Control of an Extensible Query Optimizer: A Planning-Based Approach

Gail Mitchell[†] Umeshwar Dayal[‡] Stanley B. Zdonik[§]

Abstract

In this paper we address the problem of controlling the execution of a query optimizer. We describe a control for the optimization process that is based on planning. The controller described here is a goal-directed planner that intermingles planning with the execution of query transformations, and uses execution results to direct further planning of optimizer processing.

We describe this control in the context of the Epoq extensible architecture. Epoq is an approach to extensible query optimization that integrates specialized rewrite strategies through its extensible control mechanism. This paper describes our planning-based approach to extensible control and illustrates it with a simple example.

1 Introduction

Optimization of a query is inherently a process of searching the space of expressions equivalent to the query. Typically, a given optimizer can only visit some portion of this space, since the set of transformation rules is usually incomplete, the cost of optimization must be bounded, and the optimizer control strategy limits the search. The control strategy of an optimizer determines, for any query, the equivalent queries that will be searched as well as the order in which they are considered.

The extensible nature of object-oriented systems requires that an optimizer's search be expanded in response to the new kinds of expressions that can be written. The approach taken by many extensible optimizers is to add new rules for transforming queries. The kinds of rules used to describe query transformations, and the control over execution of those rules, differ in all systems [5, 7, 16, 19].

Some systems also recognize a need to support new strategies for optimization; i.e., extensibility of the optimization process itself [11, 14, 18]. The control we present here is designed to support this kind of extensibility.

The Epoq approach to extensible query optimization allows extension of the collection of control strategies that can be used when optimizing a query [14]. Each strategy can search some portion of the space of queries equivalent to the optimizer input query. Different strategies will usually search different (possibly overlapping) parts of the search space, although different strategies may simply offer alternative ways to search the same space.

An Epoq optimizer is a collection of concurrently available *region* modules, each of which embodies one strategy for the optimization of query expressions. 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. This is illustrated in Figure 1.

The root module of the optimizer communicates with the query processing system. It receives a query to optimize, and produces an optimized result. This result is computed with the assistance of its child regions. Child modules transform queries at the request of a parent region, and may also act as parents by using subordinate regions to assist with this transformation.

Structuring the region modules hierarchically puts knowledge about regions that can cooperate to process a single query in one place—i.e., a parent. The parent's strategy, and the characteristics of the query expression being optimized, determine how the subordinate regions will cooperate. A parent region composes the transformations of its subordinates to produce an equivalent result query.

We distinguish two kinds of regions in an Epoq optimizer–interior regions (including the root) and leaf regions. Both kinds of regions are transformations, but they differ in that the control of interior regions has to manipulate transformations represented by other regions in the optimizer, whereas

[†]GTE Labs, Waltham, MA 02254 USA. gmitchell@gte.com

[‡]Hewlett-Packard Labs, Palo Alto, CA 94304-1120 USA. dayal@hplabs.hp.com

[§]Computer Science Department, Brown University, Providence, RI 02912 USA. sbz@cs.brown.edu

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

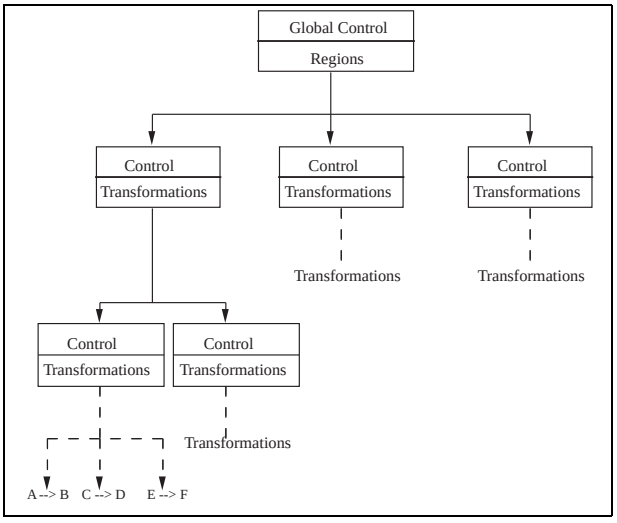

Figure 1: Epoq architecture

the control and transformations of a leaf region are internal to the region and are not explicitly addressed in the architecture. Indeed, in Figure 1 the leaf regions are shown simply as transformations. Of course, these transformations will, in practice, be complicated strategies for manipulating queries. The control presented here concentrates on the integration of transformation strategies.

Most extensible optimizers (e.g., [5], [7], [8], [10], [20]) provide a fixed control over the application of strategies to manipulate queries. This makes it difficult for these optimizers to adapt to a changing repertoire of strategies. Epoq allows extension of the control as well as the addition of new optimization strategies.

Epoq was motivated by a need to address extensibility in the design of object-oriented query optimizers, but we believe that it has more general utility. The architecture and control of Epoq increase the range over which any optimizer might be extended.

The major contributions of this paper are

- a definition of the control problem for extensible query optimizers
- an extensible, planning-based approach to solving the optimizer control problem.

In the next section we present the control problem. We approach this problem in the context of the Epoq architecture for query optimizers, described further in Section 3. In Section 4 we present a simple example optimizer that instantiates this architecture and can be used to illustrate the discussion in later sections. In Section 5 we present an architecture for optimizer control and a design for a planning system for controlling the activities of an optimizer. In Section 6 we compare our work to other systems, and we summarize results in Section 7.

2 The Control Problem

The strategy for achieving some goal is encapsulated inside a region. This strategy is implemented through the region's control over the application of query transformations. In an Epoq optimizer, subordinate regions act as query transformations, and a parent region controls the application of those transformations through requests to subordinates to transform a query expression.

The major decisions that need to be made by a region are 1) which query or subquery to process and 2) which region to execute. A region receives a single query to transform, as well as a goal for the transformation, and needs to decide what transformations to apply to the query, or any subqueries, in order to achieve its goal. Throughout the transformation process a region will usually maintain a number of alternative query expressions, and will work on different of those expressions at different points in its processing.

One way to approach this process is to pair query expressions with applicable regions, then select an expression/region pair to execute.[1] Such a control is like a search through transformation rules, where the rules are the child regions. One difficulty in doing this is in determining when a region (rule) is the right one to apply to a query. Normally, rule-based optimizers do pattern matching (and usually condition testing) of the query expression with the left hand sides of rules, then perform some conflict resolution if more than one rule matches a query (e.g., assign weights to rules as in [7]). This approach is not satisfactory for an Epoq optimizer because the regions do not behave as precisely as rules behave.

In a rule-based optimizer, rules provide complete information to a search engine, and can be applied by a rule execution process. In an Epoq optimizer, a region, behaving as a rule, provides incomplete information to its parent, and applies itself. The result of a region execution is returned to a parent, but the actual execution of the region is done independently of the parent processing. As a result, a region may not achieve its goal and may return a message to its parent indicating such a failure.

An alternative to rule search is to view region executions as actions in an optimization planning system. The decisions that need to be made (which region to execute, which query expression to manipulate) are managed by a planning system that is driven by its own planning rules. The planning rules are heuristics about orderings of region applications that will (hopefully) achieve the region's goals. Thus, the planning system is planning the execution of the optimizer. Since, in Epoq, a region may fail to transform a query (i.e., fail to achieve the region's goal), the planning process cannot proceed independently of the region results and is thus interleaved with region execution. This is discussed in more detail in Section 5.

The *goals* of a region characterize the output queries that can be produced by a region. Given a particular query,

[1] An *applicable* region is one that expects to be able to achieve its goal on the query. Applicability is assessed through functions a region provides to its parent control. Such measures test necessary conditions on a query for a region to transform it.

success indicates whether the region was able to attain a particular goal for that query. For example, if a region's goal is to lower cost, and the result query computed by the region has a lower cost than the input query, then the region was successful at achieving its goal.

Termination refers to stopping the execution of a region. In general a region will have its own internal criteria for termination, since termination is an integral part the region's control. Termination may be related to success—a region may terminate processing a query when it discovers it is successful at achieving its goal. However, termination will often be conditional on more than success; for example, a region with a goal to lower cost will usually not quit as soon as the cost is lower, but will continue to try to improve the cost until it decides that further work will not be cost effective. In all cases, termination must involve conditions that are independent of success. A region will not necessarily achieve success, so termination conditions must ensure that the region stops regardless of success at achieving a goal.

3 Region Architecture

All regions have control over the transformation of queries. This conceptual view of a region describes the fact that a region implements a control strategy for manipulating queries. The transformations of a region are those described by the control strategy, and the region control implements this strategy. For interior regions, the transformations used by the control are subordinate regions, as well as, possibly, internally defined transformations. For leaf regions, all transformations are internally defined.[2]

A region interface provides the support for communication between the control of a region and its parent or child regions. The interface to its parent allows a region to be used as a transformation by the parent. The interface to a region's children allows the region to use the children as transformations. This interface can request information from children which is then supplied to the region's control. Region interfaces are described more completely in [15].

Epoq defines a common structure for the interface to ensure structural compatibility. This supports communication between regions as well as the addition of new regions to an optimizer. The architecture of all parent-child interfaces is the same, although the implementation of this architecture may be different for each interface. For example, the architecture specifies that a query passes between a parent and a child region. Although we would expect all interfaces to use the same query representation, it is possible for a particular parent-child interface to use a different query representation. The root region, for example, might accept and return a query expression as a string if it can translate that form back and forth between the representation to be used in the optimizer.

[2] Transformations may be explicitly defined in leaf regions; for example, a rule-based optimizer contains an explicit set of rules and a control mechanism that implements some sort of rule search. The transformations will often be implicit, i.e., built into the control mechanism. For example, a region that uses dynamic programming to generate efficient join orderings uses, implicitly, commutativity and associativity transformations on the join operators.

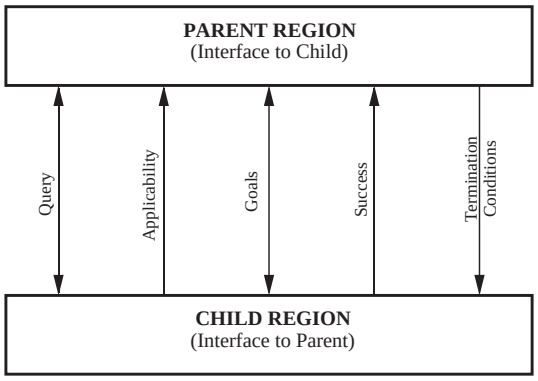

Figure 2: Interface Between a Parent and Child Region.

The diagram of Figure 2 indicates the kinds of information passed between a parent and child region. The arrows in the figure indicate the direction of information flow. For example, queries are passed both ways: a query to be transformed is passed from a parent region to a child, and a transformed query is passed from a child region to its parent.

We say a goal is 'recognized' by a parent region when the parent region contains the control mechanisms to request that a child region work towards that goal. A goal is 'attainable' by a region when the region has the control mechanisms to work towards that particular goal. It is important to ensure that the attainable goals of a subordinate intersect the goals recognized by a parent, since the subordinate is useless if it cannot achieve any of its parent's goals.

Applicability refers to the ability of a region to transform a query. Thus, applicability is directly related to the control strategy of a region; a region for which some transformation applies to a query q is said to be *applicable to* q. Of course, assessing applicability of a region to a query by finding a transformation sequence is not feasible, so a region needs to use other measures to assess its applicability. An applicability measure for a region should indicate necessary conditions for the region to transform a query.

Practical measures of applicability may consider goals, the input query, termination conditions, transformations, region control, etc. Applicability measures provide a means for a parent region to eliminate from the decision-making process regions that will not be able to process the current query. Thus, they are used to improve the efficiency of the optimizer.

Applicability in an Epoq optimizer is similar in function to pattern-matching and condition-matching of left-hand sides in more traditional rule-based optimizers. Applicability differs in that it doesn't guarantee that a region can process a query, but only tells when a region can probably process a query. This difference is accounted for in the region's control. The region requests a subordinate to achieve a particular goal on a query, and must be able to accommodate failure of the subordinate to achieve that goal.

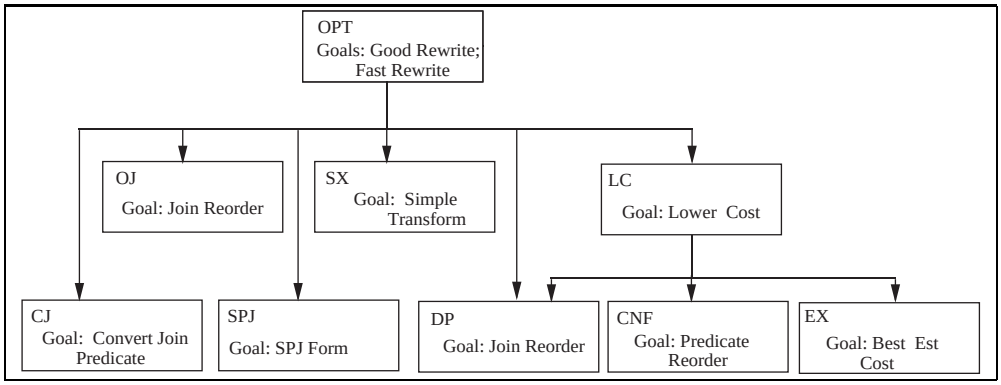

Figure 3: Example optimizer design

4 Running Example

In this section we give a design for a simple query rewrite system to illustrate the capabilities of the Epoq architecture and, in particular, to illustrate the planning-based control that will be presented in Section 5. The purpose of this example is not to define new optimization heuristics or propose new optimization strategies. Indeed, designing good optimization strategies, and heuristics about interactions between different optimization strategies, is an open area of research. The Epoq architecture is a vehicle for defining such interactions and can be a testbed for experimentation with optimizer design heuristics.

The example optimizer we use throughout the rest of this paper consists of nine regions connected as depicted in Figure 3. Each box in the picture is a region and is labelled with a name for the region as well as each region's goal(s).

The root region, named OPT, uses the following regions to do its transformations:

- SX does simple query transformations(X) such as predicate simplification, view substitution, collapsing Project operations, etc.
- CJ converts nested predicates to Join operations, when possible.
- OJ reorders join operations. In particular, this region can handle OuterJoin and Join operations.
- SPJ tries to convert nested queries involving Select, Project and Join operations into a canonical form with all Joins followed by Selects followed by Projects. Such a form could be conducive to lower level query optimizations.
- DP reorders join operations using a dynamic programming algorithm and a simple cost model.
- LC tries to lower the expected cost of the query.

The OPT region takes a query in a high-level algebraic language and applies its subordinate regions to manipulate the query and produce an algebraic query with lower expected cost. OPT is not a complete optimizer but could be used, for example, as a region for a higher-level optimizer with a control that uses other regions to translate a declarative query to the algebra and manipulate the result of OPT to produce a query plan.

OPT can use its subordinates in different ways depending on the nature of the query it is trying to optimize. For example, a simple query with no nested expressions could be simply processed by region LC to lower the expected cost. More complex queries, with nested expressions, can be processed by a region that works to unnest the expressions (e.g., SPJ or CJ) followed by a region that reorders the resulting Join operations (OJ or DP). The specification of such orderings, and the means for choosing among regions with the same goals (e.g., OJ and DP) are presented in Section 5.

Region OPT can also choose to process queries using a "pilot pass" style algorithm [17] that first applies simple transformations to the query (region SX) and is satisfied with the result (and quits) if those transformations reduce the expected cost by some amount. If the preliminary pass is not satisfactory, more complete processing of the query can be undertaken by the lower cost region (LC), for example. Such an algorithm could speed processing of simple queries by the optimizer.

The lower cost region (LC) takes advantage of the hierarchical structure of the Epoq approach. This region has three subordinate regions that it can use in trying to achieve its goal of lowering the expected cost of its input query expression. Region EX could be a rule processor (perhaps built by an optimizer generator [9]) that estimates the cost of transformed queries and returns the lowest cost query it can find. Region LC can choose to use this region to attain its own goal, or can choose to send the query through a sequence of modules (CNF and DP, here) each of which has its own strategy for applying its smaller collection of rules to the query (similarly to [5] or [18]). LC could even try both strategies, choosing the best result, or could use the strategies in a pilot pass type of approach.

The leaf regions in this example are complicated strategies for the application of transformations to queries. These strategies are controlled by higher-level regions. It should be noted that the granularity of leaf strategies could be much

finer; for example, a leaf could be a single transformation rule, with strategies for controlling the rules described in a parent region.

The optimizer in this example ignores many of the problems that are encountered with object-oriented queries. For example, this optimizer design doesn't include any processing for path expressions (as in [12], for example) or any type specific optimization (e.g. [13]). Such optimizations could be included as leaf regions, and the control of OPT could use these strategies in conjunction with the other strategies when appropriate. For example, path expression processing might be combined with join/outerjoin reordering. The Epoq architecture allows such additions to the repertoire of strategies, and the control we present next provides for extensions that can incorporate these additions.

5 Control Architecture

The actual transformation of a query expression is the responsibility of the region control. The region interface provides information to the control, and relays control results and requests to other regions. The control components interact with each other, and with the interface, to effect the transformation process. Through its management of the transformation process, the control implements a control strategy for the region.

A decision-making component is the central part of a region control. This component makes the decisions about what processing the region must do to transform a query. It interacts with a control store to obtain information it uses in making decisions and to store information that may be used in later decisions. The decision-making component decides what transformations should be performed (which, for interior regions, translates to which subordinate regions should be executed) to manipulate a query. Conversely, the results of transformations can be input to decisions about how to (and whether to) continue the transformation process.

The decision-making functionality consists of components to choose a focus for processing (i.e., a query and goal), choose a transformation to execute, decide what to record in the store as a result of a transformation, decide when to terminate processing, and determine the result of the region's execution. These decision "modules" are reflected in the execution depicted in Figure 4.

The region focuses on a particular query and goal, and chooses a process for transforming the query. These two decisions are complementary—the transformation to perform depends on the query and goal, and the query and goal chosen may depend on what transformations are available. This duality is evident in rule-based optimizers, where queries are matched to transformation rules and a best query/rule pair is chosen for next execution. In Epoq, these decisions are made by a planning system. The query and goal form a task for a region to perform, and the planning system finds a sequence of transformations that may perform the required task.

Termination refers to the conditions under which the region stops its processing and returns to its parent. These conditions are checked after each query transformation

```
initialize store
while termination conditions not met do
   choose a query and goal
   choose a transformation to process
   transform query
   update store with result info
endwhile
return "best" query and success indicator
```

Figure 4: Basic Region Execution

(see Figure 4) and, if the conditions are met, processing is completed. In general, the conditions for termination are built into the control. For example, a control may decide to terminate after processing a certain number of transformations, after finding a particular number of alternative queries, after generating an alternative that attains the region's goal, after finding no good prospects for further processing the input query, etc.

However, the hierarchical organization of region processing means that a parent region may want (or need) to have some control over the termination of its subordinate regions. Such control can be implemented by sending termination conditions to a child region through the parent-child interface. The disjunction of the built-in conditions and the parent's condition then become the termination criteria for the region.

A parent can send two kinds of termination conditions: conditions involving the state of the query and conditions involving the utilization of resources. The former can be submitted to a subordinate as Boolean functions with one parameter—a query. At each termination check, the child can supply the query argument to the parent's termination function and execute the function. For example, suppose a parent will be satisfied if a subordinate that works to improve cost can improve the cost of the query by 15%. The parent would supply a Boolean function Improve(initial query, current query) that computes the percentage cost difference between an initial query state and the current query state and returns true if that percentage is greater than or equal to fifteen. The parent would send a closure of Improve, with the initial query instantiated, to its child through the interface. At each check for termination, the child supplies the current query and executes the function.

Termination conditions involving resource utilization generally require reasoning on the part of the child region. As a result, if a region wishes to convey such conditions to its children, the children must be constructed to directly respond to the particular conditions. For example, suppose a parent wishes to set time limits on a child's execution. The child must be able to interpret the time limits and, if it in turn calls other regions, must be able to apportion time limits to its children.

The final decision made by a region is the choice of query result. This decision is related to the processes that store alternative queries since the final query will be one of the stored alternatives. It is also related to the region's goal, since a region will choose (if possible) to return an equivalent

query that meets the goal. A region must return a query that is equivalent to the input query, thus if no satisfactory alternative is stored the region must return the original query.

A high-level view of region execution is depicted in Figure 4. Basically, a region continues transforming a query until the decision to terminate the processing is made. A region first chooses a query or subquery to process and a goal for that processing. An initial query and goal are provided by a parent region through the interface, but as processing continues the region may work on intermediate results, or on subqueries, with different goals in order to try to attain the required goal on the initial query. Intermediate transformation results may be stored, and after termination a result to be produced by the region is chosen. That result may, or may not, reflect success at achieving the initial goal for the transformation process.

The optimizer control is, in effect, choosing sequences of subordinate regions to manipulate a query, since the transformations it chooses to perform on queries are subordinate region executions. The extensibility of an Epoq optimizer requires, then, that the control component react to the addition of new regions. We satisfy this requirement with a control that is based on planning the optimization process.

5.1 Planning-Based Control over Regions

Planning reduces a task to primitive actions [2]. In the Epoq context, a region's task is to achieve some goal on a query, and primitive actions are subordinate region executions.

Plans use goals to describe tasks and subtasks that can be performed to achieve other goals. The planning process uses goals to progressively break a task into subtasks, and eventually to primitive tasks (i.e., region executions). Thus the result of planning is a sequence of subordinate region executions that may achieve the region's goal.

Plans are defined in a rule-based language, where the rules provide heuristics about potentially good interactions between region executions. A rule search engine matches rules to the current goal and control state (including the current query) and chooses rules to execute. Rules describe three kinds of actions that can be performed: planning actions, primitive actions, and memory updates. Primitive actions result in subordinate region executions. Planning actions induce a forward chain through rule goals to find primitive actions. Update actions modify the region's working memory with information that may be used later in the rule search.

The rule language and working memory are patterned after rule-based languages such as OPS5 [4]. A major difference, though, is that the Epoq rule interpreter interleaves rule manipulation with the execution of subordinate regions. This interleaving of planning and execution is required because subordinate regions can fail to achieve their stated goal, and the success or failure of a region execution to achieve its goal affects the optimization process. In other words, the sequence of actions that is generated as a plan is affected by the execution of those actions.

Get_Cost:	Query	$\longrightarrow$	cost
Get_Equiv:	Query	$\longrightarrow$	QuerySet
Choose_Best:	QuerySet, Goal	$\longrightarrow$	Query
Prune:	QuerySet	$\longrightarrow$	QuerySet
Add_Choice:	QuerySet, Query	$\longrightarrow$	QuerySet

Figure 5: Methods supporting decisions and queries.

5.2 Control Store

The control store supports decision-making by maintaining information used during region execution. Items in the store form the working memory of a control. Items are named and typed so that they can be directly accessed by the region control and can be manipulated by methods defined over their type. The control state consists of the contents of the store, along with information about the current processing step.

The store always includes an initial and current version of the query being transformed, as well as intermediate (equivalent) versions of the query. Intermediate query versions provide choices of ways to execute a particular query, and also provide processing choices for an optimizer. For example, in a rule-based optimizer intermediate versions of a query or subquery are usually maintained and the rule search engine (decision-making component) will match rules to these versions for further processing.

In order to support control decisions, the store includes a log; i.e. a history of previous processing. The log is used to keep track of the actions performed during the processing of queries. Such information can be used in control decision-making, for example, to prevent repetitive processing. It will also be used to support the undoing of partial results after failure to achieve a goal.

The store may also include other data defined by a particular control. For example, data could be used to help control rule execution. The working memory elements can be matched in rules and, in that way, can provide information referenced in rule actions as well as information that helps control the selection of rules to execute.

A region's store is local to the region. Any information that should be visible to other regions is passed as arguments through the region interface. In general, a region's control store persists for the duration of the region execution.

5.2.1 Query Storage and Management

The store will contain queries, and two relationships between these queries: subquery and equivalence. The subquery relationship is an explicit part of the query—arguments to a query can be other queries. Equivalence relationships are instances of a type called QuerySet. The instances of this type are, abstractly, sets of queries; all queries in a single queryset are equivalent to each other. The store will always contain at least one queryset—the set of queries equivalent to the initial query.

Some methods for supporting decisions about queries and querysets are shown in Figure 5. Methods Get_Cost and Get_Equiv retrieve information about the state of a query. The Get_Cost method retrieves the current cost of a query.

That cost will depend on the cost model being used in the optimizer. Retrieving the cost is supported by methods of the cost model. The Get_Equiv method returns the queryset associated with the input query parameter. This gives access to the queries that are equivalent to the input query, thus allowing further processing such as iterating through the equivalent queries, or choosing an equivalent query for processing.

Methods defined for Type QuerySet may involve decision-making. For example, a queryset supports a Choose_Best method to find the query, in a set of alternative queries, that best achieves the required goal. Add_Choice is used in the control whenever a new query is generated through transformation. The region control can decide, through the implementation of Add_Choice, whether or not a particular transformation result is stored for future manipulation. Method Prune further manages querysets by removing equivalent queries that no longer are useful to the control.

The Query and QuerySet types are global to an optimizer, but the type representations and method implementations may be redefined within a region. For example, if a region processes a query as simply a sequence of transformed expressions, then the representation could store only the most recent query version. In this case, the Add_Choice routine would replace the previous query with the new choice, the Prune routine would be null, and the Choose_Best routine would simply return the query.

5.2.2 Log Management

The log is a record of actions taken in a region. It includes records of query modifications (i.e., transformations) and may include a record of modifications to other information stored in memory. The log can support control decisions by recording the results of previous decisions. It can also support a simple recovery scheme, by offering the ability to restore the memory to a point prior to some transformation of a query.

The log must support a method Append(Log,LogRecord) to add a new record to the log. The Retrieve actions supported by a log depend on the requirements of the particular decision-making component, and could include retrievals to answer such queries as: What was the last transformation performed? What is the last transformation performed on query Q? What is the last transformation that resulted in query Q? Has query Q been transformed yet? Has a particular transformation been used?

5.3 Rules for Planning

The planning process uses rules that describe heuristics for good interactions between optimization tasks. These heuristics guide the decision-making process.

Rules have the general form

$$\textit{condition test} \longrightarrow \textit{action sequence}$$

with the semantics that *if* the left hand side condition is satisfied, *then* the actions on the right hand side are executed in sequence. The left hand side conditions are predicates over working memory and applicability predicates over the current query. The right hand side actions are planning or primitive actions, or memory updates.

```
GOAL PACKAGE Good_Rewrite
SEARCH priority by rule number ∧ one success per rule
TERMINATION no rule applies
METHODS
1. single_var(Q) ∧ has_op({Select},Q)
        ⟶   ACHIEVE Simple_Transform ON Q.

2. single_var(Q)
        ⟶   ACHIEVE Fast_Rewrite ON Q.

3. nested(Q)
        ⟶   ACHIEVE Flatter ON Q;

4. has_op({Join},Q)
        ⟶   ACHIEVE Join_Reorder ON Q.

5.      ⟶   ACHIEVE Lower_Cost ON Q.
```

Figure 6: Good_Rewrite goal package.

Applicability conditions for a region describe query states that can be manipulated by the region; the predicates in a rule further specify the queries to which the particular rule is applicable. For example, the Lower_Cost region (LC) specifies that it can process any query. However, one of the rules in the region may specify that it is a heuristic for processing queries with only Select, Project or Join operations.

Working memory predicates can test to see if objects with particular values are stored in working memory and, if so, may match variables to values of those objects. For example, in Figure 7 the WM_MATCH? predicate finds any Control_Flag in the local memory with an id of "rule1" and a val of "True", and matches variable ?Q to the query field of the matching memory item. ?Q is then bound for the next clause of the rule predicate and for the UPDATE action of the right hand side of the rule.

Rules also have an implied predicate that tests the current goal of the region. This predicate is implemented by collecting rules that test for the same goal into *goal packages*. For example, the goal package of Figure 6 collects five rules that are heuristics for rewriting different kinds of queries.

Goal packages modularize the planning process in the same way that regions modularize the optimization process. Collecting rules with the same goals into a package allows for the definition of package search strategies that can take advantage of the smaller sets of rules and of any particular characteristics of the rule sets [16, 19]. Each goal package has its own execution and private control store. Thus, as for regions, different goals can define different search control strategies and termination conditions.

The right hand side of a rule describes a sequence of steps, or subtasks, that should be taken to attain the desired goal. The steps are either goal actions or memory updates. A goal action has the form

ACHIEVE ≪goal_index≫ ON ≪query_variable≫
[GIVING ≪query_variable≫]

indicating that the goal identified by ≪goal_index≫ be

```
GOAL PACKAGE Fast_Rewrite
SUCCESS one rule successful
SEARCH one success per rule
TERMINATION no rule applies
-----------------------------------------------------
LOCAL STORE
   TYPE Control_Flag: Record of (id: string,
                        val: Bool, query: Query)
   OBJ Q1: Query
-----------------------------------------------------
METHODS
   ⟶   ACHIEVE Simple_Transform ON Q
                                    GIVING Q1;
       UPDATE_Local(Add, (Done,(Control_Flag
         new(id = rule1, val = True, query = Q1)))).

WM_MATCH?_Local (Control_Flag
   (id = rule1 ∧ val = True ∧ query = ?Q))
   ∧ est_cost(?Q) ≤ .5 * est_cost(Q)
   ⟶   UPDATE_Local(Q ← ?Q).

WM_MATCH?_Local (Control_Flag
   (id = rule1 ∧ val = True ∧ query = ?Q))
   ∧ est_cost(?Q) > .5 * est_cost(Q)
   ⟶   ACHIEVE Lower_Cost ON ?Q GIVING Q.
```

Figure 7: Fast_Rewrite goal package.

achieved on the query represented by the variable in the ON clause. The GIVING clause is optional and can be used to specify a new variable for storing the result. If no GIVING clause is used, the input query is modified.

Achieve actions describe subtasks that need to be accomplished to attain the rule goal. The goal can be a primitive goal that will be attained by the execution of a subordinate region, or a planning goal described by a goal package. Planning goals elicit a forward chain through goal packages, searching for primitive goals to execute.

For example, consider rule 4 of Figure 6. The goal named "Join_Reorder" is a primitive goal of the region that can be directly attained by a subordinate region (OJ or DP). On the other hand, goal "Flatter" (rule 3) is a planning goal of the region. Subtasks that can achieve this goal are described by another goal package (not shown, see [15]).

Memory update actions can be used to affect subsequent rule control. For example, the Fast_Rewrite goal package (Figure 7) uses the local control store to implement the conditional decision structure of the following pilot pass style algorithm:

1. Achieve Simple_Transform on Q Giving Q_1.
2. If Q has been improved by 50%, quit;
3. Else Achieve Lower_Cost on Q_1.

A variable named Done of type Control_Flag is declared in local memory and is used by the rules to determine when the first rule has completed. The goal package assumes that the result is stored, by the rules, in variable Q. Thus, if the Q_1 result is satisfactory the second rule of the package must move Q_1 to Q . The third rule in the package uses the GIVING clause to move the result to Q.

Memory update actions can also be used to affect global processing. In particular, a rule might insert a record into global memory to suggest a goal that would be desirable to achieve on an intermediate query result.

5.4 Region Execution

The desired execution model for the region's planning system is eager in the sense that a single rule (i.e., task) is executed to completion before trying any other rules. The Achieve actions of the rule induce either a forward chain through rule packages or a region execution, and Update actions modify working memory. The actions of a rule are performed in sequence, so the execution will basically chain through rule packages until a primitive goal is encountered. Primitive goals are immediately executed, then processing is returned to the next step in the rule that invoked the primitive goal action. When a rule successfully completes, the newly transformed query is stored and processing may continue with another query, goal and rule. If a rule fails, another rule with the same goal will be tried.

In order to effect this execution model, processing is divided into the following modules:

- High-level region execution – interacts with the interface modules; searches for queries, or subqueries, and goals to process; executes goal packages
- Package execution – chooses rules to achieve the required goal
- Rule Execution – processes, in sequence, the actions of a rule
  - Achieve actions – invoke processing to achieve the stated goal
  - Update – updates working memory
- Primitive Action – executes subordinate region

High-level region execution begins execution of the first package and continues execution until region termination conditions are met. Package execution continues executing rules to achieve the package goal until its termination conditions hold. The combination of these two execution modules is basically a rule search system. Details of the execution modules are given in the following subsections.

5.4.1 High-level region execution

The invocation of a region passes information needed for the region execution through the parent-child interface and begins execution of the control loop of Figure 8.

This loop, along with the nested loop for package execution, effects a search through the rules that guide the optimization process for rules that match the current query and control state. Initially, a region is given a query and a goal to achieve on that query. The high-level execution module can decide to work on that query, or may decide to process a subquery. As a query is processed, transformations of the query create alternative queries that may be chosen for processing in later executions of the high-level loop.

```
INPUT: query Q, goal G, termination conditions
OUTPUT: query, Boolean (success indicator)

initialize current goal, current query,
     global termination conditions
while ≪termination conditions≫ not met do
     choose current goal and query
     invoke package for goal
endwhile
result ← Choose_Best(Get_Equiv(Q), G)
return result and Success?(Result)
```

Figure 8: High-level region Execution

5.4.2 Package execution

The combination of high-level and package execution is analogous to rule search. The high-level execution module determines what goal will be pursued, and package execution uses additional conditions on rules to find rules to achieve the goal. The main job of package execution is to ensure that

```
INPUT: query Q
OUTPUT: query, success indicator

Q_save ← Q
set up a priority queue of rules whose conditions are met
while (queue not empty)
    ∧ (≪termination conditions≫ not met) do
        execute first rule in queue on Q
        if rule completes
            reinitialize priority queue of rules
        otherwise
            remove first rule from queue
    endwhile
    return result and Success?(result, package goal)
```

Figure 9: Package Execution

rules are executed to completion. A rule that completes may achieve the required goal, but a rule that doesn't complete will not achieve the goal. Thus, package execution will try rules applicable to a query until a rule executes to completion.

A single rule application may meet the package's goal, but the rules in a goal package may also be applied iteratively to work towards the goal. For example, a goal of lower cost may actually be achieved by successively lowering the cost of the query until a satisfactory result is obtained. The termination conditions of the package determine the amount of iteration necessary to achieve the goal.

In order to find a rule that may be successful, the package execution module uses the conditions on the left-hand sides of rules to determine a priority ordering for the rules. For example, the search strategy for the Good_Rewrite goal package (Figure 6) defines that rule priority is by number, as long as the rule has not already been successfully applied in the region. Thus, if a query satisfies the conditions of rules 1, 2 and 5, but rule 1 was applied earlier in the search, the priority queue will contain rules 2 and 5, in that order.

After a rule is applied, the module checks to see if the rule executed to completion. If the rule completes, termination conditions are used to determine whether the package will further manipulate the result. If the rule doesn't complete, other rules are tried, in the priority order, until a rule is successful or there are no more rules that can be applied to the query.

One built-in termination condition for a rule package is that all rules have been tried. The 'empty queue' condition reflects this termination condition. Other termination conditions for a package are determined by the requirements of the package goal. Termination may be related to success at achieving the goal but must also have conditions that are independent of success. In addition, region termination conditions are combined with any global termination conditions to ensure that no more rules are attempted when global termination is indicated.

5.4.3 Rule execution

The required execution is that a rule runs to completion before any further rules are executed. This is a major difference between the Epoq rule engine and most other rule systems. In Epoq we require sequential execution of rule steps, with no intervening rule execution. A major motivation for this control is the interaction between the results of execution of subordinate regions and the planning system. In particular, the fact that subordinate regions and therefore, eventually, rules can fail means that we may need to recover from changes made to control state during the execution of the rule. By not allowing concurrently executing rules, the transaction semantics of rules are simplified.[3]

The actions designated on the right hand side of a rule are executed in sequence until all actions have been successfully completed or until the first failing action. Memory update actions cannot fail, but Achieve actions can. In the event of failure, the control state must be recovered. Failure handling is discussed in Section 5.5. If all actions are successful, the rule is complete and execution returns to the package execution module.

Execution of an Achieve action depends on the goal of the action. If the goal is represented by a goal package, execution directly transfers control to the indicated package. There is no decision to be made here—all decisions about further processing are made in the goal package. If the goal is a primitive goal of the region, execution transfers to the primitive action execution module for that goal.

5.4.4 Primitive Action execution

A primitive action tries to directly satisfy a goal by executing a subordinate region that can satisfy the goal. Since more than one region may be able to achieve a particular goal, a primitive action must choose between the regions. Also, since a region may not succeed at achieving a goal, a primitive action must try alternative regions to ensure that

[3] Transaction and failure semantics for concurrently executing rules is an interesting topic for future research.

no region can achieve the goal before the action admits failure.

Primitive actions use region applicability information to determine which region to execute to achieve a goal. *Static applicability* predicates describe the form of queries that can be manipulated by a region. They are provided to a parent by its subordinate, and can be applied by the parent to decide whether to eliminate a region from consideration. If there is more than one statically applicable region the results of *dynamic applicability* functions, executed at the subordinate regions, are used to further filter out regions or to determine an order for trying similar regions. If the dynamic applicability information cannot distinguish a single region to try, similar regions are ordered randomly. This process is described in Figure 10.

```
INPUT: query Q
OUTPUT: query, success indicator

initialize success := false
initialize continue := true
use rule conditions to find applicable regions
if there is more than 1 applicable region
    order regions (dynamic applicability; random choice)
while continue
    ∧ (≪termination conditions≫ not met) do
    allocate termination conditions for subordinate and
    execute first region in queue on Q' ← Q
    if region returns success
        set success := true and continue := false
    otherwise
        remove region from queue
        if queue-empty then set continue := false
endwhile
if success then Add_Choice(Get_Equiv(Q), Q')
return Q' and Success?(Q', this primitive goal)
```

Figure 10: Primitive Action Execution

A Primitive action sends a copy of the query to be processed to a subordinate region. The subordinate region will make modifications directly to the query copy and, if the region is successful, the primitive action can update the global query information with the transformed result. The copy semantics put all decisions about maintaining transformed results in the domain of the query manipulation routines. The advantage of this approach is the protection of the parent region's memory. The disadvantage of this approach is that transformed results will usually reference many of the same subqueries as the original query. These references can get lost unless the query manipulation routines can recognize common subexpressions using just the copied query representations.

The search for a region to achieve the required goal requires only a single successful result. The primitive action execution is terminated when a subordinate region is successfully executed, when there are no more regions to try, or when global termination conditions indicate that no more searching is desired.

5.5 Handling Failure

The planning system uses the heuristics defined in the planning rules, along with applicability information provided by subordinate regions, to determine sequences of region applications that will transform a query to attain the desired goal. However a region execution will not necessarily achieve the region's goal on every query to which the region is applicable. This happens because applicability information provides necessary, but not sufficient, conditions for the query in order for the region to be able to process the query and attain the goal.

Region execution is a primitive action, and region execution failure can propagate into primitive action failure, rule failure, goal package failure, and eventually parent region failure. Thus, each of these modules must accommodate the possibility of failure.

The primitive action module handles region execution failure by executing other regions that can achieve the same goal as the failed region. It will try all regions until it finds one that can achieve the goal, or it finds that no region will attain the goal. In the latter case, the primitive action admits failure, which propagates to the rule.

The sequence of actions on the right hand side of a rule are considered as a single transaction for recovery purposes. If any action on the right hand side of a rule fails, the rule itself fails. Any updates made to memory, in particular updates to query choices, must be backed out to a point before the first rule action. This recovery is handled through maintenance of a log of updates to memory state, delimited by transaction boundaries. A transaction, in this case, encompasses the execution of an entire rule.[4] A transaction starts before the first action of a rule is executed and ends when the last rule action is successfully completed. Since rules can be nested within other rules (through Achieve actions) the recovery of a transaction can require backing-out of successful, nested rule executions.

When a goal package terminates without achieving success, that failure propagates to the rule executing the Achieve action that invoked the package. If the goal package is a high-level execution no further recovery is required. In this case the high-level execution module must choose another query/goal pair to execute.

Failure of a primitive action or rule will not necessarily propagate to region failure, nor does successful processing of goal packages indicate that the region will be successful at achieving its goal. The success, or failure, of a region to attain its goal depends solely on the definition of success for the goal and the queries that are generated as choices during the region's processing.

[4] There are certainly situations in which one would want to save the results of intermediate actions, since they may offer opportunities for later processing. Potentially good intermediate results could be indicated by incorporating save point actions in rules – where a save point indicates at point at which alternative queries generated by actions of the rule should be made persistent. Since rules are effectively nested, an interesting question is how to handle save points in the resulting nested transactions. We defer an answer to the semantics of save points in nested transactions to later work.

6 Related Work

The main difference between the Epoq approach to optimization and other approaches for extensible or object-oriented systems is that Epoq provides for extensibility of the optimization process itself. Most extensible optimizers (e.g. [5], [7], [8], [10], [20]) provide a fixed strategy for searching for and applying rules for query transformation. In other words, although the possible optimizer results can be extended, the optimization process is fixed. Proposals for object-oriented optimizers either use one of these extensible approaches [1] or provide some fixed sequence of optimizer processing strategies [3, 21].

The Epoq approach is motivated by the desire to extend an optimizer with new strategies for optimization. In other words, the optimization process can be extended. This leads to the need for an extensible control to direct the optimization process. Optimizer strategy extensibility also motivates the approaches of Lanzelotte and Valduriez [11] and Sciore and Sieg [18], so we discuss these in more detail here.

Sciore and Sieg [18, 19] group query rewrite rules into modules, where different modules can have different rule search and termination strategies. We use a similar approach in our planning system. The difference here, of course, is that our rules plan the operation of the optimizer itself.

In the Sciore and Sieg approach, modules interact with each other in ways that are fixed when the modules, and the rules, are written. In Epoq, control over the execution order of regions is separated from the regions being controlled. This results in a more modular optimizer and a control which can respond to the particular query being processed and to the dynamics of the processing of that query.

Lanzelotte and Valduriez address the problem of customizing the optimization process to a particular query by focussing on an extensible way to define strategies for manipulating query expressions [11]. Different search strategies are related through a sub-type hierarchy of strategies, with higher level specifications describing the methods present in a search strategy and lower level specializations (i.e., the specific strategies) implementing these methods (in different ways). A particular strategy can be modified by changing the implementation of any of its methods.

Different strategies are integrated in the sense that they all specialize a common model for search strategies. The common model is the search strategy for the optimizer and, at optimizer execution time, a specialization of the strategy can be used to process a particular query. The search strategy specializations are analogous to our leaf regions (and to the modules of Sciore and Sieg) and, indeed, may provide useful tools for specifying the implementation of regions. However, this work does not address the integration of the different strategies to process a single query at optimizer execution time. Given a query to process, one of the strategies present in an optimizer is chosen to optimize that query.

The Epoq approach to query optimization is related to the knowledge-based approach of [22]. Epoq regions form a knowledge base of information about query processing strategies. The control presented here contains knowledge about ways to combine these strategies to process a query.

The Epoq planning-based control is based on rule-based programming languages [4] and reactive planning [6]. Our rule execution system, though, differs from either of these. A rule is a task that, if successful, will result in a desired transformation of a query. Thus, a rule describes a consistent way to process a query. Our rule engine enforces a transaction type of semantics on rules; we require that a rule execute to completion before new tasks are considered.

7 Summary

In an Epoq optimizer each region is a separate module that interacts hierarchically with other modules through a common interface and a planning-based control. The potential interaction of modules is statically defined by control rules, region goals and applicability, but the actual interaction between regions depends on the query being processed.

A region module provides, through the interface to its parent, a goal for its processing and predicates characterizing the queries it expects to be able to manipulate to achieve the goal. A parent control uses this information as it decides how to process a query.

A parent region must determine an order for executing subordinate regions to transform a query to achieve its own goal. Given a particular query, a region control plans a sequence of transformations (i.e., an ordering of subordinate region executions) that will, hopefully, manipulate the query to achieve the region's goal. The planning process is influenced by intermediate results of the plan—i.e., planning is interleaved with the execution of subordinate regions.

Planning rules describe heuristics for interactions between regions. These rules also support extensibility in the optimizer. The addition of a new region to an optimizer may require new rules to describe how this region may successfully interact with other regions. These rules are added to the planning system's rule set and manipulated in the same way as existing rules. The extensibility of the control itself is a unique feature of the planning-based control in supporting the extensibility of Epoq.

Acknowledgements

The research described in this paper was performed while Umeshwar Dayal was affiliated with Digital Equipment Corporation, Cambridge Research Lab, and Gail Mitchell was affiliated with Brown University.

Partial support for this work was provided to Brown University by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052 ARPA order 8225, and contract DAAB-07-91-C-Q518 under subcontract F41100.

References

[1] Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proceedings ICDT*, Paris, France, 1990.

[2] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, MA., 1985.

[3] Sophie Cluet. *Langages et Optimisation de Requêtes pour Systèmes de Gestion de Base de Données Orientés-Objet.* PhD thesis, Université de Paris-Sud - Centre d'Orsay, June 1991.

[4] Thomas A. Cooper and Nancy Wogrin. *Rule-based Programming with OPS5.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[5] Béatrice Finance and Georges Gardarin. A Rule-Based Query Rewriter in an Extensible DBMS. In *Proceedings of the 7th International Conference on Data Engineering*, pages 248–256. IEEE, 1991.

[6] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University, January 1989. YALEU/CSD/RR#672.

[7] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, Univ. of Wisconsin-Madison, November 1987.

[8] Goetz Graefe. Volcano, an Extensible and Parallel Query Evaluation System. Technical Report CU-CS-481-90, University of Colorado at Boulder, July 1990.

[9] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *SIGMOD Proceedings*, pages 160–172. ACM, May 1987.

[10] Laura M. Haas et al. Extensible Query Processing in Starburst. In *SIGMOD Proceedings*, pages 377–388. ACM, June 1989.

[11] Rosana S. G. Lanzelotte and Patrick Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proceedings of the 17th VLDB Conference*, pages 363 – 373, 1991.

[12] Rosana S. G. Lanzelotte, Patrick Valduriez, M. Ziane, and J.-P. Cheiney. Optimization of Nonrecursive Queries in OODBs. In *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, December 1991.

[13] Christopher V. Malley and Stanley B. Zdonik. A Knowledge-Based Approach to Query Optimization. In *Proceedings of the First International Conference on Expert Database Systems*, pages 329–344, 1987.

[14] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. An Architecture for Query Processing in Persistent Object Stores. In *Proceedings of the Hawaii International Conference on System Sciences*, volume II, pages 787–798, January 1992.

[15] Gail A. Mitchell. *Extensible Query Processing in an Object-Oriented Database.* PhD thesis, Department of Computer Science, Brown University, May 1993. Technical report CS–93–16.

[16] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD Proceedings*, pages 39–48. ACM, June 1992.

[17] Arnon Rosenthal, Umeshwar Dayal, and David Reiner. Speeding a Query Optimizer: The Pilot Pass Approach. Computer Corp. of America, unpublished note.

[18] Edward Sciore and John Sieg, Jr. A Modular Query Optimizer Generator. In *Proceedings of the 6th International Conference on Data Engineering*, pages 146–153, 1990.

[19] John Connor Sieg, Jr. *Making extensible database technology work.* PhD thesis, Boston University, 1989.

[20] Michael Stonebraker. Inclusion of New Types in Relational Database Systems. In Michael Stonebraker, editor, *Readings in Database Systems.* Morgan Kaufmann Pub. Inc., 1988.

[21] Dave D. Straube and M. Tamer Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[22] H.J.A. van Kuijk and P.M.G. Apers. Semantic Query Optimization in Distributed Databases: A Knowledge-Based Approach. In Goetz Graefe, editor, *Workshop on Database Query Optimization*, pages 53–58, Portland, OR, May 1989.