather than extending the search space to explore alternative plans holding different degrees of parallelism, these degrees are also incorporated into the cost model. Thus, the global processing costs of an operator, i.e. the sum of the costs of its inputs plus the operator's local processing costs, are calculated for different degrees of parallelism and maintained in an array (see Section 4.4).

2. Phases To overcome the drawback of poor optimization performance due to inefficient pruning, parallelization is split into different phases, each phase concentrating on particular aspects of parallel execution. The first phases focus on global optimization of the entire plan w.r.t. dataflow, execution time, and resource utilization. This allows the parallelizer to take global dependencies into account, detecting those locations in the plan where the benefit in exploiting some forms of parallelism is maximized. In the subsequent phases decisions are based on a local view of the QEP, i.e. a view restricted to only one operator or a block of operators and the costs involved in their execution. Another way to express this strategy is that each phase uses as a starting point the result of the previous one to expand a specific region of the search space. These regions do not overlap, since they are expanded using different transformations, i.e. different rule sets. However, the size of the explored search space regions decreases in each phase, as they refer to successively refined aspects of parallel query execution. The final refinement is made by the QEC; it can further adjust certain parameters, like memory usage, degree of parallelism etc. according to the run-time environment. Thus the overall approach to handling the huge search space for parallelization in MIDAS is neither enumeration nor randomization but a cost-based *multi-phase pruning*. This strategy is detailed in Section 5.

3. Pruning Package (*ParPrune*) Global parameters [GGS96], [GI97] are incorporated in TOPAZ by means of an additional pruning strategy. *ParPrune* further limits the complexity in each phase, as it guides the search only towards promising regions of the search space. It works in combination with the top-down search engine and consists of two parts: first, in the course of a pre-analysis different global measures are calculated: critical path length, expected memory usage, average costs per CPU, average operator costs etc. Second, these measures serve as constraints (i.e. conditions for rule activations) for all subsequent parallelization phases. Apart of the fact that this strategy reduces the optimization effort itself, it can in some cases influence also the quality of the final plans, as e.g. the global pre-analysis permits a better estimation on the search space regions that are worthwhile to be explored in more detail.

1. These refer to particular locations within query execution, where the complete intermediate result table has to be derived before the next operation can start.

4.2 Control of the Granularity of Parallelism

Prior work on parallel execution and cost models as well as scheduling rely on the assumption that the QEP is *coarse-grain*, i.e. the parallelization overhead for each operator exceeds only up to a specific factor the total amount of work performed during the execution of the operator [GI97], [GGS96], [DG92]. However, this requirement is not always assured by practical database execution plans. An example coming from traditional QEPs is a restriction evaluating only a low-cost predicate. Some PDBMSs have solved this problem using heuristics, as e.g. parallelizing these operators always together with their predecessors. However, in PORDBMSs this is not possible if e.g. a user-defined predicate or low-cost aggregation requires a special partitioning strategy. It is an open problem how to deal with these operators. Parallelizing them separately causes obviously too much overhead, while a sequential execution can cause bottlenecks at several places of the PQEP and thus suboptimal performance. This is confirmed also by the measurements presented in Section 6.

Our response to this problem is *cost-based block building*. This strategy accounts for operator costs, selectivities, and intermediate result sizes to construct *coarse-grain blocks*, i.e. to perform several operators within a single execution unit. Moreover, these can be used for further block-building in order to achieve mutually adjusted processing rates among communicating, i.e. neighboring blocks. If the rate at which tuples are sent to an execution unit is much higher than the rate at which tuples can be processed, the communication buffer can overflow, forcing the sender to block. On the other hand, if the rate at which tuples are received is much lower than the highest possible processing rate, the corresponding execution unit will frequently be idle and will waste non-preemptive system resources, as e.g. memory. Hence, mutually adjusted processing rates are prerequisite to efficient pipelining [MD95]. Additionally, through block construction intermediate result materialization and inter-process communication between operators can be avoided. This implies savings in main memory or even I/O costs.

Intra-block parallelism is analogous to intra-operator parallelism and requires to execute several instances of the complete block by different execution units. Each instance has to work on different sets of data, i.e. the processing

within one instance of the block is independent from all the other instances of this block. In the PQEP shown in Figure 2c, the largest block is formed by the *sort*, projection (*proj*), *group*, and hash-join (*hjoin*) operators having a DOP of 5.

The necessary conditions to bundle operators within a block are: ***same degrees of parallelism*** and ***same partitioning strategies***. Thus, in order to achieve efficient block building, a flexible control of these properties is necessary, as described in the following sections. However, these conditions are not sufficient. A cost-based analysis has to decide if a specific block construction also leads to a decrease of the overall processing costs.

4.3 Control of Partitioning Strategies

In order to have the necessary degrees of freedom TOPAZ distinguishes between logical and physical data partitioning. The strategies for physical data partitioning implemented in MIDAS are ***round-robin, hash*** and ***range partitioning***. We are in the process of implementing ***user-defined*** partitionings as well. Which of the above mentioned techniques is used depends on the type of the operator that has to be parallelized. In many cases, the partitioning has to keep track of the attribute values, like in the case of hash- or range-based partitioning. For instance, in Figure 2c the *send* operator highlighted by an exclamation mark performs a hash partitioning on the first attribute into 5 partitions as indicated in the operator description by the parameter *H[5] [1]*. However, TOPAZ differentiates only between the following logical partitionings:

- ***Any***: This parameter indicates that the parallelized operator (or block) doesn't necessarily need a specific partitioning (as e.g. the *sort* operator).
- ***Attr***: If an operator needs a value-based partitioning on certain attributes (as e.g. in the case of certain aggregations), the corresponding *send* operator is extended by the *Attr* parameter together with the identifiers of the required partitioning attributes.

Thus, if a block construction becomes necessary in the course of parallelization, TOPAZ can change a less strict partitioning (like *Any*) into a stricter one (like *Attr*). This can be done easily, only by taking into consideration the required physical properties. At the end of the parallelization, when block construction is finalized, these logical parameters are mapped to one of the above mentioned physical partitioning strategies.

4.4 Control of the Degrees of Parallelism

Consider a QEP having two adjacent high-cost operators. In Figure 3 (left), these are the final phase of a *sort* (merging of sorted runs) and an aggregation operator (*group*). As both of them are coarse-grain, both are processed using intra-operator parallelism. Suppose that by taking into account only the local costs of the operators and the intermediate result sizes, the best degree of parallelism for the *sort* operator results to 3 and that for the aggregation is 2. Due to the different degrees of parallelism, a repartitioning of the intermediate results of the *sort* operator is necessary,

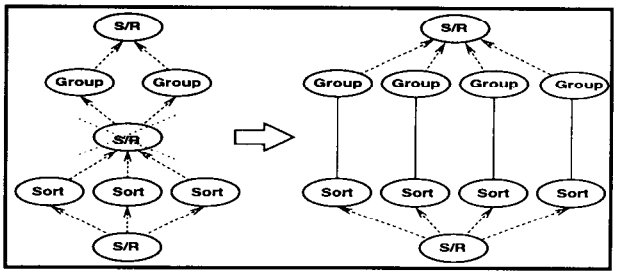


Figure 3: Adjusting the DOP for Block Construction

implicating high communication costs.

If the degree of parallelism of the *group* is increased to 3, pipelining between the two operators becomes possible. This reduces communication costs, but increases the number of execution units from 5 to 6. Actually, the optimal execution for the two operators would be within the same block, but with an increased degree of parallelism according to the higher block processing costs, as shown in Figure 3 (right). This implies less execution units and less communication costs, as only the aggregated result of the *group* has to be transmitted. A plan with similar response time but reduced resource consumption is also more suitable for a multi-query environment.

Considering e.g. a bottom-up optimizer, it first optimizes the *sort*, finding the best degree of parallelism of 3 and prunes all the other plans, as they are (locally) more expensive. At the next level, when optimizing the *group*, the search engine cannot find the best plan shown in Figure 3 (right), because the search space doesn't contain the plan and costs for the *sort* operator in combination with a degree of parallelism of 4. However, keeping the plan alternatives for all possible DOPs is also an impractical solution with regard to optimizer performance.

We have elaborated the following solution to this problem: To keep the degrees of parallelism flexible, TOPAZ incorporates this aspect only in the cost model, without explicitly extending the search space with alternative plans that differ only in the degrees of parallelism. If an operator gets parallelized by partitioning its inputs, the corresponding *send* operator doesn't hold any specific information on the number of partitions. A parameter like *"Attr[2] 1"* in the course of the parallelization only means that this *send* operator performs a value-based partitioning on the first attribute and that the number of partitions is greater or equal 2. At the same time the costs of the operator are calculated for all possible degrees of parallelism, storing them in an array. This cost calculation is propagated upwards. The global processing costs of the successor can also be calculated correctly for different DOPs, since its local processing costs are known and the processing costs of its input are available for every considered DOP. Thus, e.g. the decision on combining two blocks can be taken on the basis of the lowest value in the cost array of the topmost operator. In the example, this is the *group* and the entry in its cost array corresponding to the minimal global processing costs will be found for a DOP of 4.

In Section 2, we have already mentioned some heuristics used in practice, as e.g. choosing the same DOP for the

whole PQEP or limiting the considered degrees to a few alternatives [BF97], [Or98], [JMP97]. New query types, as e.g. DSS and object-relational ones, make the usage of CPU-intensive operators and UDFs more and more popular. In these scenarios, the operator costs in a QEP can differ significantly. We believe that the degree of parallelism for these operators can rely only on cost-based decisions, as in TOPAZ, whereas using only restricted heuristics like the ones mentioned above can lead to truly suboptimal parallel plans.

5 Multi-Phase Parallelization

In the following we describe the parallelization phases that exploit the strategies described in the previous section, using as example the TPC-D query Q3 (Figure 2a). Please note that the PQEPs presented in each phase are complete physical trees, having specific data partitionings and degrees of parallelism, although we mentioned before that these aspects are kept flexible. In TOPAZ each phase can be separately turned on or off. Thus the following examples rather reflect the physical trees that are obtained if the phases are turned on successively, starting with the sequential one (Figure 2b). We accentuate that this is only for illustration purposes, as the final parallel plan is the result of *all* constituting phases that explore different regions of the parallel search space.

As each phase is characterized by a separate rule set, examples of representative rules and of rule applications will be provided as well. Since the *send* and *receive* operators appear always in pairs, they are internally considered as a single operator, called *S/R*, holding the parameters for the respective *send (S...)* and *receive (R...)* part. However, in a physical plan, they are represented separately at the end and at the beginning of neighboring blocks, constituting a data river.

5.1 Phase 1: Inter-Operator Parallelism and Refinement of Global Costs

This phase starts from the sequential plan and analyzes the possibility of reducing the critical path length through inter-operator parallelism. An additional goal is to achieve a mutually adjusted processing rate over all blocks in the QEP, thus considerably reducing query execution overhead, as described in Section 4.2. The transformations considered in this phase expand a search space region containing alternative plans that exploit only pipelining and independent parallelism. The decision criteria comprise sizes of intermediate results, expected resource consumption, processing costs of the emerging blocks as well as blocking operators.

A naive strategy would be to define a single rule for insertion of *S/R* nodes and let the search engine find the optimal places for inter-operator parallelism according to the cost model. But this increases unnecessarily the parallelization effort, since alternatives that are unlikely to lead to the best plan are explored as well. For example, pipelining shouldn't be considered in combination with subplans that are not on the critical path. This naive strategy would lead

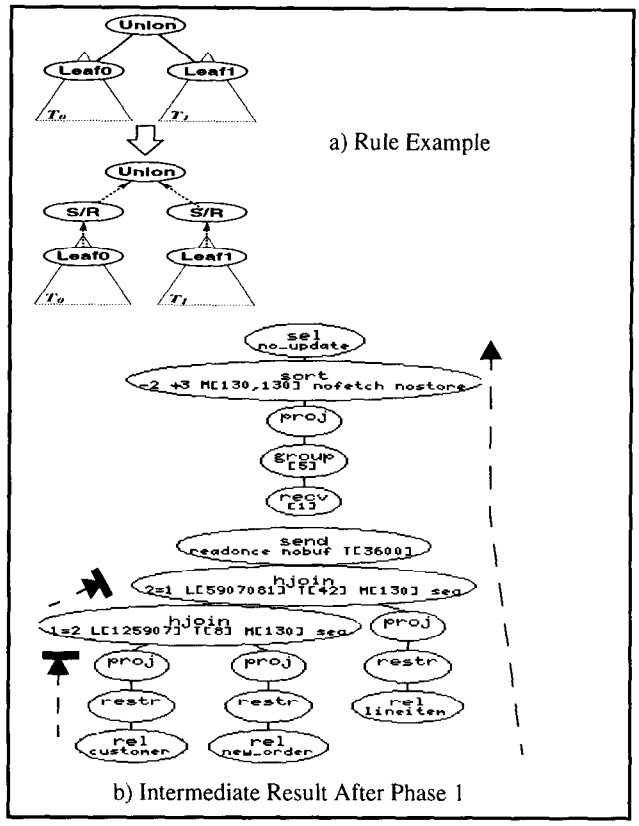


Figure 4: Inter-Operator Parallelism

already for this first phase to an unacceptable performance. Hence, the considered alternatives are restricted by *ParPrune*. In this phase it accounts for the relative costs of the operators and the critical path length computed during the pre-analysis. Thus, inter-operation parallelism is considered only in combination with certain subplans and operators that are reasonable from a global point of view. In Figure 4a, a rule for the insertion of pipelining *S/R* nodes below a binary operator is presented. The condition for the consideration of this transformation within a QEP is that both inputs *T0* and *T1* exceed certain cost limits.

Our example query resulting from this phase is presented in Figure 4b. As shown by the interrupted dashed arrows, the left inputs of the join operators are blocking, since they are used to build the hash tables. Hence, efficient pipelining is only possible in the segment marked by the continuous dashed arrow at the right side of the figure. In this segment, the *group* is recognized as a costly operator due to the size of the intermediate result and the (high) local processing cost. Thus only one pipelining edge has been identified, defining two blocks with similar processing rates. Please note that the goal of this phase is not to come up with the final set of edges for inter-operator parallelism. Due to modified cost proportions in the next phases, some of these edges may be replaced by neighboring ones. The result of this phase are refined cost limits that have been established w.r.t. critical path length and average block processing costs. These refined costs are exploited by the subsequent phases.

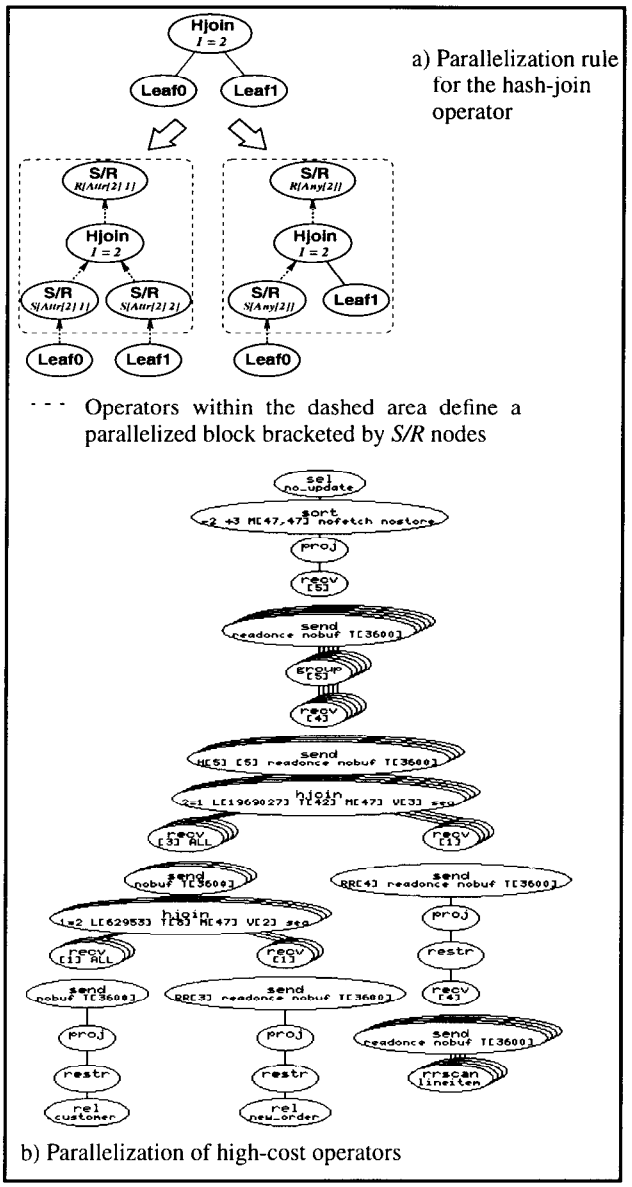


Figure 5: Intra-Operator Parallelism

5.2 Phase 2: Intra-Operator Parallelism applied to High-Cost Operators

The result of the previous phase is now used to span a new search space region, exploring the possibility of further reducing the critical path length and block processing costs by controlled introduction of intra-operator parallelism. Therefore operators that already meet the coarse-grain demand are individually parallelized, bracketing them with *send-receive* nodes.

Depending on the type of the operator, one or both inputs have to be partitioned. Hence, partitioning *send* nodes are inserted such that each operator instance processes one partition. The intermediate results produced by these instances are collected by a *receive* node that is placed at the output of the operator. For each operator separate parallelization rules have been defined, considering the operators' characteristics, as e.g. some operator types admit more alternatives. As shown in Figure 5a, e.g. a hash-join can be parallelized by partitioning both inputs or only one input combined with a replication of the other. The solution chosen by TOPAZ depends on the cost distribution in the QEP. For instance, if one of the inputs is replicated, there exists no requirement concerning the partitioning strategy of the other. Thus repartitioning can be omitted, an aspect that is especially beneficial if this input has a high cardinality. The *S/R* node parameters only indicate logical partitionings (*Attr* or *Any*) without specifying any concrete degrees of parallelism or physical partitioning strategies (see Section 4.3).

Global execution performance and critical path length are mostly influenced by nodes having high local processing costs. The effect of *ParPrune* in this phase is to take into account the average costs per operator computed during the pre-analysis and to consider only those operators that beside the coarse-grain requirement, also exceed a minimal cost limit.

In Figure 5b the result for our example query is shown, if parallelization is stopped at this stage. Thus it includes also tasks that are usually performed only after the last phase, like mapping from logical to physical partitioning strategies and setting of concrete DOPs. These have been chosen according to intermediate result sizes, local processing costs, and disk partitioning. The parallelized operators are the *group* (DOP=5), the two *joins* (DOP=4 and DOP=3), and the *scan* of the LINEITEM table (*rrscan*: round-robin scan) with DOP=4. The *group* requires a partitioning on the 5th attribute, as indicated by the parameter *(H[5] [5]...)* of the corresponding *send* operator, identifying a hash partitioning of attribute 5 into 5 buckets. In contrast to shared-disk or shared-everything architectures, in a shared-nothing environment the parallelization of the relation scans in this phase is restricted by the given physical disk partitioning strategies. This constraint can be modeled as an additional required physical property.

As a result of this phase simple blocks that hold one parallelized operator show up. The parallelization of these *driver nodes* impose certain physical properties, like data partitioning, degree of parallelism, and sort order that will bias the parallelization of the remaining operators.

5.3 Phase 3: Block Expansion and Treatment of Low-Cost Operators

Phase 3 analyzes the possibility of expanding the one-operator blocks obtained in the previous phase. The resulting blocks incorporate also operators that individually don't meet the coarse-grain requirement or have low processing costs. As shown in Section 4.2 this achieves a minimization of the resources needed to process the given set of operators and avoids bottlenecks. The DOPs are adjusted according to the block processing costs (see Section 4.4).

The corresponding search space region is expanded by transformations that slide the *S/R* operators towards not yet parallelized operators, thus including them into existing blocks. If in the course of this sliding two *S/R* nodes meet, they are transformed into a single repartitioning

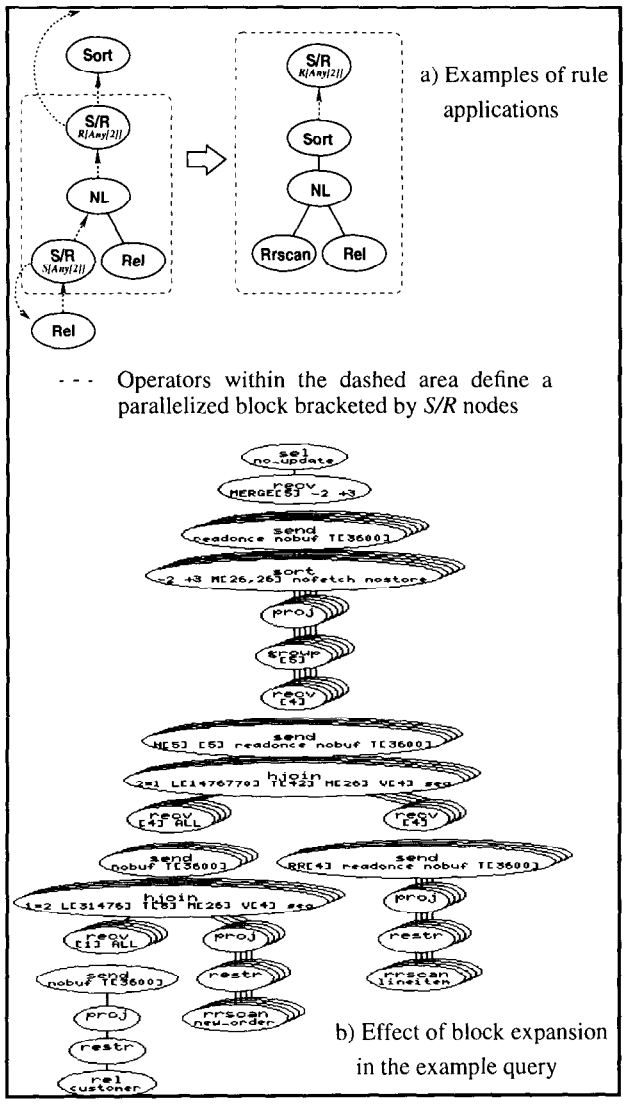


Figure 6: Block Expansion

node. In Figure 6a, a situation is shown where the nested-loop operator *(NL)* has been parallelized in Phase 2 by repartitioning an input and replicating the other. In Phase 3, one of the *S/R* nodes is pushed up. As a result, the *Sort* operator becomes part of the block taking over the parallelization decisions and properties of that block. The other *S/R* node is pushed downwards, thus parallelizing the relation scan (*Rel* operator). This transformation is specific to shared-disk and shared-everything architectures, expressing an additional degree of freedom compared to shared-nothing environments. As stated before, in the latter case the scans have to be parallelized in Phase 4, accounting also for physical disk partitionings. The result of the two transformations is a block consisting of the 4 operators, having the same DOP and the same partitioning strategy. The *rrscan* operator, a parallel scan, reads different partitions of the first input table in each block instance. The *Rel* operator reads the entire second input table and replicates it to all block instances.

All of the above mentioned transformations, e.g. pushing an *S/R* node through an operator, merging of two neighboring *S/R* nodes into a single repartitioning node etc., are defined as rules, the resulting plans being added to the search space and maintained according to cost-based decisions. To reduce the number of worthless transformations, *ParPrune* for instance checks in advance if a given partitioning strategy can be taken over by a candidate operator. In our example query (Figure 6b), the parallelization of the lower *hash join* has been extended downwards, parallelizing also the *scan* of the NEW_ORDER table and adjusting the DOP of the block from 3 to 4. The parallelization of the LINEITEM *scan* and the *group* have been extended upwards. Due to low processing costs, the *scan* of the CUSTOMER table is done sequentially, however replicating the result for further parallel processing.

5.4 Phase 4: Block Combination Further Decreasing Parallelization Overhead

As described in Section 4.2, bundling coarse-grain blocks can lead to a further reduction of resource utilization and intra-query communication, thus contributing even to the decrease of the critical path length. Therefore, the last parallelization phase analyzes the possibility of combining adjacent blocks with comparable partitioning strategies.

Phase 4 operates with a single rule for the elimination of repartitioning nodes between two adjacent blocks. This is only possible if the partitioning strategy of the candidate blocks is equal or comparable. As shown in Section 4.3, this condition is satisfied if e.g. the logical partitioning of at least one block is *Any*. For the final plan shown in Figure 2c, the *group* block has been bundled together with the upper *hash join* block adjusting the DOP to 5. The required partitioning imposed by the *group* has been taken into consideration by modifying the partitioning of the join block from round-robin (*send(RR[4]...)*) to hash (*send(H[5] [1]...)*), as highlighted by the exclamation mark. Hence, repartitioning has been pushed down to be performed *before* the join operator, where it is more beneficial w.r.t. intermediate result sizes. This proves again that TOPAZ keeps track of all cost factors also on a global level.

6 Performance Investigation

The TOPAZ data and cost models have been implemented using the Cascades Optimizer Framework [Gra95]. The current version has approximately 80 rules, divided into 4 categories, one for each parallelization phase. We have validated our approach by using different applications, such as OLAP, DSS, and digital libraries. In this section, we report on the performance of TOPAZ by using a series of TPC-D queries performed in a single-user environment on a 100 MB database, running on a cluster of 4 SUN-ULTRA1 workstations with 143 MHz Ultra SPARC processors, connected via a Fast Ethernet network. In order to perform a detailed analysis of the separate parallelization phases, we took the result of each plan and executed it on our cluster. Figure 7 shows the average speedups obtained after each phase for all queries of the test series, parallelized for the 4 workstations; the speedup obtained by our

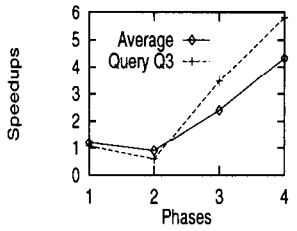


Figure 7: Speedups After Each Phase

Table 1: Speedups for the 16 Test Queries

# queries with superlinear speedup (4.5 to 13)	6
# queries with near-linear speedup (4)	5
# queries with sublinear speedup (1.5 to 3.5)	5

running example TPC-D query Q3 is illustrated in a separate curve. We would like to remind that this is for demonstration purposes, since parallelization is made up of all phases, the actual result being that obtained after Phase 4. The first two are only preparatory phases that result into the insertion of different forms of parallelism according to a global cost-based analysis (see Section 4.1). These are carried over in subsequent phases to the rest of the QEP, considering also physical properties in the top-down parallelization, as e.g. partitioning and sort orders (see Sections 5.3 and 5.4). These are the phases where the real speedups are achieved. In Phase 2 coarse-grain operators that significantly contribute to the critical path are parallelized separately. The negative speedup demonstrates quite dramatically our statement (Section 4.2) that ignoring non-coarse-grain operators causes bottlenecks in parallel execution, thus influencing negatively performance. The difference between Phase 2 and 3, respectively Phase 3 and 4 shows the importance of block construction, optimal setting of degrees of parallelism, and other TOPAZ strategies as described in Section 4.

Please note that in some cases, as e.g. for query Q3, we obtained superlinear speedup (see also Table 1). This is due to the fact that scaleup refers not only to CPUs, but also to other resources. Hence, if a query is parallelized correctly it can benefit also from parallel I/O facilities and from the increased database cache that can reduce disk spoolings. The results show the importance of incorporating these aspects into the cost model, as proposed by TOPAZ. Of course, this situation can change in a multi-user environment, due to general resource contention.

Table 1 shows also some sublinear speedups. As mentioned before, the implemented base version of TOPAZ gets as an input a complete sequential tree, produced by a sequential optimizer. As TOPAZ doesn't perform any rewrites, this can influence the quality of the final parallel plan. We observed that the suboptimal speedups are mostly related to queries containing a correlation, with this property preventing an efficient parallelization. However, we have never observed a deterioration w.r.t. the (sequential) performance, as all TOPAZ strategies account for parallelization overhead and thus introduce parallelism only where it is truly beneficial.

W.r.t. the importance of a global view in the parallelization process, we have parallelized and executed the queries with and without the *ParPrune* technique that can be easily switched on or off in our prototype. As described in Section 4.1, *ParPrune* is used to provide an additional

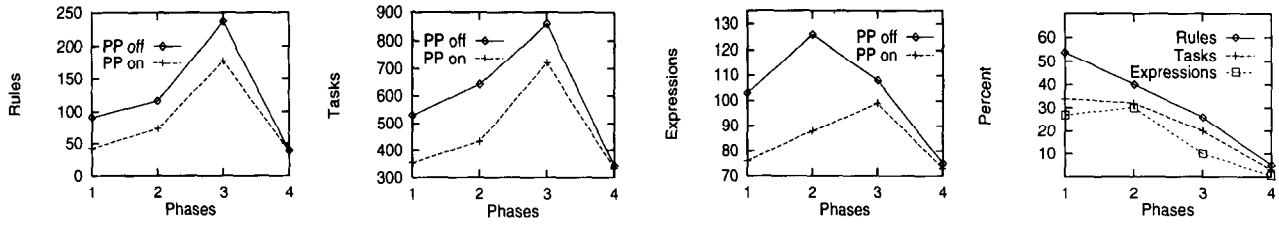
Table 1: Effect of Pruning and Global View on Execution and Parallelization

Resource and response time metrics	ParPrune off	ParPrune on
Average execution time for modified queries (ms)	25943	23717
Average number of execution units for modified queries	11.125	8.25
Overall average parallelization time (ms)	884	703

guidance throughout the parallelization phases. This is to reduce optimization complexity. However, as a side-effect, *ParPrune* can also improve the quality of the final plan as the global pre-analysis permits a better estimation on the search space regions that are worthwhile to be explored. In the test series, *ParPrune* modified the final plan in 50% of the test cases. As can be seen in Table 2, for these queries an additional performance improvement has been achieved. An interesting aspect is that this performance gain has been achieved with explicitly less resource consumption. We have only listed here the number of execution units, that in this way has been reduced by 34%. But even where *ParPrune* didn't come up with a more efficient plan, the best plan has been found with clearly less effort. This can be seen already by comparing the average parallelization times in the last row of Table 2. However, these numbers also include some organization overhead, as e.g. the time necessary to copy the QEPs into and out of the Cascades memory structure. Please note that the numbers are comparable to sequential optimization efforts. Internal program optimization will reduce this overhead further.

To evaluate only the search complexity, we have used as measures the number of expressions generated, the number of tasks and the number of rule applications in the course of the parallelization. Tasks are one of the basic mechanisms used by the Cascades search engine [Gra95]. They are used to perform a particular optimization objective, as e.g optimizing a single expression or group of expressions.

In Figure 8a, b, and c the average number of rules, tasks, and expressions participating in the each phase of the parallelization are compared. As can be seen, by using *ParPrune*, these numbers could be drastically reduced as compared to a non-pruned parallelization attempt. In order to get a better understanding, a summarization is given in Figure 8d, showing for each phase the reductions (in percent) achieved for these measures. Thus, e.g. the number of applied rules in the first phase is reduced drastically, by 54%. Generally, the impact of *ParPrune* is the highest in the first two phases, as these are the ones that participate most in the determination of the final character of the PQEP. It is here that a guidance given by a pruning strategy can help the most in finding the right regions of the search space. Later on only a gradual refinement of the parallel plan takes place that translates to a search only



a) Rules b) Tasks c) Expressions d) Differences in Percent

Figure 8: Influence of *ParPrune (PP)* on Rules, Tasks and Expressions Participating in Each Phase

around the regions found in the earlier phases. Thus, in these last phases pruning can only contribute to the reduction of unnecessary transformations and this impact is not so visible.

7 Conclusions and Future Work

In this paper we have shown that our approach, called TOPAZ, fulfills all basic requirements of a modern parallelizer. Its 'rule-driven' property guarantees for the necessary extensibility. Both language extensions and extensions to the database engine itself, as well as changes to the parallel system architecture can be accomplished by means of respective rules. Its 'multi-phase' property realizes an overall strategy that considers all forms of parallelism. It splits the parallelization task into subsequent phases, with each phase concentrating on particular aspects of an efficient parallel execution. In addition, this property turned out to be a major concept to handle the inherent complexity of parallelization. Its 'cost-based' property guarantees that all decisions w.r.t. investigating the parallel search space are cost-based. Hence, promising search space regions are explored to derive the best parallel plan. The concept of blocks enables (coarse-grain) parallelism to low-cost as well as high-cost operators. It further guarantees economical and efficient resource consumption. A prerequisite to optimization performance is pruning. The strategy developed for TOPAZ, called *ParPrune,* is exploited throughout the parallelization phases, within each focusing on valuable search space regions.

A thorough performance analysis and evaluation of our parallelizer technology clearly showed that the complex parallelization task can be conducted by TOPAZ's underlying parallelization strategies as well as internal optimization and control measures such as *ParPrune*. These measurements further indicate that the parallel plans created by TOPAZ are executable by state-of-the-art parallel database engines showing linear speedup. Our approach, i.e. TOPAZ embedded into MIDAS, has been validated also by other applications, such as OLAP and digital libraries, yielding similar speedup results even for very complex queries. In summary, our investigations manifested that these results can only be achieved by the integration of all beforementioned parallelizer properties.

For the implementation, we have used the Cascades Optimizer Framework. The running first version of TOPAZ uses the same top-down search engine for the optimizer and the parallelizer, but different models, one for the sequential execution space and one for the parallel one. As with the implementation and validation of TOPAZ this first phase of elaborating suitable parallelization strategies is finalized, we will further concentrate on analyzing and extending them to other scenarios as well. The primary focus is to integrate the models for optimization and parallelization. In our opinion, a combined approach, i.e. an optimizer taking into account some parallel aspects, followed by a detailed parallelization as described in this paper, will be the most suitable for forthcoming query scenarios. Another possibility is to use TOPAZ for hybrid optimizer solutions as well, e.g. to map logical trees, obtained by a bottom-up search strategy, to physical ones, similar to the NEATO optimizer [MB+96]. Here, the bottom-up search strategy is used to enumerate all join orders and the top-down strategy is used to perform the mapping from logical to physical operators in a parallel environment. Although only joins have been considered, optimization time was dominated by the mapping phase, due to the high number of possible mappings from logical operators to physical solutions in a parallel DBMS. We believe that the mapping problem becomes even more complex when new operator types, as e.g. UDFs, have to be considered as well.

Acknowledgments

The cooperation of the whole MIDAS project staff (esp. M. Jaedicke, M. Fleischhauer, and S. Zimmermann) is gratefully acknowledged as well as the support from Götz Graefe and from Leonard Shapiro and his research group.

References

[BJ+96] G. Bozas, M. Jaedicke et al: On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project, In: *Proceedings of the EUROPAR Conf.*, 1996.

[BF+95] C. K. Baru, G. Fecteau et al: DB2 Parallel Edition, In: *IBM Sytems Journal*, Vol 34, No 2, 1995.

[BF97] C. Ballinger, R. Fryer: Born to be Parallel, In: *Data Engineering Bulletin*, 20(2), 1997.

[BFV96] L. Bouganim, D. Florescu, P. Valduriez: Dynamic Load Balancing in Hierarchical Parallel Database Systems, In: *Proc. VLDB Conf*, India, 1996.

[Ce96] P. Celis: The Query Optimizer in Tandem's new ServerWare SQL Product, In: *Proc. VLDB Conf.*, India, 1996.

[CJ+97] A. Clausnitzer, M. Jaedicke et al: On the Application of Parallel Database Technology for Large Scale Document Management Systems, *Proc. IDEAS Conf.*, Montreal, 1997.

[De+90] D. DeWitt et al: The Gamma Database Machine Project, In: *TKDE* 2(1), March 1990.

[DG92] D. DeWitt, J. Gray: Parallel Database Systems: The Future of High Performance Database Systems, In: *CACM*, Vol.35, No.6, pp.85-98, 1992.

[GHK92] S. Ganguly, W. Hasan, R. Krishnamurty: Query Optimization for Parallel Execution, In: *Proc. SIGMOD Conf.*, San Diego, California, USA, 1992.

[GI97] M. Garofalakis, Y. Ioannidis: Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources, In: *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[Gra94] G. Graefe: Volcano-An Extensible and Parallel Query Execution System, In: *TKDE*, 6(1), 1994.

[Gra95] G. Graefe: The Cascades Framework for Query Optimization, In: *DE Bulletin*, 18(3), 1995.

[Gra96] G. Graefe: Relational Engine and Query Processing in Microsoft SQL Server, In: *Proc. of the Intl. Conf. on Data Engineering*, New Orleans, 1996.

[Gr95] Gray, J.: A Survey of Parallel Database Techniques and Systems, In: *Tutorial Handout at the Int. Conf. on Very Large Databases*, Zurich, 1995.

[GB+96] J. Gray, A. Bosworth et al: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals, In: *Proc. Intl. Conf. on Data Engineering*, New Orleans, 1996.

[GGS96] S. Ganguly, A. Goel, A. Silberschatz: Efficient and Accurate Cost Models for Parallel Query Optimization, In: *Proc. SIGACT-SIGMOD-SIGART Symp. on Principles of DB Systems*, Montreal, 1996.

[Ha95] W. Hasan: Optimization of SQL Queries for Parallel Machines, *PhD Thesis*, Stanford Univ., 1995.

[HK+97] L. Haas, D. Kossmann et al: Optimizing Queries across Diverse Data Sources, In: *Proc. of the 23rd VLDB Conf.*, Athens, Greece, 1997.

[HS93] W. Hong, M. Stonebraker: Optimization of Parallel Query Execution Plans in XPRS, In: *Distributed and Parallel Databases*, pp. 9-32, 1993.

[JMP97] A. Jhingran, T. Malkemus, S. Padmanabhan: Query Optimization in DB2 Parallel Edition, In: *Data Engineering Bulletin*, 20(2), 1997.

[JM98] M. Jaedicke, B. Mitschang: A Framework for Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS, *Proc. SIGMOD Conf.*, Seattle, 1998.

[KD96] N. Kabra, D. DeWitt: OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization, *Proc. ACM SIGMOD Conf.*, 1996.

[Lo88] G. Lohman: Grammar-like Functional Rules for Representing Query Optimization Alternatives, In: *Proc. of the ACM SIGMOD Conf.*, Chicago, 1988.

[LC+93] M.-L. Lo, M.-S. Chen et al: On Optimal Processor Allocation to Support Pipelined Hash Joins, In: *Proc. SIGMOD Conf.*, Washington D. C., 1993.

[LST91] H. Lu, M. Shan, K. L. Tan: Optimization of Multi-Way Join Queries for Parallel Execution, In: *Proc. VLDB Conf.*, San Mateo, USA, 1991.

[LVZ93] R. Lanzelotte, P. Valduriez, M. Zait: On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces, In: *Proc. VLDB Conf.*, Dublin, 1993.

[MB+96] W. McKenna, L. Burger et al: EROC: A Toolkit for Building NEATO Query Optimizers, In: *Proc. of the 22nd VLDB Conf.*, Mumbai, India, 1996.

[MD95] M. Mehta, D. DeWitt: Managing Intra-operator Parallelism in Parallel Database Systems, In: *Proc. VLDB Conference*, Zurich, , 1995.

[NJM97] C. Nippl, M. Jaedicke, B. Mitschang: Accelerating Profiling Services by Parallel Database Technology, In: *Proc. PDPTA Conf.*, Las Vegas, 1997.

[NM98] C. Nippl, B. Mitschang: TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer, TR, Technische Universität München, 1998.

[Or98] Oracle8 Parallel Server Concepts, Oracle Corp.

[OL90] K. Ono, G.M. Lohman: Measuring the Complexity of Join Enumeration in Query Optimization, In: *Proc. Intl. Conf. on VLDB*, Brisbane, 1990.

[ON+95] F. Ozcan, S. Nural et al: A Region Based Query Optimizer through Cascades Optimizer Framework, In: *DE Bulletin* 18(3), Sept 1995.

[PGK97] A. Pellenkoft, C. Galindo-Legaria, M. Kersten: The Complexity of Transformation-Based Join Enumeration, In: *Proc. VLDB Conf.*, Athens, 1997.

[Sch97] D. Schneider: The Ins and Outs of Data Warehousing, In: *Tutorial on the VLDB Conference*, Athens, 1997.

[SD90] D. Schneider, D. DeWitt: Tradeoffs in Processing Complex Join Queries via Hashing in Multi-processor Database Machines, In: *Proc. of the Intl. VLDB Conference*, Melbourne, Australia, 1990.

[SM+98] L. Shapiro, D. Maier et al: Group Pruning in the Columbia Query Optimizer, *http://www.cs.pdx.edu/~len.*

[STY93] E. Shekita, K. L. Tan, H. Young: Multi-Join Optimization for Symmetric Multiprocessors, In: *Proc. VLDB Conf.*, Dublin, 1993.

[SM96] M. Stonebraker, D. Moore: ORDBMS - The next Great Wave, *Morgan Kaufman Publishers*, 1996.

[TD93] J. Thomas, S. Dessloch: A Plan-Operator Concept for Client-Based Knowledge Processing, *Proc. 19th VLDB Conference*, Dublin, 1993.

[TPC95] Transaction Processing Performance Council. TPC Benchmark D, Stand. Spec,, Rev. 1.0, 1995.

[WFA95] A. Wilschut, J. Flokstra, P. Apers: Parallel Evaluation of Multi-Join Queries, In: *Proc. ACM SIGMOD Conf.*, 1995.

[Zou97] C. Zou: XPS: A High Performance Parallel Database Server, In: *DE Bulletin* 20(2), 1997.

[ZZS93] M. Ziane, M. Zait, B. Salamet: Parallel Query Processing with ZigZag Trees, In: *Very Large Databases Journal*, 2(3), March 1993.