A Region Based Query Optimizer Through Cascades Query Optimizer Framework

Fatma Ozcan Sena Nural Pinar Koksal Mehmet Altinel Asuman Dogac

Software Research and Development Center of TUBITAK
Dept. of Computer Engineering
Middle East Technical University
06531, Ankara Turkiye
email: asuman@srdc.metu.edu.tr

Abstract

The Cascades Query Optimizer Framework is a tool to help the database implementor (DBI) in constructing a query optimizer for a DBMS. It is data model independent and allows to code a query optimizer by providing the implementations of the subclasses of predefined interface classes. When the implementations of the required classes are provided properly, the generated optimizer produces the optimum execution plans for the queries. Although providing the complete set of rules and thus finding the optimum execution plans are beneficial for most of the queries, the query optimization time increases unacceptably for certain types of queries, e.g., for star queries. Hence it is important to be able to limit the number of alternative plans considered by the optimizer for specific types of queries by using the proper heuristics for each type. This leads to the concept of region based query optimization, where different types of queries are optimized by using different search strategies in each region.

This paper describes our experiences in developing a region based query optimizer through Cascades. Cascades' guidance structures provide the facilities required for the design and implementation of a region based optimizer. The performance comparisons between a region based query optimizer and an exhaustive (which uses the complete rule set without heuristic guidance) query optimizer, both generated through Cascades, indicate that the region based optimizer has a superior performance. In the performance analysis, both the sum of optimization and execution times, namely the response time, and the quality of the plans generated are investigated.

1 Introduction

The Cascades Query Optimizer Framework [Gra 94], which is being used in Microsoft's forthcoming SQL Server and Access query optimizers as well as Tandem's NonStop SQL product, is a tool to help the database implementor (DBI) in constructing a query optimizer for a DBMS. It is data model independent and allows to code a query optimizer by providing the implementations of the subclasses of the predefined interface classes. The interface classes include classes like Operator, Property and Rule. When the implementations of the required classes are provided properly, the optimizer generated through Cascades produces the optimum execution plan. It is an extensible system, that is, adding or deleting rules or operators can easily be accomplished. Furthermore, Cascades provides facilities for incorporating heuristic guidance to the optimizer. These features of Cascades make it a very attractive tool for developing query optimizers.

One of the earliest efforts in developing a tool for query optimization with minimal assumption on the data model is given in [Frey 87] where a rule-based description of generating equivalent query execution plans from an initial query specification is proposed. The EXODUS project of [GD 87] focuses on how to include rules into an architecture of an optimizer generator. The concepts used in EXODUS optimizer generator are, data model description as input file, rules to specify alternatives, compilation of rules into source code and separation of logical and physical operators. The drawback of the EXODUS query optimizer is the inefficiency and the ineffectiveness of its search strategy [Gra 94]. Another effort in this respect is the Volcano Query Optimizer Generator [McK 93, GM 93]. Volcano provides an efficient search engine based on dynamic programming and memoization. However, the Volcano technique generates all equivalent logical expressions in the first phase.

Even if the actual optimization phase uses a greedy search algorithm, this first phase in Volcano must still be exhaustive. In Cascades, this represents the worst case that happens when there is no heuristic guidance.

The work presented in this paper has evolved through our experiences in developing a query optimizer for METU Object-Oriented DBMS (MOOD) [DAOD 95, Dog 95]. MOOD query optimizer [Dur 94] is developed through Volcano by using the full set of rules. Experiments with MOOD optimizer revealed the fact that the optimizer took unacceptably long for certain queries. We have identified basicly two types of queries with unacceptable response times; first type is the star queries and the second type of queries involves many (more than 5) selection predicates in the where clause. The poor performance of the optimizer for star queries is obvious; the number of alternative plans that the optimizer considers is exponential in the number of relations or classes involved. The poor performance of the queries with many selection predicates stems from the fact that there are many rules for ordering the select predicates. We have concluded that for these types of queries, instead of searching the space of alternative plans exhaustively, heuristics must be introduced. Using different sets of rules in conjunction with different heuristics for different types of queries led us to the region based optimization. We have identified three types of queries to be optimized with three different strategies and implemented each strategy in a region without any interaction among the regions. Note that this is an initial step towards the region based optimization described in [Mit 93]. We then developed such a region based optimizer through Volcano [Kok 95]. However, since it is not possible to add new control strategies to the Volcano search engine, we had to introduce outside control over the Volcano, which in turn reduced the effectiveness of the approach.

In this paper, we describe our experiences in developing the region based query optimizer through Cascades. The rule set used [BMG 93] is provided in the Appendix. These rules are sufficient to optimize both relational and object-oriented queries. The performance comparisons between a region based query optimizer and an exhaustive query optimizer, both generated through Cascades, indicate that the region based optimizer has a superior performance.

The paper is organized as follows: Section 2 contains a brief summary of the previous literature that is directly related to our work. A short summary of Cascades Query Optimizer Framework is given in Section 3. In Section 4, generating a region based optimizer through Cascades is described. Section 5 presents the performance comparisons between a region based query optimizer and an exhaustive query optimizer, both generated through Cascades. Finally, Section 6 contains the conclusions.

2 Related Work

We begin by noting that some instances of the query optimization problem are NP-complete [IK 84]. In [OL 90], it is shown that the complexity of optimizing the order of join operations is dependent upon the shape of the query. The shape of the query indicates how tables are connected with predicates. In linear queries, tables are connected by binary predicates in a straight line; whereas in star queries, a table at the center is connected by binary predicates to each of the other surrounding tables.

The computational complexity (i.e. the number of joins that must be considered when using dynamic programming for optimization) of linear queries with composite inners (bushy trees) is $(N^3\text{-}N)/6$. If the composite inners are not considered the complexity reduces to $(N\text{-}1)^2$. On the other hand, using dynamic programming to optimize a star query with N quantifiers requires evaluating $(N\text{-}1)2^{N-2}$ feasible joins [OL 90]. This study indicates that it is not feasible to consider all possible alternative plans for star queries. Yet, certain other types of queries might benefit a lot from considering all possible plans. The space of alternative plans must therefore be adjustable for each type of query. This leads to the concept of region based query optimization.

In [Mit 93, MZD 92, MDZ 93, MDZ 94] an architecture for region based extensible processing and optimization of queries is proposed. An Epoq optimizer is a collection of concurrently available region modules, each of which embodies one strategy for the optimization of query expressions. Different regions often accomplish different query transformation tasks, but regions may also represent different strategies for accomplishing the same task in different ways. The Epoq architecture integrates the regions through a common interface for the region modules, and a global control that combines the actions of subordinate regions to process a given query. The region modules are organized hierarchically, with a parent region controlling its subordinate regions as though they were a collection of transformations. The regions define, through their interface, the characteristics of queries they can process, goals for the transformation of queries, and the characteristics of result queries. A higher level control uses this information to plan a sequence of region executions to process a given query expression.

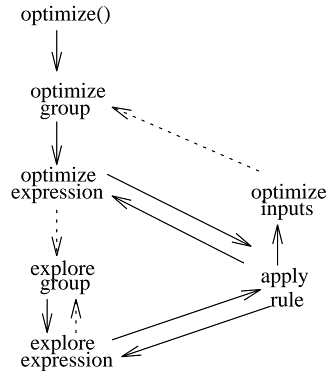

Figure 1: Cascades task structure

In [Loh 88, HFLP 89], in the framework of the extensible query processing in Starburst, productions of a grammar are used to define query execution plan alternatives. The terminals of this grammar are base-level database operations on tables and the nonterminals are defined declaratively by production rules which combine those operations into execution plans. Each of these productions produce a set of alternative plans, each having a vector of properties, including the estimated cost. In addition, productions can require certain properties of their inputs, such as sortedness.

3 Cascades Query Optimizer Framework

In Cascades [Gra 94], the optimization algorithm is broken into several parts, which are called tasks. A task is realized as an object and a task object exists for each task that has yet to be done. All the tasks are collected in a task structure, a last-in-first-out stack. In Figure 1, tasks that make up the optimizer's search engine are shown. The "optimize" procedure first copies the original query into an internal structure, namely the "memo" structure, which holds all the equivalent logical and physical expressions. Then "optimize" triggers the entire optimization process with a task, namely "opt_group", to optimize the class corresponding to the root of the original query tree, which in turn triggers optimization of smaller subtrees.

A task to optimize a group, which is a collection of equivalent expressions, or a single expression combines a group or an expression with a cost limit and with required and excluded physical properties. Performing such a task results in either finding the best plan, or failure.

Exploring a group or an expression is a new concept that has no equivalent in Volcano. In Volcano search strategy, all rules are applied in the first phase to create all possible logical expressions of a query and its subtrees. In the second phase, implementation rules are applied to these logical expressions to obtain plans and the best plan is chosen using branch and bound pruning. In Cascades, this separation into two phases is abolished, since it is not useful to derive all logically equivalent expressions. A group is explored by applying rules only on demand.

While the EXODUS and Volcano optimizer generators had the concept of support functions, Cascades is completely based on C++ subclasses. Using the object-oriented paradigm, Cascades provides a clear interface between the optimizer and the DBI supplied functions. Each of the classes that make up the interface between the optimizer and DBI is designed to be the root of a subclass hierarchy. The DBI creates a new optimizer by providing the implementations of these subclasses.

In Cascades, an operator can be both logical and physical. For each operator, one method indicates whether an operator is a logical operator, while a second method determines if an operator is a physical operator. The definition of operators includes their arguments, thus, no separate mechanism is provided for "argument transfer". All operators must provide a method which determines how many of the given set of rules will be applied.

Methods for matching, finding and improving logical properties must be provided by the DBI for operators that are declared to be logical. Similarly, for physical operators, methods for finding an operator's output properties and for computing and inspecting an operator's cost must be supplied by the DBI. Another method that maps an expression's cost limit to a cost limit for one of its inputs is also required.

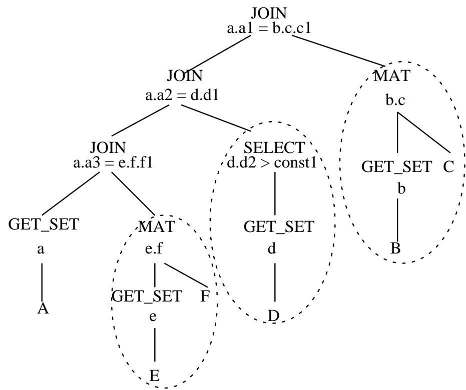

Figure 2: Example star query

Another important class of objects in Cascades Optimizer Framework is the class RULE. Cascades does not distinguish between (logical) transformation and (physical) implementation rules. All rules have a name, an antecedent (the pattern) and a consequent (the substitute) both of which can be arbitrarily complex. Two types of condition functions are supported which consider the rule and the current optimization goal, i.e., the cost limit and the required and excluded properties. In addition to cost limits, required and excluded properties, rule application can also be controlled by heuristics. For this purpose, before optimization starts, "promise" functions inform the optimizer how useful the rule might be. When there is no guidance, all promise functions should return 1 to indicate that the rule will be pursued. A value 0 or less will prevent the application of the rule. All promise and condition functions must be supplied by the DBI. Guidance information is passed to the functions by the help of the guidance class. The guidance class captures knowledge about the search process and the heuristics for future search activities and it is handled by the DBI.

4 A Region Based Query Optimizer Generated Through Cascades

As described in Section 1, we have identified three query optimization regions. In order to classify the queries into these regions, the following criterion is used: A query that has two or more join operators which have the same bind variable as one of their operands is classified as a star query. If a query is not in this region and has five or more select operators then it is classified as a select query. Otherwise, it is classified as a default query and optimized with the complete rule set. It should be noted that, currently a query falls only in one region and optimized in that region without any interaction with the other regions.

In the following, the heuristics introduced for star and select regions along with their implementation in Cascades are presented.

4.1 Star Region

It has been observed that with exhaustive search strategy Cascades Optimizer Framework spends too much time to optimize star queries, and most of this time is spent in ordering the join classes that cause the query to be star shaped. The ordering of join classes is provided by commutativity and associativity rules of join classes (Rules 8,9). If these two rules are disabled, optimization time can be saved. However, the plans generated by disabling these rules result in large deviations from the optimum plan therefore heuristics must be used. The heuristic that we have introduced is as follows: we assume the linear subpart of a star query as a unit of processing and we process a star query by executing the linear subqueries in the order from the least costly to the most costly. The intuition behind this heuristic is the following: executing a linear subquery will reduce the size of a partial result and this will be achieved in the least expensive way by executing the least expensive subqueries first. Yet, not only the size of a partial result of a join, but join selectivity also affects the order of joins. We have developed another heuristic to consider join selectivity and join cost together which is not described in this paper due to

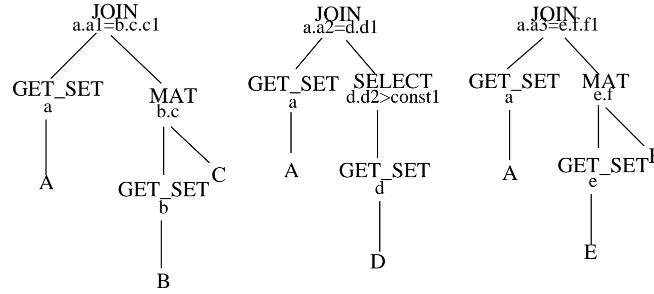

Figure 3: Subqueries of example star query

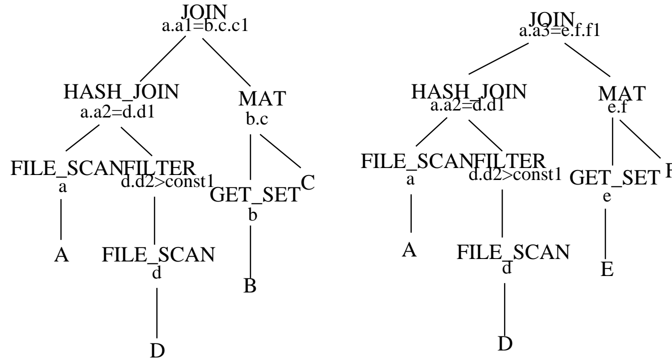

Figure 4: Subqueries after the first pass

space limitations, [ODE 95].

In star region, each linear subquery of a given query, (shown with dotted lines in Figure 2) is thought as a unit of processing and optimization of this linear part with associativity and commutativity of join rules enabled (Rules 8,9), takes polynomial time. We need to optimize these linear subqueries (Figure 3) one at a time according to our heuristic. However, calling the "optimize" routine once for each of these subqueries is not efficient since a new "memo" structure is created for each subquery, making it impossible to share common subqueries and to use previously generated physical plans. Therefore, the search engine of Cascades is extended to allow the subqueries to be optimized one at a time without losing the previously generated physical plans. In order to achieve this, the task structure of Cascades is exploited. In Cascades, the task "opt_group" takes a group to optimize, combines the group with a cost limit, required and excluded physical properties and returns either a plan or failure. In its original form, optimization begins by triggering an "opt_group" task for the class corresponding to the root of the input query. We have modified this concept and for star queries we triggered an "opt_group" task for each subquery. After the optimization of subqueries is completed, the subquery with the minimum estimated cost is chosen and joined with the other subqueries, as shown in Figure 4. Then in the next pass, these newly created subqueries are optimized. While optimizing a group corresponding to one of these newly created subqueries, previously generated physical plans are used, thus, no rule application take place in these parts, since Cascades explores a group only on demand, i.e. when there is no plan satisfying the current optimization goal. This process continues until no subquery is left. Thus, there are as many passes as the initial number of subqueries.

The modified form of the "optimize" routine of Cascades is given in Algorithm 1.

Algorithm 1: Modified "optimize" routine of Cascades

```
optimize(qry,guidance){
    if guidance.region is star_region{
        create_subqueries(qry) //creates subqueries in the "memo" structure
        set pass_count to number of subqueries
    }
    for i= 1 to pass_count do{
        if guidance.region is star_region{
            for j=1 to current number of subqueries
                push "opt_group" task for jth subquery to task_list
        }
        else //not star region
            push "opt_group" task for the root of the query
        while task_list is not empty
            perform task
        if guidance.region is star_region{
            find the subquery with the minimum cost
            modify_memo() //update "memo" structure such that the subquery
                            with the minimum cost is joined with other subqueries
        }
    }
    return plan
}
```

4.2 Select Region

In the complete rule set of the optimizer given in the Appendix [BMG 93], there are many rules for select classes (Rules 1,2,3,4,5,6,7,10,15,16). It has been observed that these transformations increase the optimization time a lot for queries with many select predicates. We have further noted that some of these rules can be disabled by using effective heuristics instead.

In the following, we will describe these heuristics through examples. In order to be able to apply these heuristics, certain modifications are required in the input query tree. The first modification to the query tree is placing the select operators at the top of the tree. This eliminates the need for the rules (Rules 4,10) that move the selections up in the tree. Second modification requires the decomposition of select predicates beforehand so that there is no need for the rule (Rule 1) that decomposes them. Another rule that can be disabled is the rule (Rule 3) for the commutativity of select operators. Yet, when all these rules are disabled, there will be large deviations from the optimal plan. As an example consider the query tree of Figure 5.a and assume the corresponding optimum execution tree to be as given in Figure 5.b. When all these rules are disabled, the optimizer can not generate the tree of Figure 5.b because SELECT_1 can not be moved beneath SELECT_2. This problem can be eliminated by introducing the following heuristics: First sort the select operators for the same range variable according to their number of path expression elements. The intuition behind is, the path with fewer path expression elements can be moved beneath in the query tree more deeply during the optimization. Next, the select operators are sorted according to their selectivities by preserving the relative order imposed by the first sorting. The intuition behind this heuristic is to let more selective predicate go down the query tree. With these sortings, it is possible for Cascades to generate the optimum execution plan for the example given in Figure 5 even when the rules mentioned above are disabled.

After the required modifications are performed on the query tree (i.e. moving the select predicates and decomposing), the query is optimized in the select region. For the select region, two pass optimization is performed by calling the "optimize" routine twice. In the first pass, all select rules (Rules 1,2,3,4,5,6,7,10) are disabled and the join and materialize operators are ordered. At the end of this pass, the "optimize" routine of the Cascades returns a physical plan. This physical plan is converted to a logical query which the "optimize" routine takes as input, and optimized again in the second pass. Although it is possible to make two pass optimization inside

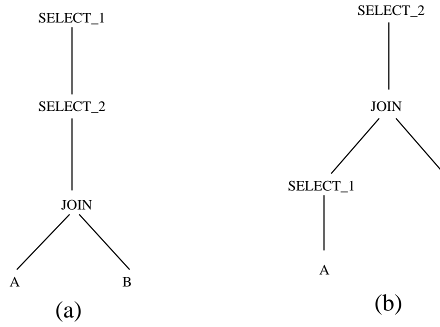

Figure 5: Select commutativity example

the "optimize" routine, it is not very efficient since in the first pass, Cascades creates a lot of equivalent logical and physical expressions, which are not required in the next pass, and if these are not deleted, a large number of rules will match to these large number of expressions. Therefore, the "optimize" routine is called twice making it possible to use a new "memo" structure in the second pass in which the select operators are ordered by disabling join and mat rules (Rules 8,9,11,12). The overall approach is presented in Algorithm 2.

Algorithm 2: Region based optimizer

```
Region_based_optimizer ( QUERY qry, PLAN plan){
    create_operator_table(qry,op_table) //creates a table to count select and join classes
    decide_qry_type(op_table) // decides the query type
    if query is star_query
       set guidance.region to star_region
    else if query is select_query
       set guidance.region to select_region
    else
       set guidance.region to default_region
    if guidance.region is select_region{
       order_query(qry) //reorders the query such that select operators are at the top of the query
       set guidance.count to 0 //to order joins, all select rules are disabled
       order_selects (qry) // sort selects
       plan = optimize (qry,guidance) // a plan is found
       convert_qry (plan,qry) //converts the coming physical plan into an equivalent logical query
       set guidance.count to 1 //enables select rules while disabling join and mat rules
    }
    plan = optimize (qry,guidance)
}
```

5 Performance Evaluation

This section presents the performance comparison of the region based optimizer and an exhaustive optimizer, both generated through Cascades. Their optimization and response times and the quality of the plans generated are compared.

In the experiments, a workstation running Microsoft NT with 64 MB main memory and 250 MB swap space is used. The queries are generated by a random query generator, and both optimizers are run with the same set of queries. The optimization time of these queries are obtained from the system and the execution times are estimated using the same cost functions for both optimizers. The results are presented in the following sections.

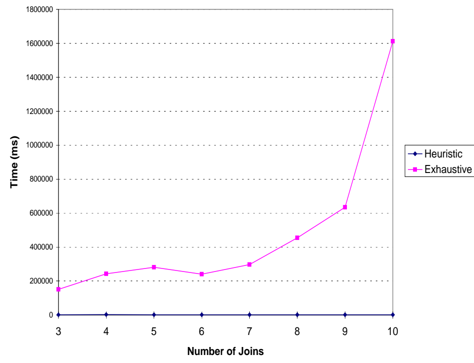

Figure 6: Optimization times for the star region

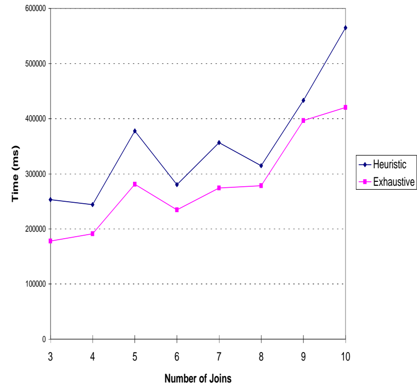

Figure 7: Estimated execution times for the star region

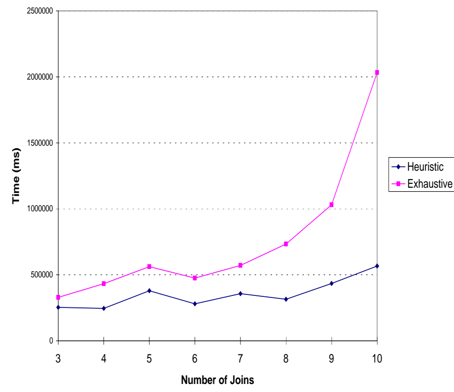

Figure 8: Response times for the star region

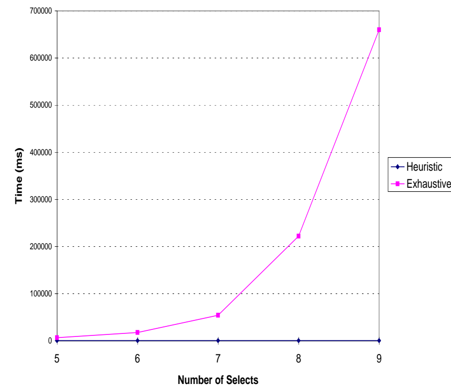

Figure 9: Optimization times for the select region

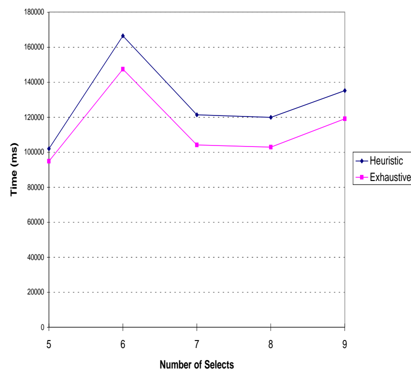

Figure 10:Estimated execution times for the select region

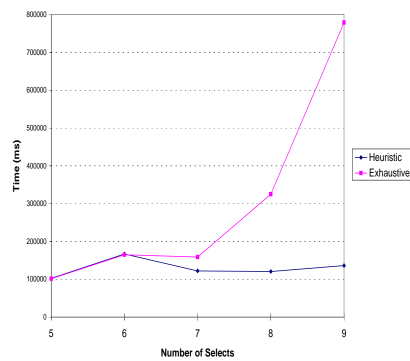

Figure 11: Response times for the select region

5.1 Star Region

For the experiments in the star region, 80 star queries are generated randomly. Number of join operators in this query set, range from 3 to 10, and for each query type 10 queries are generated, and the average of these 10 values are used.

Figure 6, Figure 7 and Figure 8 depict the results of the experiments. As shown in Figure 6, optimization time difference between two optimizers increases dramatically as the number of join operators increases. Although the plans generated using the region based optimizer are slightly worse than the optimal (Figure 7), the total response time is better in the region based optimizer (Figure 8). Since the aim of query optimization is to minimize the response time, these results justify the heuristic used in the star region of the region based optimizer.

5.2 Select Region

For the experiments on queries with many select operators, 50 select queries are generated randomly with the number of select predicates ranging from 5 to 9 and with 2 join and 3 materialize operators. Comparison of the optimization time, quality of the plans generated, and response times are shown in Figure 9, Figure 10 and Figure 11 respectively. The region based optimizer spends less time to optimize queries than the exhaustive optimizer (Figure 9). The quality of the plans generated using the region based query optimizer is slightly worse than that of the optimal ones as shown in Figure 10. This is an expected result, since when some rules are disabled, the generated plans deviate from the optimal. However, the total response times of queries are better for the region based optimizer (Figure 11). These results also indicate that the heuristic suggested is an effective one.

6 Conclusions

A region based optimizer that contains three optimization regions each of which optimizes different queries with different optimization strategies is generated through Cascades Query Optimizer Framework. Heuristic guidance feature of Cascades proved to be very useful in this implementation.

The performance of the region based optimizer is compared with an optimizer that uses the complete rule set, both generated through Cascades. The results of the experiments show that although with the region based optimizer the quality of the plans are not as good as the plans generated by the exhaustive optimizer, optimization time gain is uncomparably larger. For this reason, region based optimizer's total time for queries, that is, the query response times are better.

Appendix: The Complete Rule Set

1. Select (op,pred) -> Select (Select (op,pred1) , pred2)
2. Select (Select(op,pred1),pred2) -> Select (op,pred)
3. Select (Select(op,pred1),pred2) -> Select (Select (op,pred2),pred1)
4. Mat (Select (op,pred)) -> Select (Mat (op),pred)
5. Select (Mat (op), pred2) -> Mat (Select (op,pred))
6. Select (Join (op1,op2,pred1),pred2) -> Join (Select (op1,pred2),op2,pred1)
7. Select (Join (op1,op2,pred1),pred2) -> Join (op1,Select(op2,pred2),pred1)
8. Join (op1,op2,pred) -> Join (op2,op1,pred)
9. Join (Join (op1,op2,pred1),op3,pred2) -> Join (op1,Join (op2,op3,pred3),pred4)
10. Join (Select (op1,pred1) ,op2,pred2) -> Select (Join (op1,op2,pred2),pred1)
11. Mat1 (Mat2 (op)) -> Mat2 (Mat1 (op))
12. Mat (op) -> Join (Get_set,op1,pred)
13. Get_set -> File_scan
14. Mat (op) -> Traverse (op)
15. Select (op,pred) -> Filter (op,pred)
16. Select (op,pred) -> B_tree_select (op,pred)
17. Join (op1,op2,pred) -> Merge_join (op1,op2,pred)
18. Join (op1,op2,pred) -> Hash_join (op1,op2,pred)
19. Join (op1,op2,pred) -> Nested_loop_join (op1,op2,pred)

20. Join (op,Get_set,pred) –> Ptr_hh_join (op1,pred1)

References

[BMG 93] Blakeley, J. A., McKenna, W. J., Graefe, G., "Experiences Building the Open OODB Query Optimizer", Proc. of the ACM SIGMOD Conf., 1993.

[DAOD 95] Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., "Implementation Aspects of an Object-Oriented DBMS", in ACM SIGMOD Record, Vol.24, No.1, March 1995.

[Dog 95] Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., Altintas, I., "METU Object-Oriented DBMS Kernel", 6th International Conference on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer Verlag 1995).

[Dur 94] Durusoy, I., "MOOD Query Optimizer", M.Sc. Thesis, Dept. of Computer Eng., Middle East Technical University, Ankara, Turkey, 1994.

[Frey 87] Freytag, J.C., "A Rule-based View of Query Optimization", in Pr oc. of ACM SIGMOD Conf., 1987.

[GD 87] Greafe, G., DeWitt, D. J., "The EXODUS Optimizer Generator", in Proc. of ACM SIGMOD Conf., 1987.

[GM 93] Greafe, G., McKenna, J. W., "The Volcano Optimizer Generator:Extensibility and Efficient Search", Proc. IEEE Conf. on Data Eng., Vienna Austria,1993.

[Gra 94] Graefe, G.,"Query Optimization in the Cascades Project", unpublished manuscript, 1994.

[HFLP 89] Haas, L.M.,Freytag, J.C.,Lohman, G.M.,Pinaresh, H., "Extensible Query Processing in Starburst", in proceedings of ACM SIGMOD Conf., 1989

[IK 84] Iberaki, T., and Kameda, T., "On the Optimal Nesting Order for Computing N-Relational Joins", ACM Transitions on Database Systems, Vol. 9, No. 3, 1984.

[Kok 95] Koksal, P., "Design and Implementation of a Region Based Query Optimizer for Object-Oriented DBMSs", M.Sc. Thesis, Dept. of Computer Eng., Middle East Technical University, Ankara, Turkey, 1995.

[Loh 88] Lohman, G.M., "Grammar-like Functional Rules for Representing Query Optimization Alternatives", in proceedings of ACM SIGMOD Conf., 1988

[McK 93] McKenna, W.J., "Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator", Ph. D. Thesis, Univ. of Colorado, 1993.

[MDZ 93] Mitchell, G., Dayal, U., and Zdonik, B.S., "Control of an Extensible Query Optimizer: A Planning-Based Approach", Proc. of Intl. Conf. on Very Large Databases, 1993.

[MDZ 94] Mitchell, G., Dayal, U., and Zdonik, B.S., "Optimization of Object-Oriented Queries: Problems and Approaches", in Advances in Object-Oriented Database Systems, Springer Verlag, 1994.

[Mit 93] Mitchell, G., "Extensible Query Processing in an Object-Oriented Database" PhD thesis, Brown University, 1993.

[MZD 92] Mitchell, G., Zdonik, S., and Dayal, U., "An Architecture for Query Processing in Persistent Object Stores", Proc. of the Hawaii Intl. Conf. on System Sciences, 1992.

[ODE 95] Ozkan, C., Dogac, A., Evrendilek, C., "A Heuristic Approach for Optimization of Path Expressions in Object-Oriented Query Languages", 6th International Conference on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer Verlag 1995).

[OL 90] Ono, K., Lohman, G. M., "Measuring the Complexity of Join Enumeration in Query Optimization", Proc. of Intl. Conf. on Very Large Databases, 1990.