An Extensible Query Optimizer for an Objectbase Management System*

M. Tamer Özsu   Adriana Muñoz   Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

We describe an extensible query optimizer for objectbase management systems. Since these systems are expected to serve data management needs of a wide range of application domains with possibly different query optimization requirements, extensibility is essential. Our work is conducted within the context of TIGUKAT, which is a uniform behavioral system that models every system component as a first-class object. Consistent with this philosophy, we model every component of the optimizer as a first-class object, providing ultimate extensibility. We describe the optimizer architecture and how the optimizer components are modeled as extensions of a uniform type system.

1 Introduction

The early objectbase management systems (OBMSs) have been criticized [SRL$^+$90] for their lack of declarative query capabilities (both languages and optimizers). These facilities are common in the state-of-the-art relational database management systems (DBMSs).The newer commercial OBMSs and research prototypes have started to include these capabilities, but their optimization is still not very well understood.

We have been studying query optimization issues for some time [SÖ90, SÖ95]. Our more recent work is within the context of the TIGUKAT project[1] [ÖPI$^+$95]. TIGUKAT is an OBMS with an extensible object model characterized by a purely behavioral semantics and a uniform approach to objects which treats everything (including queries) as first-class objects. The query model consists of a calculus, an equivalent algebra and an SQL-based user query language [PLÖS93a].

An important design consideration for an OBMS query optimizer, especially one which has a very flexible and extensible object model such as TIGUKAT, is *extensibility*. This allows the optimizer to accommodate different search strategies, different algebra specifications with their different transformation rules, and different cost functions. There are a number of reasons for extensibility [ÖB94]:

1. The application domains to be supported by the object-oriented technology have different query processing and optimization requirements. Therefore, the optimizer design should support easy customization of the system.

2. There is no consensus about an object algebra and there are many proposals. Furthermore, OBMSs should allow application-specific algebra operators to be defined and managed by the system (see, for example, [BG92]). The optimizer should be able to deal with these operators uniformly.

3. The optimization techniques for OBMSs are not fully developed and the alternatives are not completely understood. Thus, the techniques that are included in today's optimizers are likely to change as research results emerge. It should be possible to easily incorporate these changes into the system.

4. Even in traditional query optimizers (i.e., relational ones which deal with a well-defined and fixed set of algebra operators), the entire set of transformation rules that are exploited by the optimizer in determining the alternative execution plans are unlikely to be fully specified [HCF$^+$89]. Therefore, it is necessary to be able to define new rules for the optimizer as they are discovered.

In this paper we describe the architecture of an extensible query optimizer for OBMSs. Even though this work is conducted within the context of the TIGUKAT system, the general approach is valid for other OBMSs and the specifics hold for any system which uniformly models system components as first-class objects. The architecture described in this paper allows every aspect of the query optimizer (search strategies, search space as identified by the algebraic transformation rules, and the cost functions) to be extended. It incorporates an object-oriented approach to extensibility, representing components of the optimizer as objects.

The remainder of the paper is organized as follows. In Section 2, we review some of the other extensible optimizers. In Section 3 we summarize the TIGUKAT object model and the query model. This discussion is restricted to only

*This research is supported by the Natural Sciences and Engineering Research Council of Canada under grant OGP0951.

[1]TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

those object model and query model characteristics that are relevant to the optimizer design described here. The overall architecture of the query optimizer is described in Section 4. Sections 5, 6, and 7, respectively elaborate on the extensible representations of the search space, the search strategy, and the cost functions. Section 8 describes how these pieces are put together in the design of the optimizer. This discussion is aided by the presentation of an example query creation and optimization scenario. We conclude, in Section 9, with a discussion of the advantages of our approach and the current state of our development.

2 Related Work

There has been significant amount of research on extensible DBMSs. EXODUS [CD86], Genesis [BBG+86], and Postgres [SK91] are examples of such systems. These provide customization of system components for individual applications. In this paper, our interest is in the extensibility and customization of the query optimizer rather than the customization of the overall system architecture.

Many existing OBMS optimizers are either implemented as part of the object manager on top of a storage system, or they are implemented as client modules in a client-server architecture. In most cases, the search algorithms, transformations rules and cost functions are "hardwired" into the query optimizer. Rule-based query optimizers [Fre87] provide a limited amount of extensibility by allowing the definition of new transformation rules. However, they do not allow extensibility in other dimensions.

The Open OODB project at Texas Instruments concentrates on the definition of an open architectural framework for OBMSs and on the description of the design space for these systems. Query processing in Open OODB [BMG93] is largely influenced by the extensibility goals of the Open OODB project. The query optimizer, built using the Volcano optimizer generator [GM93], is extensible with respect to algebraic operators, logical transformation rules, execution algorithms, implementation rules (i.e., logical operator to execution algorithm mappings), cost estimation functions, and physical property enforcement functions (e.g., presence of objects in memory). The search algorithms are built-in, however, and cannot be extended.

Quite a different approach to extensibility is described in [MZD92], where the search space is divided into *regions*. Each region corresponds to an equivalent family of query expressions that are reachable from one another. The regions do not have to be mutually exclusive and differ in the queries that they can manipulate, the control (search) strategy that they use, or in the objectives that they want to achieve in query manipulation. For example, one region may have the objective of minimizing a cost function, while another region may attempt to put queries in some desirable form. Alternatively, one region may cover transformation rules that deal with simple select queries, while another region may deal with transformations for nested queries. Since a region incorporates a transformation strategy, it can be treated as the transformation of an input query to an equivalent output query. This is what allows movement between regions. There is a global control strategy to determine how the query optimizer moves from one region to another [MDZ93]. Since search strategies, transformations and optimizations can change from one region to another, the approach allows strategies to be changed within a single query.

Starburst is an extensible DBMS with an extensible query optimizer [HCF+89]. It extends SQL and allows user-defined extensions to the language. It also allows extensions to the transformation rules that are used during optimization. The rules are of the *condition-action* type and the base system provides three different classes of rules: predicate migration, projection push down, and operation merging. System implementors can either use these rules or define application-specific rules of their own. Starburst provides a limited set of search algorithms and does not allow their extension. Cost functions cannot be changed or extended either. Since the underlying model is relational, there is no need to extend the set of algebraic operators.

Our approach differs from all of these in that we use an object-oriented approach to extensibility that models each component of the query optimizer, listed above, as an object. This is similar to [LV91] in the general approach, but differs in the details and in the extent object-orientation is used as the fundamental means of extensibility. The incorporation of these optimizer components into the type system provides extensibility via the basic object-oriented principle of subtyping as we describe in the rest of this paper. Consequently, we are able to allow extensibility of *all* optimizer components.

3 TIGUKAT Object and Query Models

The TIGUKAT object model is defined *behaviorally* with a *uniform* object semantics. The model is *behavioral* in the sense that all access and manipulation of objects is based on the application of behaviors to objects, and the model is *uniform* in that every component of information, including its semantics, is uniformly modeled by objects and has the status of a *first-class object*. This uniformity provides reflection to the model [PÖ93].

The primitive type system of TIGUKAT is depicted in Figure 1. The primitive objects of the model include: *types* for defining common features of objects; *atomic types* (reals, integers, strings, etc.); *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying implementations of behaviors over types[2]; *classes* for automatic classification of objects based on type[3]; and *collections* for supporting general heterogeneous groupings of objects. In the remainder of the paper, an identifier beginning with the prefix T_ refers to a type, **C_** refers to a class, **L_** refers to a collection, and *B_* refers to a behavior. For example, T_person is a type reference, **C_person** a class reference, **L_seniors** a collection reference, *B_age* a behavior reference, and a reference such as **xyz** without any prefix represents some other application specific reference. In this paper we discuss only a subset of the primitive types and behaviors that are relavant to this topic.

The access and manipulation of an object's state occurs exclusively through the application of behaviors. An important primitive behavior defined on objects is *identity equality* that compares two object references based solely on the identities of the objects they denote. This is the only type of equality that is defined in the primitive type system.

In addition to *types* and *classes*, we define a *collection* as a general user-definable grouping construct. A *collection* is similar to a *class* in that it groups objects, but it differs in the following respects. First, no object creation may occur through a collection; object creation occurs only

[2]The association of behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors.

[3]Types and their extents are separate constructs in TIGUKAT.

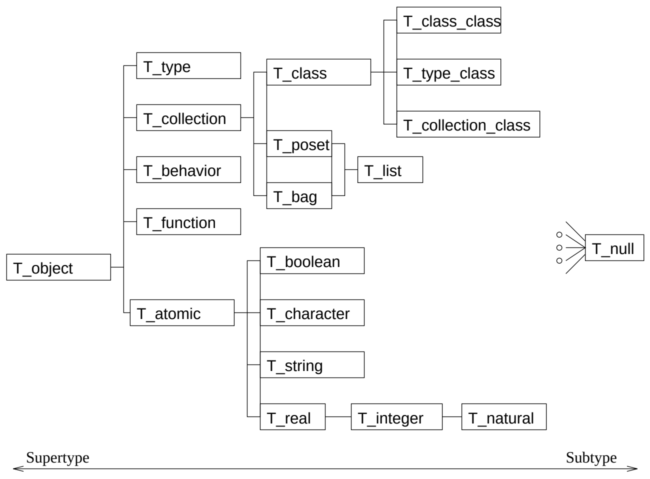

Figure 1: Primitive Type System

through classes. Second, an object may exist in any number of collections, but is a member of the shallow extent of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the subtype lattice whereas the management of collections is *explicit*, meaning the user is responsible for their extents. Finally, the elements of a class are homogeneous up to inclusion polymorphism while a collection may be heterogeneous in the sense that it can contain objects which may be of different types.

We define *class* as a subtype of *collection*. This introduces a clean semantics between the two and allows the model to utilize both constructs in an effective manner. For example, the targets and results of queries are typed collections of objects. This means targets also include classes because of the specialization of classes on collections. This approach provides great flexibility and expressiveness in formulating queries and gives *closure* to the query model.

Two other fundamental notions are *behaviors* and the *functions* (known as *methods* in other models) that implement them. We clearly separate the definition of a behavior from its possible implementations (functions/methods). The benefit of this approach is that common behaviors over different types can have a different implementation for each of the types. This is direct support for behavior *overloading* and *late binding* of implementations to behaviors, which are important aspects of object-oriented computing.

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of a behavior. The implementation of a particular behavior may vary over the types which support it. However, the semantics of the behavior remain constant and unique over all types supporting that behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code and a *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavioral application as the invocation of a function, regardless of whether the function is stored or computed. Functions are examined more closely in Section 4. We define queries as specialized functions so that they carry all the semantics of function objects, meaning they can be used as implementations of behaviors.

The query model consist of a complete object calculus, an equivalent object algebra, and an SQL-like user language. The entire query model is defined in detail in [PLÖS93b]. In the remainder of this section we provide an overview.

The syntax of the TIGUKAT query language (TQL) is based on the SQL *select-from-where* structure, and the formal semantics is defined by the object calculus. Thus, it combines the power of relational query languages with object-oriented features.

The calculus has a logical foundation and its expressive power is outlined by the following characteristics. It defines predicates on collections (essentially sets) of objects and expressions represent collections of objects that satisfy these predicates to give the language *closure*. It incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavioral applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has rigorous definitions of safety (based on the evaluable class of queries [GT91]) and typing which are compile time checkable. It supports controlled creation and integration of new collections, types and objects into the existing schema.

Like the calculus, the algebra is *closed* on collections. Algebraic operators are modeled as behaviors on the primitive type `T_collection`. They operate on collections and return a collection as a result. Thus, the algebra has a behavioral/functional basis as opposed to the logical foundation of the calculus. Composition over these behaviors brings closure to the algebra. We do not elaborate on the algebra operators in this paper.

The algebra and calculus are proven to be equivalent in expressive power, meaning that all queries expressed in one language can also be expressed in the other. Space limitations do not allow us to include them here, but in [PLÖS93b] we prove the equivalence of our object calculus and algebra in both directions and present the reduction of the user query language to the calculus. Moreover, the safety of our languages is proven in that report as well.

4 Optimizer Architecture

Query optimization can be modeled as an optimization problem whose solution is the choice of the "optimum" *state* in a *state space* (also called *search space*). In query optimization, each state corresponds to an algebraic query execution schedule represented as a processing tree [KBZ86]. The state space is a family of equivalent (in the sense of generating the same result) algebraic queries that can be generated by applying the transformation rules defined for the specific algebra. The goal is to move from one state to another using a *search strategy*, applying a *cost function* to each state and finding the one with the least cost. Thus, to characterize a query optimizer three things need to be specified:

1. the transformation rules that generate the alternative query expressions which constitute the search space;
2. a search algorithm that allows one to move from one state to another in the search space; and
3. the cost function that is applied to each state.

We model each of these components as objects as described in Sections 5 – 7.

TIGUKAT query model is a direct extension to the object model, defined by type and behavioral extensions to the primitive type system. We define a type `T_query` as a

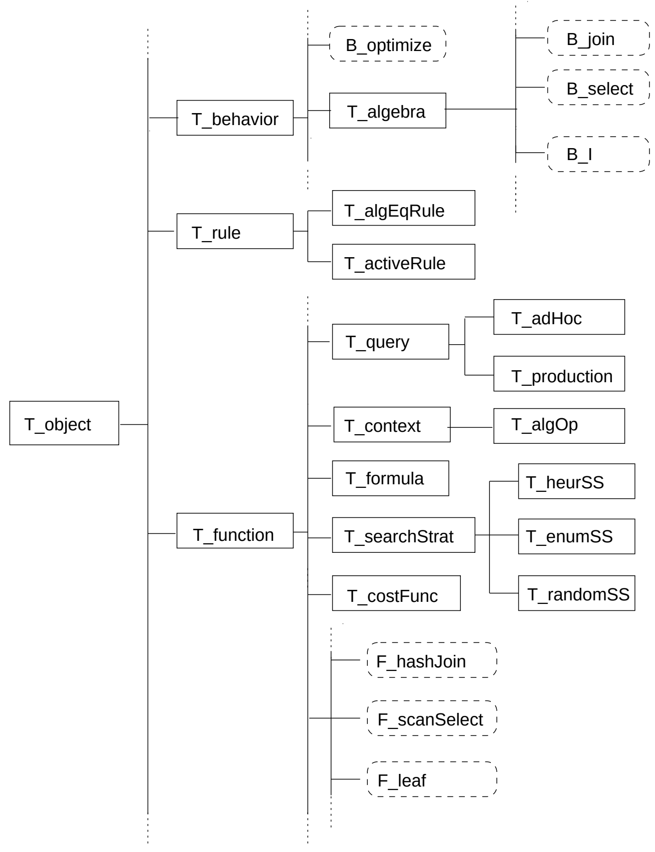

Figure 2: Optimizer as part of the type system

subtype of `T_function` in the primitive type system as illustrated in Figure 2. The dashed boxes indicate instances of types which are solid boxes. For example, *B_join* is an instance of type `T_algebra` which is a subtype of `T_behavior`. This means that queries have the status of *first-class objects* and inherit all the behaviors and semantics of objects. Moreover, queries are functions so they can be used as implementations of behaviors, they can be compiled, and they can be executed.

Since `T_query` is a subtype of `T_function`, it inherits all of the behaviors of `T_function` and defines new ones. Each function knows the types of its arguments and the type of its result (via *B_argTypes* and *B_resultType*, respectively), constituting its signature. They can be compiled (*B_compile*) and executed (*B_execute*).

For a query, *B_compile* is re-implemented to include translating the query statement into an algebraic expression represented as an *object algebra processing tree* (OAPT), optimizing it and generating an execution plan. Similarly, *B_execute* is re-implemented to submit the execution plan to the object manager for processing. There is the capability of storing the result of query execution so that it is returned the next time the query is posed without re-execution, allowing the amortization of optimization effort over multiple executions. In addition to the behaviors it inherits from `T_function`, `T_query` has native behaviors that return the initial (*B_initialOAPT*) and optimized OAPT (*B_optimizedOAPT*), the search strategy (*B_searchStrat*), the transformation rules used (*B_transformations*), the cost model function (*B_costModelFunc*), the family of execution plans generated during optimization (*B_genExecPlan*), the result of the query execution (*B_result*), and some behaviors to store and access statistical data as well as behaviors that start the optimization (*B_optimize*) and that generate a family of execution plans (*B_execPlanFamily*). The details of these behavior definitions are omitted here and can be found in [Mun93].

Incorporating queries as a specialization of functions is a very natural and uniform way of extending the object model to include declarative query capabilities. The major benefits of this approach are:

1. Queries are *first-class* objects, so they support the uniform semantics of objects. They are maintained within the objectbase and are accessible through the behavior of the object model.

2. Since queries are objects, they can be used in queries, and behaviors can be applied to them. This is useful in generating statistics about the performance of queries and in defining a uniform extensible query optimizer.

3. Queries are uniformly integrated with the operational semantics of the model and thus, queries can be used as implementations of behaviors (i.e., the result of applying a behavior to an object can trigger the execution of a query).

4. The query model is extensible in a uniform way since the type `T_query` can be further specialized by subtyping. For example, we can subtype `T_query` into `T_adHoc` and `T_production` and then define different evaluation strategies for each.

The TIGUKAT query optimizer follows the philosophy of representing system concepts as objects along the lines of [LV91]. We model the algebraic operators as objects, specifically as behaviors over the `T_collection` type. In the type lattice, they appear as instances of type `T_algebra` which is a subtype of `T_behavior`. In addition, the three components of the optimizer, namely the search space (`T_algOp` and `T_algEqRule`),the search strategy (`T_searchStrat`) and the cost function (`T_costFunc`) are modeled as objects (see Figure 2). The incorporation of these components of the optimizer into the type system provides extensibility via the basic object-oriented principle of subtyping and specialization. In the following three sections we discuss the modeling of the components of the optimizer as part of TIGUKAT's type system.

5 Representation of Search Space

As indicated earlier, compilation translates a query to a corresponding algebraic expression represented as an OAPT. OAPTs are slightly different from their relational counterparts in that OAPT nodes, even the leaves, uniformly correspond to functions that are implementations of algebraic operators.

The object algebra operators are modeled as behaviors on type `T_collection` whose implementations are modeled as instances of `T_function`. Since each algebraic operator has its own characteristics (i.e., predicates, functions to apply, etc), objects of type `T_function` are created according to the different TIGUKAT algebra operators. Furthermore, since there may be a number of different algorithms to implement each algebraic operator (e.g., different join algorithms), there may be many implementation functions as instances of T_function. For example, *F_hashJoin* is an object of type T_function that models the algorithm that implements the join algebraic operation *B_join* defined as a behavior on T_collection. This gives flexibility in redefining algebraic operators and in extending the algebra (e.g., by adding transitive closure for recursive query processing).

We cannot use these implementation functions as the nodes of an OAPT, however. The nodes of the tree should represent execution functions all of whose arguments have been marshalled. Therefore, we define T_algOp whose instances are functions with marshalled arguments and they make up the nodes of OAPTs. In this fashion, each node of an OAPT represents a specific execution algorithm for an algebra expression. Instead of defining T_algOp as an immediate subtype of T_function, we define it as a subtype of T_context which, in turn, is a subtype of T_function. In a sense, a T_context instance corresponds to delayed execution of a function. This allows further optimization possibilities. Since the nodes of OAPTs are instances of T_context (due to subtyping), we could relax the restriction that they be instances of T_algOp and represent the functions that implement behaviors in predicates of query expressions as nodes in the OAPT as well. This would open the possibility of optimizing the execution of the functions that implement behaviors together with algebraic operators in a query. Commonly called the *method optimization problem*, this is a serious concern in OBMSs.

Since the nodes of OAPTs are instances of type T_algOp, to provide uniformity, we define the object *F_leafAlgOp* to model the leaf nodes of the OAPTs. This object can be thought of as a container that holds a reference to one of the input collections of the query that the OAPT represents. *F_leafAlgOp* objects model the delayed execution of the algebraic identity operator *B_Identity* that is defined as part of the interface of the type T_collection. Thus, **all** the nodes of an OAPT are uniformly modeled as instances of T_algOp rather than making an exception for the leaf nodes which correspond to collections[4].

An OAPT is recursively defined as an object of type T_algOp as follows: the root node of the OAPT is an algebraic operator of type T_algOp whose children are also of type T_algOp with the restrictions that no interior node can be the object *F_leafAlgOp* and every leaf node is the object *F_leafAlgOp*.

Instead of discussing the behaviors defined on T_algOp (see [Mun93]), we present the general approach with an example. Consider a geo-information system application that stores information about dwellings, the zones that these dwellings are in and the maps of these areas. Four types, among many, are defined: T_map, T_person, T_dwelling, and T_zone. T_map is a subtype of T_displayObject which models all displayable objects; the other types are direct subtypes of the root type, T_object. Using, TQL, the query "Return the maps which show the areas where senior citizens live" is expressed as

```
select o
from o in C_map
where exists (
        select p
        from p in C_person, q in C_dwelling
        where (p.B_age() ≥ 65 and q = p.B_residence()
                  and q.B_inzone() ∈ o.B_zones()))
```

[4]This is a conceptual model; for efficiency reasons, the optimizer may represent the leaf nodes directly as collections and handle them as special cases.

where the types over which behaviors are defined can be determined from the range variables. The algebraic expression of the same query is

$$Result \leftarrow \mathbf{C\_map}_o\ \sigma_F < \mathbf{C\_person}_p,\ \mathbf{C\_dwelling}_q >$$

where σ indicates the selection algebraic operator, **C_map** is the target class of selection, $\mathbf{C\_Class}_i$ indicates that variable i ranges over class **C_Class**, and F is the selection formula:

$$F \equiv p.B\_age \geq 65\ \wedge q = p.B\_residence\ \wedge$$
$$q.B\_inzone \in o.B\_zones$$

Figure 3 shows the OAPT for this example. The first information in the box represents an object instance reference and the mapping to its type. Then the behaviors that are relevant to the subsequent discussion are listed. The f in the figure is the formula $p.B\_age \geq 65\ \wedge\ q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones$ and is represented as an object of type T_formula.

As indicated before, the search space consists of a family of equivalent plans, each of which is represented as an OAPT. The equivalence of two OAPTs corresponding to the same query is established by means of the equivalence-preserving algebraic transformation rules. In [SÖ90], we give transformation rules for a less powerful algebra than the one supported by TIGUKAT. These rules are being extended for the new algebra specification.

To provide extensibility, we model these transformation rules as objects of type T_algEqRule which is a subtype of T_rule. Even though at this stage we have not yet added "active DBMS" capabilities to TIGUKAT, we have defined the type T_rule as an abstract supertype with subtypes T_activeRule and T_algEqRule. T_activeRule would, in the future, model ECA-type rules [DBM88] when active capabilities are added.

Matching rules to OAPTs is analogous to unification in logic programming where a two step process is used. The first step is a syntactic match based on identical symbols. Only if the first step succeeds is the second more computationally expensive match attempted. In logic programming the second step is the identification of a unifying substitution. In this case, the match is also based on semantics.

The application of rules by rule-based optimizers such as Exodus [GD87] and Starburst [HCF+89] optimizers is done by a pattern matching engine that matches subexpressions of a query against algebraic rules. Additionally, the firing of rules is dependent on the satisfaction of the conditions that involve user-defined functions. A major difference between the rules defined for those systems and the ones defined for TIGUKAT is that the rules for the former systems are based on operators of fixed arity while the rules for TIGUKAT are based on algebraic operators which can have varying numbers of arguments. In order to use those rule-based optimizers for the application of rules defined herein, their pattern matching engine would have had to be modified to handle algebraic operators which can have varying numbers of arguments. Considering the uniform manner in which the TIGUKAT optimizer is defined as an extension of the object model, it would be difficult to integrate these optimizers with the TIGUKAT system. Hence, the need for a special pattern matching engine for the TIGUKAT optimizer.

6 Modeling of Search Strategies

The search strategy determines the use of the rules for controlling the search. There are many alternative strategies,

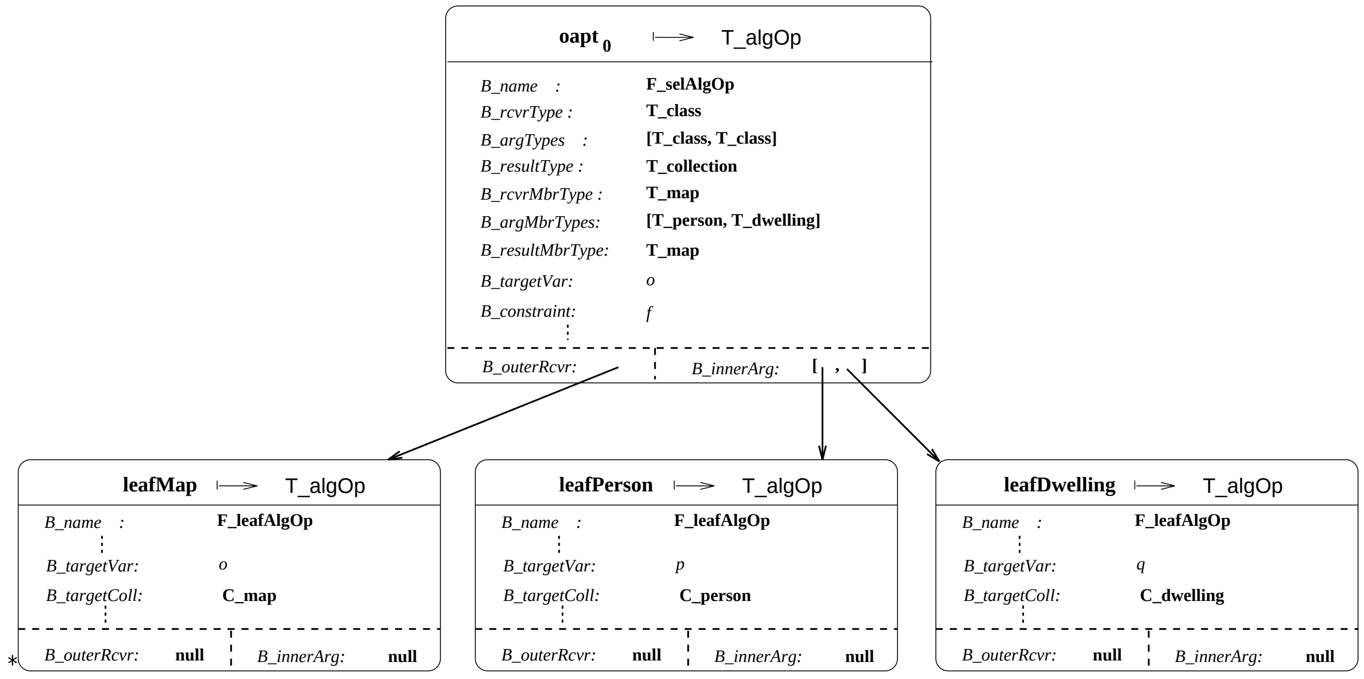

Figure 3: Construction of the OAPT

ranging from static priorities to heuristics that determine which rules would be applied under various conditions.

In traditional optimizers, the search space (usually identified as a set of processing trees) and the search strategies that control the movement through this search space are coupled together. However, in an extensible query optimizer, they need to be decoupled. Consequently, we define T_searchStrat as a subtype of type T_function which can in turn be specialized. The type T_searchStrat is an abstract supertype whose behaviors are implemented in its subtypes. In other words, its extent is empty. Figure 2 shows the specialization of T_searchStrat into enumerated search strategies, T_enumSS, randomized search strategies, T_randomSS, and heuristics-based optimization strategies, T_heurSS. The search strategy component of the query optimizer can easily be extended by further subtyping T_searchStrat when new search strategies are defined.

In this architecture, search strategies can be changed among queries. Before optimizing a query, the system selects the appropriate search strategy for the particular type of query under consideration. The flexibility of supporting various search strategies, each one best for a particular class of queries, enables the handling of different kinds of applications such as GIS, hypertext, and CAD/CAM by specializing T_query according to the different categories or classes of applications.

Since T_searchStrat is a subtype of T_function, it inherits all its behaviors and defines new ones. The native behaviors defined on T_searchStrat are *B_initSS* and *B_optimal* which return the initial OAPT(s) at which the search starts and the "optimal" OAPT which is reached at the end. Optimal OAPT is selected from a set of "good" candidate OAPTs which are returned by *B_goal*. The three behaviors, *B_nextState*, *B_stopCond*, and *B_action* control the execution of the search algorithm.

Since *B_searchStrat* is an abstract type, it does not implement any of these behaviors, leaving that to its subtypes (e.g., T_enumSS and T_randomSS). Through subtyping and the overloading of the behaviors defined on T_searchStrat, customization of the search strategy is possible. Due to space limitations, we cannot demonstrate the customization of search strategies with an example. We refer the reader to [Mun93].

7 Modeling of Cost Functions

The final optimization-related concept that needs to be incorporated into the model is the cost function. Cost-based optimization strategies apply a predetermined cost function (total time or response time) to an OAPT to calculate the cost of executing the corresponding query according to that OAPT. The issue is how these cost functions are modeled.

In TIGUKAT, each function is associated a cost through *B_costFunction*. Applying this behavior to a function object f returns another function object g of type T_costFunc that implements the computation of the cost of executing function f. T_costFunc is a subtype of T_function. When function g is executed, it returns the actual cost of executing function f. It is important to associate a cost function with each function instead of a cost value since the cost is often dependent on other factors that can be modeled as arguments to the cost function.

Algebraic operator context nodes (i.e., the instances of T_algOp) redefine the behavior *B_costFunction* to return a function object of type T_costFuncAlgOp, which is a subtype of T_costFunc. This redefinition is necessary since the cost of executing algebraic operators require the incorporation of various optimization issues into these functions. These issues are typical ones such as the availability of indexes over the collection on which these operators are defined, the statistical information about these collections, etc.

Calculation of the cost of executing one node of an OAPT is recursive in nature. The cost of a node is defined in terms of the costs of its children whose costs in turn depend on the costs of their own children. Recursion naturally terminates at the leaf nodes. Consequently, the application of a function of type `T_costFunc` to the root context of an OAPT calculates the estimated cost of executing that query according to the execution plan represented by that OAPT. Thus, our definition of cost functions are based on graphs. Given an OAPT, its total time is calculated by summing up the costs of all of its nodes; the calculation of the response time is dependent upon the shape of the OAPT (i.e., bushy trees vs. linear trees) and whether parallel execution is possible.

For an algebraic operator o, its cost function f includes the cost of executing algebraic operator o and the cost of executing the children of o (if any). The cost of executing the algebraic operator (i.e., the application of the behavior *B_costFunc* to the OAPT node o) may include the cost of object creation since some of the operators, such as product, create objects with (possibly) new types (we call these operators *target creating*). In these cases, the associated cost of creating a new type object and inserting it into the type lattice needs to be taken into account.

We model cost functions as instances of `T_costFunc`. Since `T_costFunc` is a subtype of `T_function`, it inherits all its behaviors. For a cost model function, *B_execute* computes the cost model equation whose algorithm is stored in *B_source*. For each different cost function, the sysem creates a different instance of `T_costFunc`. For example, for a total time cost model function, the instance *F_costTT* is created as an object of type `T_costFunc`. This gives to the cost function component of the optimizer the extensibility property required to incorporate into TIGUKAT query optimizer new cost model functions as they are found useful to measure the actions of a search strategy over the search space.

Each algebraic operator context node (of type `T_algOp`) has a different cost function because the algebraic node function incorporates optimization issues that potentially vary among the operators. For example, the union operator only needs the cost of accessing the instances of the collections C_i, C_j involved in the operation, while the select operator requires the cost of accessing the instances of the collection C, in the presence of a predicate f. This means that for each algebraic operation implementation (an instance of `T_function`), there is a corresponding cost function (an instance of `T_costFuncAlgOp`) that computes the cost of executing that operation.

Because OAPT nodes are objects of type `T_algOp` which is itself a subtype of `T_function`, they inherit the behavior *B_costFunction* from `T_function`. The application of this behavior to an OAPT node o returns the function object of type `T_costFuncAlgOp` that implements the computation of the cost of executing the algebraic operation that node o represents.

The fundamental advantage of this approach is that each algebraic operator node provides individualized behavior for its cost function. For example, if the cost for some algebraic operators is considered negligible, it is easy to make the cost functions associated with those algebraic nodes return a constant value (i.e., zero) without having to modify the implementation of the cost model function to consider these exceptions. This gives more flexibility to the optimizer to be able to extend the cost function component for new algebraic operators as they are incorporated in the object algebra.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer the extensibility inherent in object models. The optimizer basically implements a control strategy that associates a search strategy and a cost function to each query. The database administrator has the option of defining new cost functions and new search strategies or transformation functions for new classes of queries.

8 Putting it Together

TIGUKAT query optimizer is incorporated as a behavior *B_optimize* in the interface of type `T_query`. It is, therefore, modeled as an instance of type `T_behavior` (see Figure 2). This means that the query optimizer has the status of a *first-class* object in the model.

The query optimizer behavior, *B_optimize*, is responsible for applying a search strategy *B_searchStrat* to an initial OAPT, *B_initialOAPT*, in order to produce an "optimal" OAPT, *B_optimizedOAPT*, for a query object q. In case the search strategy is a cost-controlled strategy, the cost model *B_costModelFunc* is used to measure the effects of the optimizer actions. All these behaviors are defined in the interface of the type `T_query` (see Section 3).

The implementation for *B_optimize* is:

F_optimize(`T_query` **q**): `T_list<T_algOp>`
{ **return**(
(**q**.*B_getSearchSS*).*B_execute*(**q**.*B_getInitialOAPT*)) }

To demonstrate the utility of the architecture and the interaction of its components, we present the sequence of steps that would be followed in creating and optimizing the query object for the example TQL query given above. All TQL queries are either submitted from within a programming language (embedded TQL) or during a user session. In this section we will only consider the latter, since embedded TQL is not yet available. For queries submitted during a session, we have a simple session control language, called TIGUKAT Control Language (TCL), that controls the creation of the appropriate objects and interprets the optimization commands.

1. Query object creation. The TCL interpreter creates the query (say **q1**) and then sets the TQL statement of the query as the source of this query.

   q1 $\leftarrow$ **C_query**.*B_new*
   q1.*B_setSource*(TQL_statement)

   Note that in TIGUKAT, for every behavior *B_behavior* whose value can be changed by users, a pair of behaviors, *B_getBehavior*/*B_setBehavior* is defined.

2. Search strategy specification. The architecture allows the user (or the application submitting the query) to set the search strategy and the behaviors associated with it. In this case, we assume that the system defaults are used:

   q1.*B_setSearchStrat*(*F_enumSS*)

3. Cost model specification. The architecture allows the user (or the application submitting the query) to set the cost model. In this case, we assume that the system defaults are used:

   q1.*B_setCostModelFunc*(*F_costTT*)

   The cost model is only required when the search strategy is cost-controlled.

4. Compilation of the query object. The query is compiled by applying the behavior *B_compile* to query **q1**:

   q1.*B_compile*()

   The effect of applying the *B_compile* behavior is the following:

(a) Parsing and calculus-algebra translation

$$\begin{array}{lr}
\mathbf{oapto}.\mathit{B\_setName}(\mathbf{F\_selAlgOp}) & (1)\\
\mathbf{oapto}.\mathit{B\_setRcvrType}(\texttt{T\_class}) & (2)\\
\mathbf{oapto}.\mathit{B\_setArgTypes}([\texttt{T\_class},\texttt{T\_class}]) & (3)\\
\mathbf{oapto}.\mathit{B\_setResultType}(\texttt{T\_collection}) & (4)\\
\mathbf{oapto}.\mathit{B\_setRcvMbrType}(\texttt{T\_map}) & (5)\\
\mathbf{oapto}.\mathit{B\_setArgMbrTypes}(& \\
\qquad [\texttt{T\_person}, \texttt{T\_dwelling}]) & (6)\\
\mathbf{oapto}.\mathit{B\_setResultMbrType}(\texttt{T\_map}) & (7)\\
\mathbf{oapto}.\mathit{B\_setTargetVar}(o) & (8)\\
\mathbf{oapto}.\mathit{B\_setConstraint}(f) & (9)
\end{array}$$

$$\begin{array}{lr}
\mathbf{q1}.\mathit{B\_setInitialOAPT}(\mathbf{oapto}) & (10)\\
\mathbf{q1}.\mathit{B\_setResult}(\mathbf{null}) & (11)
\end{array}$$

The result of the translation of the calculus query into an algebra expression is the generation of an OAPT (referenced by **oapto** and setting various behaviors (expressions (1) – (9) above). By and large these expressions are self-explanatory. The ones that require some explanation are (9)–(11). The f in expression (9) is a reference to an object of type `T_formula` which represents the predicate of the selection operator. Statements (10) and (11) set the two behaviors of the query object **q1**. The result of the query is null at this point since it has not yet been executed.

(b) Algebraic Optimization

The second major action resulting from the compilation of a query is its optimization. This is accomplished by the application of *B_optimize* to **q1**:

$$\mathit{finalOAPT} \leftarrow \mathbf{q1}.\mathit{B\_optimize}()$$

The *B_optimize* behavior carries out plan optimization on **q1**.*B_initialOAPT* using the search strategy **q1**.*B_searchStrat*. This behavior implements the control strategy for the plan optimizer. Since the search strategy in this example is an enumerative search algorithm, the cost of the initial OAPT is calculated using the technique described in Section 7. The cost of all the equivalent OAPTs that can be obtained by the application of transformation rules are calculated in a similar manner and the one with the least cost is selected as the *finalOAPT*[5]. This optimal OAPT is saved:

$$\mathbf{q1}.\mathit{B\_setOptimizedOAPT}(\mathit{finalOAPT})$$

which records the optimal execution plan as part of the query. This is useful both for later executions which do not need to be optimized and for being able to implement operators such as "explain" which informs the requestor of the optimal execution plan that the optimizer has chosen. These operators are now quite common in state-of-the-art DBMSs.

(c) Execution Plan Generation

This is the last step in the query processing methodology whereby the algebraically optimized OAPT is submitted to the object manager for further optimization and execution. This part is outside the scope of the current research. However, our ideas about this step are described in [SÖ95]. Basically, we propose to generate the query execution plan by replacing each individual algebra operator from the optimized OAPT with a "best" subtree of Object Manager (OM) calls. These object manager calls that are part of the set of low level object manipulation primitives that constitutes the interface to the OM can be modeled in TIGUKAT as function objects.

The architecture supports execution plan generation by providing the behavior *B_genExecPlan*, which, when applied to **q1**:

$$\mathbf{q1}.\mathit{B\_genExecPlan}()$$

results in the set of execution plans being stored as part of the query. These execution plans can be accessed later by applying *B_execPlanFamily* to query **q1**.

5. Execution of the query object. The execution of the query may be invoked by the user explicitly if the query is already optimized. In the case of ad hoc queries submitted during a user session, the query is executed when it is compiled and optimized. Thus, the TCL interpreter applies the *B_execute* behavior to **q1** and uses the result of that application as an argument to *B_setResult*.

$$\mathbf{q1}.\mathit{B\_setResult}(\mathbf{q1}.\mathit{B\_execute}())$$

9 Conclusions

In this paper we describe an extensible query optimizer architecture for the TIGUKAT OBMS. Even though this research is conducted within the context of the TIGUKAT project, the general approach is applicable to other OBMS designs.

The identifying characteristic of our design is the use of the object-oriented philosophy in providing extensibility. The architecture defines all components of the optimizer (search space and transformation rules, cost function, and search strategies) as well as the queries themselves as first-class objects. This is consistent both with the TIGUKAT object model and the object-oriented design philosopy. In a sense, we are using the medicine that we normally prescribe to others. The end result, which we believe to be a significant advantage, is that both the query model and the query optimizer become direct extensions of the TIGUKAT object model which can be managed (stored, changed, queried) just like any other object.

The emphasis in this paper is on the extensible architecture of the optimizer. Therefore, we have not discussed details such as top-down versus bottom up evaluation of execution plans, logical and physical transformation rules, space management, etc. Some of these have been researched and will be presented in other papers and some of them are topics of on-going and future research.

The architecture described in this paper has been implemented. The implementation covers all the types required for the optimizer. There are a number of outstanding issues for the generation of a full-fledged optimizer. The most important is to couple this architecture with an optimizer generator that would provide a language for the specification of optimization alternatives. We intend to use a new version

[5]This is an oversimplification; no query optimizer enumerates all the alternatives, rather the search space is pruned using one of a number of algorithms. That complication is ignored in this example.

of Volcano optimizer generator that is currently under development. This version will be suitable for object-oriented systems and will allow more extensibility than the earlier versions of Volcano. Other issues we work on include the specification of the full set of logical and physical transformation rules and the physical optimization at the storage system level (similar to [SÖ95]).

References

[BBG+86] D.S. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twitchell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Eng.*, SE-1.12(11):1711–1.1730, November 1986.

[BG92] L. Becker and R.H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, June 1992.

[BMG93] J.A. Blakeley, W.J. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–296, 1993.

[CD86] Michael J. Carey and David DeWitt. The architecture of the EXODUS extensible DBMS. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, CA (USA), September 1986. IEEE.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database system. In *Proc. of the 2nd Int. Workshop on Object-Oriented Database Systems*, pages 129–1.143, 1988.

[Fre87] J.C. Freytag. A rule–based view of query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–1.180, 1987.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 160–1.172, May 1987.

[GM93] G. Graefe and W.J. McKenna. The Volcano optimizer generator. In *Proc. 9th Int. Conf. on Data Engineering*, pages 209–218, 1993.

[GT91] A.V. Gelder and R.W. Topor. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[HCF+89] L.M. Haas, W.F. Cody, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 377–388, 1989.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. 12th Int. Conf. on Very Large Databases*, pages 128,137, 1986.

[LV91] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. 17th Int. Conf. on Very Large Databases*, pages 363–373, 1991.

[MDZ93] G. Mitchell, U. Dayal, and S.B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *Proc. 19th Int. Conf. on Very Large Databases*, pages 517–528, August 1993.

[Mun93] A. Munoz. Extensible query optimizer architecture for TIGUKAT. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report TR94–01.

[MZD92] G. Mitchell, S.B. Zdonik, and U. 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.

[ÖB94] M.T. Özsu and J. Blakeley. Query processing in object-oriented database systems. In W. Kim, editor, *Modern Database Management — Issues in Object-Oriented and Multidatabase Technologies*. Addison-Wesley/ACM Press, 1994, pages 146–1.174.

[ÖPI+95] M.T. Özsu, R.G. Peters, B. Irani, A. Lipka, A. Munoz, and D. Szafron. TIGUKAT: A uniform behavioral objectbase management system. *The VLDB Journal*, 1995. In press.

[PLÖS93a] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An extensible query model and its languages for a uniform behavioral object management system. In *Proc. 2nd Int. Conf. on Information and Knowledge Management*, pages 403–412, November 1993.

[PLÖS93b] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The query model and query language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, January 1993.

[PÖ93] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. 12th Int. Conf. on Entity-Relationship Approach*, pages 37–49, December 1993.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres next generation database management system. *Comm. of the ACM*, 34(10):78–92, October 1991.

[SÖ90] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ95] D.D. Straube and M.T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Eng.*, 7(2):210–227, April 1995.

[SRL+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third-generation data base system manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.