An Architecture for Query Processing in Persistent Object Stores

Gail Mitchell[1]
Stanley B. Zdonik[2]
Umeshwar Dayal[3]

Technical Report No. CS-91-38

June 1991

[1]Brown University, Department of Computer Science Box 1910, Providence, RI 02912
[2]Brown University, Department of Computer Science Box 1910, Providence, RI 02912
[3]Cambridge Research Lab, Digital Equipment Corporation, Cambridge, MA

An Architecture for Query Processing in Persistent Object Stores[†]

Gail Mitchell[‡]and Stanley B. Zdonik[§]
Department of Computer Science
Brown University
Providence, RI 02912 USA
{gms,sbz}@cs.brown.edu

Umeshwar Dayal
Cambridge Research Lab
Digital Equipment Corporation
Cambridge, MA 02139 USA
dayal@crl.dec.com

Abstract

Query optimizers for persistent object systems should be extensible to react to user-supplied abstract types. Current architectures support only a single, non-extensible technique for controlling the optimization process. We propose an alternative to the current extensible architectures that will support multiple optimizer control strategies and the addition of new control strategies. The optimizer consists of a collection of optimization regions, each of which can transform queries according to a particular control strategy, set of transformations and cost model. A global optimizer control coordinates the movement of a query between these regions. This architecture provides extensibility in the optimizer's repertoire of control strategies through the addition of new regions.

In this paper we describe our approach and demonstrate its utility by following the optimizer as it works on an example query. The optimizer will move the query between three distinct regions. The different regions illustrate different kinds of transformations and different strategies for application of those transformations. The optimizer can be extended with additional regions to provide more transformation types and control strategies; a result that can not be achieved by existing relational or extensible optimizers.

1 Introduction

In practice, persistent object systems are used to collect large numbers of objects for very long periods of time. Naming objects in these persistent stores is an important issue. One approach is to use a hierarchically organized name space that associates a single structured name with a single object. An alternative approach is to associate a name, in the form of a query, with a collection of objects. The binding of the query to the objects that it names will vary over time as the properties of the objects in the persistent store change. This introduces the concept of associative naming into our language.

The optimization of queries is a crucial factor in providing efficient evaluation of these associative names. A query describes information one wants to retrieve. It is normally written in some language

[†]This is a revised version (September 1991) of the report initially published in June 1991. This revised version also will appear in the Proceedings of the Hawaii International Conference on System Sciences, January 1992

[‡]This work was partially supported by a gift from Texas Instruments and by the Office of Naval Research and the Defense Advanced Research Projects Agency under Contract N00014-91-J-4052.

[§]This work was partially supported by the Office of Naval Research and the Defense Advanced Research Projects Agency under contracts N00014-91-J-4052 and N00014-83-K-0146 and ARPA order 6320, amendment 1 and by Digital Equipment Corporation under Research Agreement number 686.

and translated, by a query optimizer, into a series of operations against the persistent object store to yield the desired information. The goal of query optimization is to produce an efficient execution plan for the query. Usually, there are many alternatives from which to choose. The set of query operations is typically not minimal, so there can be many alternative expressions equivalent to the original query expression. Also, each operator may have several alternative implementations (e.g. a join operator, which pairs matching objects from two collections, may be implemented by a nested iteration method, by a sort-merge method, or by using an index), and there may be alternative access paths to the data (e.g. file scans, indexes or hash keys). Typically, the optimizer explores the space of alternatives using a controlled search strategy and a cost model for evaluating the costs of alternatives. A query optimizer searches for a good execution plan for processing a query, although it may not necessarily discover the most efficient plan.

The ability to do automatic query optimization is one of the strengths of relational database systems. Relational query optimizers exploit the fixed semantics of the model and fixed sets of operators, storage structures, and implementation methods for the operators. Although different relational optimizers perform basically the same tasks, the designs of such optimizers differ and are specific to the system in which the optimizer is built. These optimizers generally embody a set of pre-defined manipulations on some internal query representation. These manipulations apply built-in heuristics to develop efficient strategies for data access. The access strategies are evaluated according to some cost formula, and a best strategy is selected. The heuristics that are applied, the algorithms for searching for strategies, and the cost model upon which strategy evaluation is based are all fixed and specific to the particular optimizer.

In this paper we will study the query optimization problem in the context of an object-oriented type system. Thus, we will use the ENCORE database system as our example. ENCORE, as well as most object-oriented databases [2, 19, 23, 24, 34], supports (among other features) abstract data types, type inheritance, and object identity. It provides the ability to extend the data model through abstract data types and subtyping. The extensibility of a model like ENCORE raises interesting problems in optimization that we will discuss later.

The application of algebraic rewrite rules has formed the basis for the design of query optimizers for object-oriented databases [11, 25, 31]. A recognized difficulty in applying transformations is the problem of manipulating query expressions containing arbitrary methods. Object-oriented query languages are built from operations defined on built-in bulk types (such as Set and Tuple) and on abstract data types, and can, therefore, access arbitrary methods defined for the types. The costs associated with the application of methods need to be considered when manipulating query expressions to determine database access strategies. We have also found that manipulations involving objects with identity complicate the definition of the equivalence of two query expressions [28] thus complicating the application of transformation rules.

Optimizers for persistent object systems must be extensible because the relevant optimization techniques vary significantly for the various extensions. Also, it seems at this point that the problems introduced by the complexities of the model will require multiple strategies for applying transformations.

Current extensible optimizers have concentrated on accommodating extensibility of the data model, new algebraic operators, new transformation rules, and new data access strategies[8, 9, 13, 26]. They are generally based on rewrite rules for a set of operators defined on the bulk types. These rules are applied to a query expression to generate equivalent, but hopefully more efficient, forms of the expression. The transformed expressions can then be evaluated according to a cost model to select an executable plan for database access. The extensibility in these optimizers derives in large part from the ability to define the set of algebraic rewrite rules that can be applied to query expressions. The quality of the result depends on the completeness of the defined set of rules, as well as the cost model. The efficiency of the optimizer itself depends on the control strategies for selecting which rules to apply, and when to apply them.

Extensible optimizers typically build in a single control strategy that determines the order in which the space of equivalent queries is searched. Often, heuristics are used to guide the search, but these are typically written into the code of the optimizer. They are, therefore, impossible to extend or change without a major rewrite of the system. Optimizer generators can partially address this problem by allowing a control strategy to be defined for an optimizer at the time it is generated[26]. Lanzelotte and Valduriez extend this idea by defining a number of search control strategies in the optimizer and choosing a strategy depending upon the query expression being optimized[17]. The architecture we present in this paper allows the definition of any number of control strategies in the optimizer and supports the use of possibly many different control strategies during the processing of a single input query.

We advocate a new, extensible architecture that encompasses extensions to the collection of control strategies and optimization techniques embodied in the optimizer. This extensibility is achieved by organizing the transformation process such that query transformation is performed by a set of concurrently available modules, each with its own control. Each module obeys a standard interface so that the modules can be combined in different ways to process different queries. The standard interface also supports the addition of new modules to the optimizer. These modules can represent new strategies for query optimization and thus can support extensions to the data model and its associated operations and access methods.

In this paper we give a preliminary description of this architecture and demonstrate its use with an extended query optimization example. Section 2 of this paper provides the background required to understand the rest of the paper. It describes the data model and the algebra that we use, some optimization problems that are encountered, and some current, related work in the optimization of queries for object bases. Section 3 describes the architecture for our optimizer. A detailed specification of this architecture is beyond the scope of this paper. Section 4 shows how this optimizer could be used to process a complex query, and Section 5 summarizes our results and thoughts about this problem.

2 Background

2.1 A data model and algebra

Query optimizers and optimization techniques are usually based on a particular data model and query language for that model. The examples in this paper are based on problems encountered when trying to optimize queries in the ENCORE model with the EQUAL algebra. The data model and algebra are part of a prototype database system under development at Brown University. In this section we briefly describe those features of the model and algebra necessary to understand the discussion in the following sections. This is not a full description of either ENCORE or EQUAL. More details can be found in [29] and [32].

ENCORE is based strongly on abstract data types [21]. All types are defined by their interface which is specified in terms of a set of method signatures. A signature for method M is a name plus an ordered list of types that correspond to the legal types for the arguments of M. Objects can be related to each other by means of these methods. The ENCORE type system has been designed to allow for static type checking. Type equivalence is determined by name equivalence.

ENCORE defines a set of atomic types, **Integer**, **Real**, **Bool**, and **String**. These types define the only values in the system. A value is something which is guaranteed to be immutable. ENCORE does not allow users to define new value types. All user-defined types describe objects. The basic type constructor in ENCORE is the abstract data type, in that it takes a type for its representation (concrete type) and generates a new type as the abstract type. Parameterized types also provide a more standard type construction mechanism.

A parameterized type is a mechanism for specifying a family of related types with one textual definition. The parameters for a parameterized type may include other types. In order to retain the ability to do static type checking, the parameters are restricted to be expressions which are statically typed. A parameterized type is like a metatype in that it is a generator for other types. An instance of a parameterized type is a type. For example, **Set[Integer]** is an instance of **Set[T: Type]**.

ENCORE defines two common parameterized types, **Set[T]** and **Tuple**[$\mathbf{a}_1$:$\mathbf{T}_1$, ...,$\mathbf{a}_n$:$\mathbf{T}_n$]. These two types play an important role in the query facility, where they are used to construct types for query results. When the parameterized type **Set[T]** is given a value such as **Integer** for the parameter **T**, it generates a new type **Set[Integer]**. The name of this type is "Set[Integer]". The type **Set[Integer]** has instances that are sets whose members are constrained to be integers. There can be multiple instances of a type **Set** containing exactly the same members.

An instance x of a type **T** has a unique identity that is independent of the state of x. Although an object's identity may be implemented by a system supplied object identifier, the existence of the identifier is transparent to the language interface. Logically, the language always sees an object, and must access that object through the interface defined for its type. Object identity allows the implementation of references between objects. When an object x refers to an object y by means of a method M (i.e., M(x)=y), applying M to x produces object y (and does **not** give access to an identifier for y).

Properties in ENCORE reflect the abstract state of an object. The notion of *property* is modelled by one or more of the methods defined on a type. A property value is accessed by a special observer method which is required to have no side-effects on the observable state of the object. This method for property P is called Get_P; we denote Get_P(x) as x.P. A property P may also support another function called Set_P that allows the value of P to be changed.

The Get_P method can return the value of a stored field or it may perform a more sophisticated computation based on the stored representation of the object. Thus, properties describe a complex *logical* structure for objects.

The ENCORE Query Algebra (EQUAL) is a collection of operators, defined for parameterized type **Set**, that can be used to construct queries over collections of ENCORE objects. The operators support abstract data types and encapsulation by accessing objects only through the methods defined for their type, in particular their properties. The result of a query is a new database object of type **Set[T]**. Each object built by a query is a new object with a unique identity.

EQUAL includes operations that are the analogs of relational algebraic operations as well as operations to manipulate the logical structure of ENCORE objects. The Select, Project and Ojoin (object join) operations are similar to their relational counterparts in meaning, but generalize the relational operations by applying functions (i.e. property methods or query operations) to the database objects. For example, the query $Project(Vehicles, \lambda v\ [V : v, Owners : Select(People, \lambda p\ v \in p.cars)])$ builds a set of tuples each of which contains a **Vehicle** object (V) and a **Set[Person]** object (Owners).[1] For each v in the Vehicles set, a tuple is created with a V attribute whose value is v and a Owners attribute whose value is built by applying the Select method to the People set object (note that it is possible for the Owners attribute to be an empty set).

The application of a Get_property method to an object x is a navigation from x to the object representing the value of the property. A navigation can be a single property retrieval, or a string of retrievals (e.g. x.mother.mother.cars to find one's grandmother's cars). The latter is commonly called a "path expression" and is analogous to a join between sets of objects. We also provide an

[1] We assume here that People has type **Set[Person]** and that type **Person** has a *cars* property that returns an object of type **Set[Vehicle]**. The λp is simply a way to generate a bound variable that will range over the members of People.

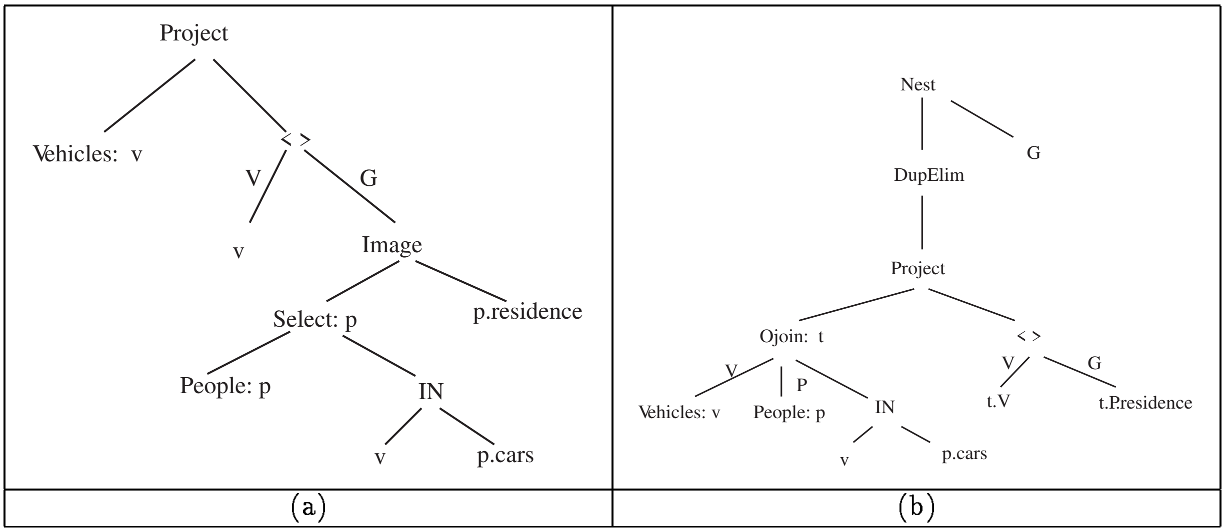

Figure 1: Query – Find all possible garaging locations for each vehicle.

Ojoin operation to explicitly join two sets of objects that are not necessarily related navigationally. For example, an Ojoin of the set People with itself can be used to find pairs of people who share the same address (i.e. Ojoin(People, People, Person, Roommate, $\lambda p_1 \lambda p_2\ p_1 \neq p_2 \ \wedge\ p_1$.residence $= p_2$.residence)[2]). Ojoin is analogous to relational Theta-join, with sets of objects of a non-tuple type considered as sets of 1-tuples. As a result, the Ojoin operation is associative.

Other operations that retrieve information are Image, Union, Intersection and Difference. Image is like a LISP mapcar; it allows the application of a function (which again may be a property or an EQUAL method) to each object in a set, collecting the results in a set object. The Union, Intersection and Difference operations are like the relational operations, but also account for subtyping in determining result type.

EQUAL also includes operations that manipulate the structure and identity of objects. The Nest, UnNest and Flatten operations work only with the structure of objects. DupEliminate and Coalesce manipulate identities. DupEliminate is necessary because we often build new tuple objects that may contain the same attribute values. The Coalesce operator is useful when we build new objects in a subquery. If two queries are executed, the results will be two distinct objects even though those objects may represent the same values. For example, suppose in the Project query matching vehicles and their owners that two people are co-owners of the same two vehicles. The Select subquery will build a new set object for each vehicle, with both sets containing the same two owners. A Coalesce operation could be used to ensure that both Owners attributes reference exactly the same set of people.

2.2 A problem

Although many of the problems that need to be solved by an optimizer for a persistent object system are similar to problems addressed by relational and extensible database optimizers, there are also many problems that are unique to the object model. These problems arise as a result of supporting objects having abstract data types with arbitrary methods, complex structures and object identity. In general, many of the assumptions about semantics and data that can be made in

[2]The result of an Ojoin is a tuple type. The first two parameters of Ojoin are input sets, the next two are optional and name output attributes. The last parameter is a Boolean function over the pairs of input objects

more traditional models no longer hold when supporting the dynamics of the object model. In this section we discuss one important problem that arises when trying to support the complex structure of objects. We show how our architecture can handle this problem in Section 4.

The complex structure of objects means that languages to query the objects have mechanisms for exploring their structure. In addition, languages which allow the creation of objects need mechanisms for building new structures. The exploration and creation of such structures can lead to the regular use of path expressions to navigate through a structure. Current research on the optimization of path expressions involves the definition of indices for an expression [4, 22] and the use of Join operations to effect path navigation [5, 14, 16, 18]. Such research leads to new techniques for the optimization of query expressions.

The exploration of complex structures also can lead to nested query expressions in which variables from outer expressions are referenced in nested expressions. For example, a query to find the garaging locations for all Vehicles could be expressed as follows:

$$Project(Vehicles, \lambda v[V : v, G : Image(Select(People, \lambda p\ v \in p.cars), \lambda p\ p.residence)])$$

This query is illustrated by the tree in Figure 1a. This expression satisfies the query by building a structure that stores a set of addresses with each vehicle. Note that, in this query, the variable representing a Vehicle (v) is referenced in the nested Select query.

A problem with nested query expressions, that we will explore in our example in Section 4, is that query transformations are not necessarily local transformations since they can involve variables and query operators at different nesting levels. Variables and query operations can also be nested in predicates as well as in input parameters for an operator. Such situations are not handled by the local, algebraic transformations that are used by rule-based optimizers.

For example in the query of Figure 1a the $v \in p.cars$ predicate indicates that there is a relationship between Vehicles (represented by variable v) and People (represented by variable p) that could be effected with a Join operation. The query expression illustrated in Figure 1b also satisfies the query.[3] However, the two query expressions differ significantly in their use of variables. In the first query, variable reference is nested, while in the second query all variables are local to the operation using them. The second query is similar to relational queries in which a Join of all relations assimilates the data, then a Project operation builds the result relation. In this query, the data is basically collected using the Join operation, then the structure for the result is built with the Project, DupElim and Nest operations.

The arbitrary nesting of query expressions means that recognition of join relationships involves global knowledge about the expression. As a result, the transformation between the Project and Join versions of a query expression will require more than the application of context-free algebraic rewrite rules. In addition, any transformations may have to preserve the structure of result objects. In the query of Figure 1 the transformation required the addition of an operation to eliminate duplication (DupElim) and another to reformat the result (Nest). Producing such structures requires knowledge which includes the effects of all parts of the query expression on the result structure.

Rule-based optimizers handle only local algebraic rewriting of query expressions. As a result, they will not perform transformations from, for example, the (a) to (b) query in Figure 1. Also, since rewrite rules are defined over the algebraic operations, they do not easily capture optimization for the predicates that are parameters for these operations. For example, rule-based optimization of a query such as $Select(People, \lambda p\ p.residence.city = \text{“}Boston\text{”})$ involves pre- and post-conditions on a rule to recognize and deal with the path expression in the predicate.

[3] The queries are slightly different, although they could be easily made the same. The Project in Query a does a left outerjoin – vehicles that don't match with any object in People are still included in the result. The usual semantics of Ojoin (query b) preclude unmatched vehicles.

An optimizer should be able to generate, and work with, both the Project and Join query representations. The Join representation of the query might result, for example, from a mechanical translation from an SQL-like query. In this case we might want to be able to translate to the Project version of the query to explore optimizations that we cannot recognize from the Join version. On the other hand, if the query is initially the Project expression we might want to generate the Join query if there are efficient techniques for dealing with Joins.

2.3 Previous work

Optimization strategies and results in relational and extensible database systems form the basis for current research in query optimization for persistent object systems. The Gemstone database, for example, focuses on indexing for processing of queries that select from a class based on a predicate [22]. A variety of proposals have been made for optimization based on algebraic transformation rules for query operations[3, 20, 25, 31, 33]. For example, Straube and Ozsu [30] propose a methodology for query optimization that includes calculus-based optimization, the transformation of calculus-based queries to algebraic form, type-checking of algebraic expressions, the application of algebraic transformation axioms, and the generation of access plans for the algebraic operations. Beeri and Kornatzky [3] assume there exists a rule-based optimizer and provide an extensive set of rules for bulk data types.

Current research on optimization in Orion [14] is directed toward the adaptation of relational techniques to the object-oriented database. In particular they explore efficient alternatives to object navigation. Research on optimization in O_2 [5] combines this work with techniques that include the factorization of common subexpressions, cost-based application of query rewrite rules, and the use of indexes and clustering to guide the query rewrite process and aid in determining access plans. These techniques are concerned, in particular, with accessing complex structures and the processing of nested query expressions. Lanzelotte et al. also explore object navigation[18]. They view path expressions as a series of join operations and apply rewrite rules to a tree of join operators.

Optimization in the presence of encapsulation and methods is a recognized problem that has not been solved [1]. Graefe and Ward propose to statically generate query evaluation plans with alternatives [12]. Information available about objects is used at execution time to choose among the alternatives and generate a final evaluation plan. Graefe and Maier submitted a preliminary architecture for an optimizer that sends messages to methods to ask them to "reveal" information about their execution [11]. The revealed information is used to expand the nodes of a query tree with execution information. The query tree, when fully expanded, will be input to a rule-based optimizer such as an EXODUS optimizer[10]. Kemper, Kelger and Moerkotte explored the materialization of function results and the requirements for updating these results as an object base is modified [15]. They propose to incorporate the materialized functions into a rule-based optimizer.

Each area of research attacks a specific problem in object-based query optimization and proposes a new technique for solving that problem. In the next section we describe an architecture that will allow the incorporation of a variety of different techniques as well as the addition of new optimization techniques.

3 An extensible architecture

An optimizer for an persistent object base must incorporate a number of different techniques for query manipulation and optimization. The techniques might differ in the classes of queries they can handle, the space of alternative query expressions that they consider, the goal of optimization (e.g., minimizing a cost function or producing a desired query form), and the control strategy used. For instance, one optimization technique might be to reorder join operations to find the

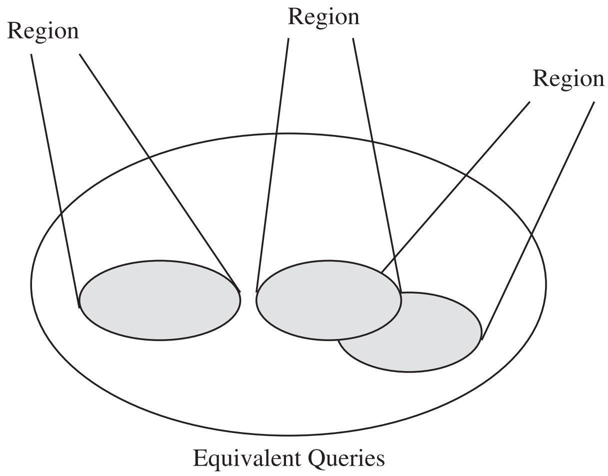

Figure 2: A region-based approach.

cheapest permutation of joins; and to arrange selections and projections as early as possible to reduce the sizes of join operands. Join orders might be enumerated using dynamic programming, and their costs evaluated using a cost model. This is the optimization technique commonly used by relational optimizers[27]. This optimization technique might be used with the query form in Figure 1b. However, it is inapplicable to the query form in Figure 1a.

A second technique might be to apply rewrite rules for a broad set of operations (not just joins, selections, and projections) to improve a query. The optimizer control might use a hill climbing or branch and bound strategy, stopping when the improvements drop below some threshold. This is the technique used by optimizers generated by the optimizer generator of [9], and it might be used with the query form in Figure 1a. Yet other techniques might be specialized to particular data types (e.g., spatial data), to particular operations (e.g., outerjoins), or to particular computing environments (e.g., parallel machines or wide-area networks). We expect the exploration of query optimization for object bases to result in new optimization techniques, and thus we believe that an optimizer must also be able to incorporate new techniques as they are developed.

Optimization of a query Q is inherently a process of searching the space of all query expressions that are equivalent to Q. Typically, a given optimizer can only visit some portion of this space, since the transformation rules are usually incomplete and the control strategies limit the search. The extensible nature of the object-oriented approach requires that we be able to expand the set of reachable expressions. We must also be able to carefully control the expanded search so that the performance of the optimization algorithm is not compromised.

We approach the control problem with a modular architecture in which each module incorporates one technique for query optimization. This approach is depicted in Figure 2. The optimizer is a collection of modules (we call them *regions*), each of which can manipulate a query expression to discover some subset of the possible equivalent expressions. The regions can differ in the queries they can manipulate, in the manipulations they can perform (i.e. the control strategy within the region), and in their goals for query manipulation. The extensibility in this optimizer architecture is founded on the ability to add new regions and, thus, incorporate new techniques for query optimization.

The regions do not necessarily partition the search for equivalent queries, and thus the inter-region (global) control may have to determine which of a set of applicable regions should work on a

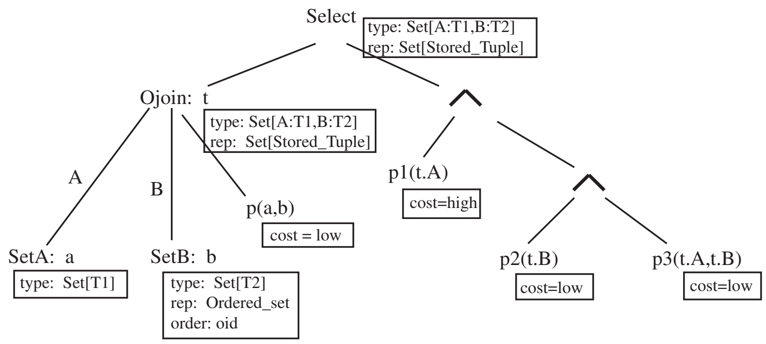

Figure 3: An annotated query tree

query at any point. Although we have not yet decided on a global control, the idea of such control is similar to that of searching for a transformation to apply to a query. Thus, current research in extensible search strategies may offer insights here[17, 26].

A query in our optimizer is represented as a tree which is annotated with information about the query operations and objects. The nodes of a tree are either objects (variables or constants) or methods on objects (functions). The methods are any methods that can be performed on objects and thus can represent algebraic operations (i.e. EQUAL query operators), relational operators (as in query predicates) or arbitrary methods defined for an object type. The annotations are used to capture cost-related information about a query and the objects a query accesses and builds.

Nodes are annotated with information that is useful for applying transformation rules. For example, we could annotate a node representing a set object with information about storage structures or access paths to the object. It might be useful to know whether there are indices on the set, or if the set is ordered in any way. This is the case in Figure 3 where SetB is ordered by an internal identifier (*rep*resentation is ordered_set and *order* is oid). Index or ordering information could be used when deciding to push a Select operation past a Join, for example. If a Select operation involves only one set in the Join, and the predicate refers to information that is indexed in the set, the Select past Join transformation could prove optimal.

A node representing a method could be annotated with the estimated cost of applying the method and with information about objects built by the method. The cost of applying a method could help in assessing the cost of applying a predicate using that method. The query operations themselves are also methods, on set types, so cost estimates for those nodes would be useful in assessing the utility of a transformation. For example, in Figure 3 the cost of evaluating predicate p_1 could be useful in determining whether to evaluate p_1 on SetA before joining. Method nodes can also be annotated with information about objects built by the method. For example, the Ojoin method in Figure 3 is annotated to note that the output is a set of stored 2-tuples. Such information could be used in assessing the cost of parent nodes.

We can identify annotations that would be useful in applying query transformations. Information about storage (such as indices, clustering, ordering, and object representation), operator implementations, object sizes, and method application costs can help in assessing the utility of transformations. Type information can also help in determining the applicability of a transformation. We envision the collection of annotations as being extensible. An extended set of annotations might be used by a new region or a new cost model, for example.

Region architecture: A region incorporates a particular strategy for query transformation. For example, one region could be responsible for using algebraic rewrite rules to manipulate general queries while another region optimizes only Select queries with no nested query operations. One region might work on arbitrary query trees while another region might manipulate queries in a particular canonical form. The kinds of queries manipulated in a region, and the kinds of manipulations performed in a region, define the semantics of that region.

For example, consider the query of Figure 1. The Ojoin version of the query (query b) represents a canonical form for queries in which data manipulation (e.g. Ojoin, Select, Image) is done first, followed by extraction of the result (Project) then formatting of the result type (UnNest, DupElim, Nest). This form could support optimization techniques for join operations and is the basis for a region in our optimizer. We are exploring the similarities between this form and query representation in OODAPLEX [7] with the goal of using OODAPLEX transformation results in such a region.

A region is defined by

- a predicate that characterizes the set of possible input query trees that can be accepted by the region (i.e., a canonical query form for the region),
- a collection of transformations that can be performed on those trees,
- a means of control over the application of those transformations, and
- a goal predicate that characterizes the output trees produced by the region.

The region control manages the rewriting of queries and must include a way to determine which transformations to apply and a means to determine how much transforming to do before relinquishing control to the optimizer. A region may include a cost model or a set of heuristics for assessing the value of a transformation. Different optimizer regions may have different goals for their results, e.g., the cheapest query that can be found in a given amount of time, a query with the fewest algebraic operations, or a query whose result is a set ordered in some interesting way.

Regions do not necessarily partition the space of query expressions – they may overlap in their sets of input trees or transformations. The regions do, however, provide the possibility of segregating sets of transformations about which there is some common characteristic. For example, if bottom-up tree search is found to be better for Join queries, but top-down is better for manipulating query predicates, then each of those strategies could be incorporated in a different region.

Regions also provide the ability to extend the optimizer with new techniques for optimization. A new technique, or strategy, is represented by a new region that can be added to the optimizer. The addition of new regions is supported by a standardized region interface. A region would normally be registered for use by the global control, although could also be registered for use by another region.

We do not define the form of transformations, but define only the behavior; a transformation takes a query expression to another query expression. Notice, then, that regions are themselves transformations in that they accept an input query and produce an equivalent output query expression. The regions can be nested to arbitrarily levels, forming a hierarchy as shown in Figure 4. At the root of the hierarchy is the global optimization region that incorporates control for coordinating the regions below it. The global control might initially send the query to one or more regions; then decide to move the query from one region to another, until finally an acceptable "optimal" query form is produced. A region at any internal node of the hierarchy might itself coordinate other subordinate regions. Finally, at the bottom are degenerate regions that encompass individual transformations and binary control that decides simply whether to apply the transformation or not.

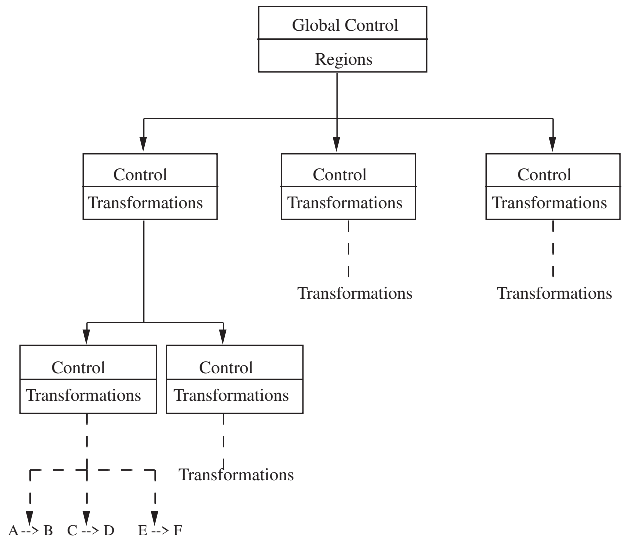

Figure 4: Optimizer architecture.

Separating the optimizer into regions results in three major advantages. The number of transformations that have to be considered by a region can be smaller, which could result in faster searches for transformations within the region. Also, different regions can use different control strategies for searching and applying transformations within the region. This could also result in more efficient application of transformations. Finally, the modularization of the transformation process into regions, each of which has a standard interface, makes extensibility of the optimizer control possible. Such extensibility supports the extensibility of the object model by providing for the incorporation of new techniques for query optimization which might best be controlled in different ways.

4 Extended example

To illustrate our optimizer architecture we will follow an example query through the optimization process. For the purposes of this example, we assume that the optimizer has the following regions:

1. Cost-guided — Uses syntactic transformation rules to manipulate arbitrary query trees. Application of the rules is directed by cost information about the operations and data involved in the query.

2. Join conversion — Transforms an arbitrary query to a query in a canonical join form in which join operations are grouped at the bottom of the query tree (if such a transformation is possible).

3. Join reordering — Manipulates queries that are expressed in terms of Join operations to determine good join orderings and access methods.

We will assume that these are the only regions in our optimizer, although a full optimizer would be expected to have regions handling a large variety of problems encountered in object queries (e.g. path expressions or semantic optimizations).

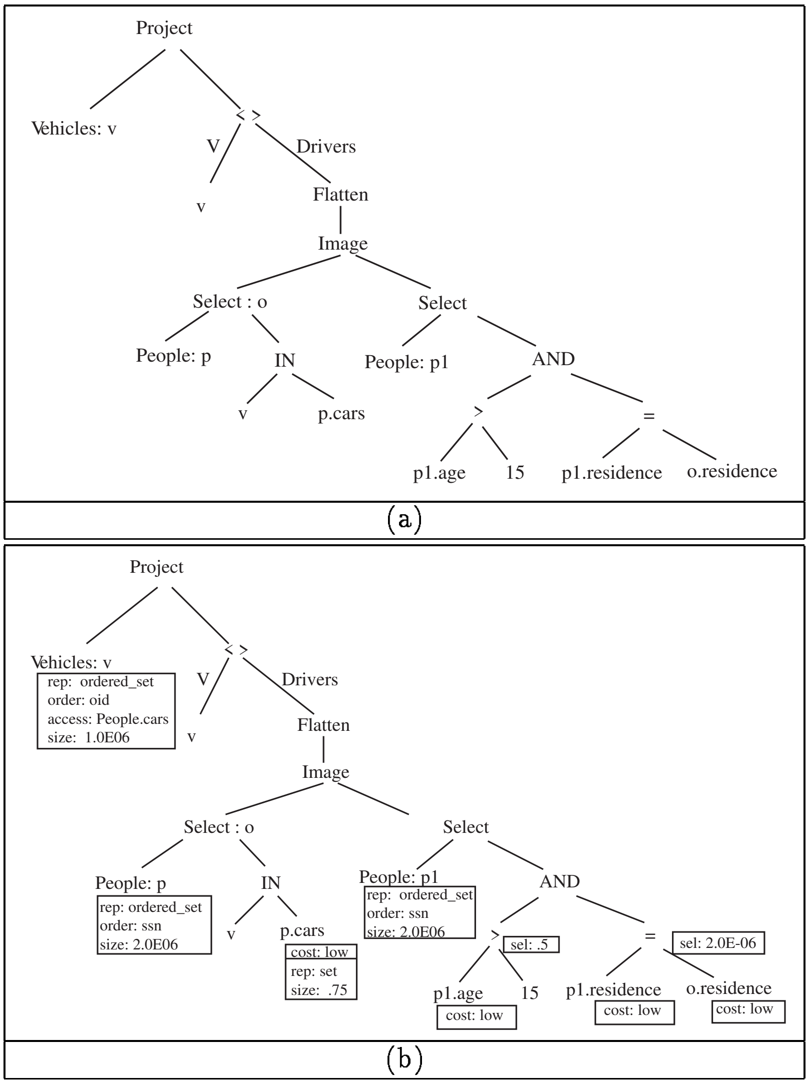

Figure 5: Query – List all possible drivers of each vehicle.

The regions in this example each represent a different strategy for query manipulation. The cost-guided region works similarly to existing rule-based optimizers that choose and apply transformations that are expected to reduce the cost of query execution. We do not include rules about join reordering here, since we feel that such transformations are better addressed by the join reordering region which uses a dynamic programming algorithm to manipulate join orderings. The join conversion region simply performs a query transformation. It is not concerned with improvement in the cost of query execution and may not even be able to discover any join relationships in a query.

The query we will manipulate is the following:

> For each vehicle, list all possible drivers. Anyone who is over the age of 15 and lives in the same house as an owner of the car is a possible driver.[4]

This query can be expressed algebraically as

[4] We call this the "insurance query" since, in Massachusetts, drivers who live with a car's owner may not be covered by the owner's insurance policy unless they are listed on that policy.

$$\begin{aligned}&Project(Vehicles, \lambda v[V: v, Drivers:\\&\quad Flatten(Image(Select(People, \lambda p\ v \in p.cars), \lambda o\ Select(People, \lambda p_1\ p_1.age > 15\\&\qquad\qquad\qquad\qquad\qquad\qquad \wedge p_1.residence = o.residence)))\\&\qquad\qquad])\end{aligned}$$

The result requested by this query expression has type **Set[Tuple[V:Vehicle,Drivers:Set[Person]]]**, i.e. each Vehicle is matched with a set of people who are considered to be drivers of the vehicle. To simplify the explanation of the problem, we will assume that each vehicle has at least one owner. This means that the results will never involve empty sets of Drivers. This simplification means that we do not have to deal here with the outerjoin semantics of Project. Solutions to problems dealing with outerjoins can be found elsewhere (e.g.[6]).

We assume that this query enters the optimizer as the tree in Figure 5a. The optimizer first annotates the tree with cost information such as ordering, indices and other storage access structures, sizes of set objects (input and output), execution cost of methods, and selectivity of predicates (Figure 5b). For this example we assume there are one million vehicles, two million people, and one half million residences. This information is reflected in the **size** annotations and in the **sel**ectivity of the residence comparison (there are four people per residence on average). We will also assume that half of the people are over the age of 15 (**sel**ectivity of $p_1.age > 15 = .5$), that half of the cars are jointly owned by two people and half of the cars are singly-owned (reflected in the average **size** of the output of p.cars). We assume that the *residence*, *cars* and *age* methods of the **Person** type all have low execution costs. This is indicated in the **cost** annotations for those methods. The People set is **order**ed by SSN (a user id number) and the Vehicles set is ordered by OID (internal object identifier). The Vehicles set can be accessed through the People set (i.e. p.cars), as indicated in the **access** annotation.

The annotated query is sent by the global controller to the cost-guided transformation region where the cost information can be used to assist in choosing the transformations to apply to the query expression. This region is using a control strategy similar to hill-climbing where tranformations are chosen that are expected to improve the expected execution cost of the query expression. The following transformation is applied to the query to yield the result in Figure 6a[5]:

$$Select(S, \lambda s\ p_1(s) \wedge p_2(s)) \equiv Select(Select(S, \lambda s\ p_1(s)), \lambda s\ p_2(s))$$

The transformation essentially chooses an ordering for the Selection predicates. In this case, the $age > 15$ predicate is chosen to be executed first because the cost of applying the *age* method is low and the predicate is independent of the bound variable for the Image operation. The latter condition means that the query $Select(People, \lambda p_2\ p_2.age > 15)$ can be pre-processed and accessed as a constant in the Select function nested in the Image query.

For different query annotations, a different transformation or predicate ordering might have been chosen within the region. For example, if the *age* method were costly, if there was an index on a person's residence or if residences were clustered, it might have been better to evaluate the residence-checking predicate first. Such an evaluation would do an indexed retrieval of a small number of co-residents, each of which could then be checked for the age requirement.

At this point the region control finds no other useful algebraic transformations to apply to the query, so the updated tree is returned to the global optimizer control. The global control then sends the query to the join conversion region. This region's control searches the tree for predicates that would indicate joins and finds two; $p_1.residence = o.residence$ and $v \in p.cars$. These predicates each match two sets in the query (the People set with the subset of People over 15 and the Vehicles set with the People set) indicating that the query could be expressed using Join operations. The

[5]To simplify the figures we will not annotate the nodes, and will discuss any relevant changes in annotations in the text.

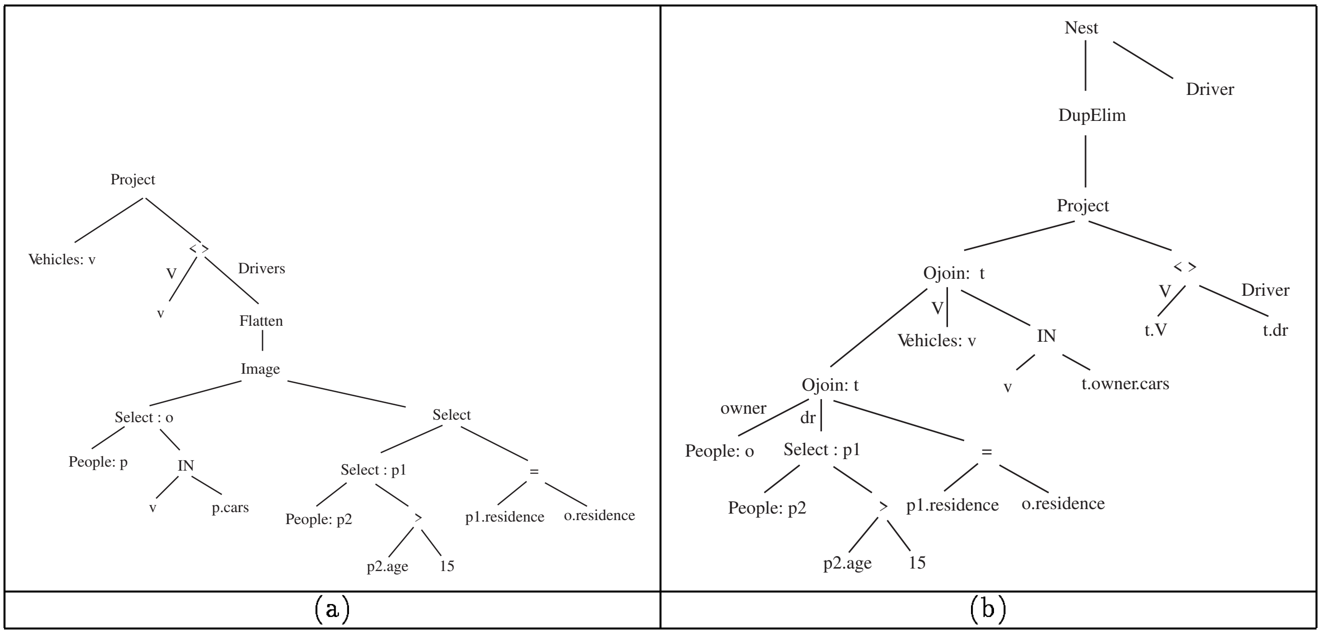

Figure 6: Tranformations of Insurance Query

query can thus be converted to a representation in a canonical join form as shown in Figure 6b. In this representation, all join operations are clustered at the base of the tree, and operations that manipulate the structure of the query result (Project, DupElim and Nest in this example) are pushed to the top of the tree.

Although this region transformed the query, the transformation that was applied was not a context-free algebraic rewrite as in the cost-guided region, but involved a tree transformation procedure written in a general programming language. The control of this region simply checks to see if the transformation is appropriate (by checking for join predicates) then executes the transformation procedure. This region does not consider expected execution cost of the result and, indeed, may produce a query expression that is more expensive to execute than the input expression. On the other hand, the output from this region may offer more opportunities for optimization in other regions.

The latter is the case in this example. The canonical join query is sent to the join reordering region where the order of the join operations is modified and join methods are recommended. In this region, a dynamic programming strategy is used to search for good join orderings (similarly to the System R strategy [27]). The region eventually settles on the join ordering represented in the query of Figure 7 (note that annotations, including join method choices, are not shown in this figure). The reordering here is chosen on the basis of the direct access from the People to the Vehicles set and the expected cardinalities of the result sets. Each member of the People set is accessed once to retrieve the *cars* property. An average of .75 cars per person will be directly retrieved (recall that half of the people are too young to own cars, and half of the cars are jointly owned), resulting in 1.5 million pairs to be matched by residence to the set of People over 15. The Join methods chosen take advantage of the direct access between sets. Alternate join sequences are not optimal since they would match the two People sets with no indexes, resulting in four million pairs of people to access into the Vehicles set.

The reordered join query is returned to the optimizer control, which can then decide to send it to another region. In this example, suppose the optimizer chooses to halt and select the query tree

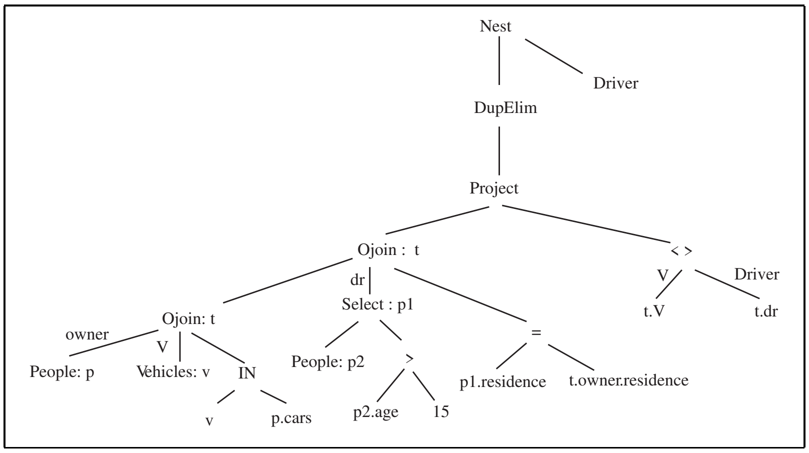

Figure 7: Query result after join reordering.

which presents the most cost effective solution. In this case, the result of the join reordering region would be found to be best and this solution would be returned to the query processing system.

Our example illustrates a simple use of the multiple region architecture for query optimization. The regions used in this example differ in the kinds of transformations they can apply and in the control used to apply those transformations. The join conversion region has a simple sequential control and can apply a single transformation procedure to a query. The cost-guided region is modelled after rule-based query optimizers, extended to consider cost information in the transformation application process. The transformation rules are based on the algebraic properties of the operations involved in the query. Cost data about those operations provides further information for the hill-climbing strategy used by the region to control application of the transformation rules. We would not expect this region to consider join ordering rules, since join reordering is better handled by the dynamic programming strategy of the join reordering region. Such a strategy results in the same effect as applying rules, but can be more efficient since it is tuned to the join reordering transformation process.

In this example the global optimizer control sent the query sequentially through the regions, and chose the result which appeared to be the most cost effective method for processing the query. We are exploring different ways for the optimizer to control movement of a query through regions. For example, the optimizer might first send a query to the region for which it most closely fits an input profile. In this scenario the join reordering region might be preferred over the cost-guided region (which can accept arbitrary input trees) for processing any query involving joins.

5 Summary and directions

This paper described concepts for a new approach to extensibility in query optimizers for persistent object bases. Our approach allows for extending the repertoire of strategies for optimizing query expressions and for combining selected strategies into a process for optimizing a given query. This approach supports extensions to the language and operations for expressing queries by allowing the addition to the optimizer of new techniques for dealing with expressions involving the new operations.

The philosophy behind the extensible strategy optimizer is that multiple regions will provide more flexibility in optimization processes that can be applied and also provide a straightforward way to extend the optimization techniques embodied in the optimizer. Regions may also provide more control over the kinds of optimization processes that can be applied. The use of multiple regions allows the application of arbitrary transformations in a controlled fashion. A region could be limited in the kinds of transformations it applies and thus could possibly be more efficient in its application of those transformations. The search for transformations could be limited within a region and thus a region could provide a better estimate of the cost of its part of the optimization process. A region might also be able to give better estimates of the current cost of a query plan. If a region is only applying local transformations then it might be able to update efficiently the current estimated cost of a query plan, since the changes in any one transformation are localized. Regions also provide the possibility of more efficient control over the optimization process since the search within a region might be limited.

In future work we will produce a detailed specification for an architecture implementing this approach to extensibility. This will include, in particular, a detailed specification of the region interface, a mechanism for inter-region control, and a language for node annotations. We plan to implement a prototype optimizer which will include the regions used in the example here, and will eventually integrate techniques proposed by others for processing queries in a persistent object system.

References

[1] F. Bancilhon and W. Kim, "Object-Oriented Database Systems: In Transition," *SIGMOD Record*, vol. 19, pp. 49–53, Dec 1990.

[2] J. Banerjee *et al.*, "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems*, vol. 5, pp. 3–26, January 1987.

[3] C. Beeri and Y. Kornatzky, "Algebraic Optimization of Object-Oriented Query Languages," in *Proceedings ICDT*, (Paris, France), 1990.

[4] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects," Tech. Rep. ACT-OODS-132-89, MCC, 1989.

[5] S. Cluet and C. Delobel, "Towards a Unification of Rewrite Based Optimization Techniques for Object-Oriented Queries," tech. rep., Altaïr, B.P. 105, 78153, Rocquencourt France, May 1991.

[6] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers," in *Proceedings of the 13th VLDB Conference*, pp. 197–208, 1987.

[7] U. Dayal, "Queries and Views in an Object-Oriented Data Model," in *Proceedings of the 2nd International Workshop on Database Programming Languages*, June 1989.

[8] J. C. Freytag, "A Rule-Based View of Query Optimization," in *SIGMOD Proceedings*, pp. 173–180, May 1987.

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

[10] G. Graefe and D. J. DeWitt, "The EXODUS Optimizer Generator," in *SIGMOD Proceedings*, pp. 160–172, ACM, May 1987.

[11] G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: A Prospectus," in *Advances in Object-Oriented Database Systems*, pp. 358–363, International Workshop on Object-Oriented Database Systems, September 1988.

[12] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," in *SIGMOD Proceedings*, pp. 358–366, ACM, June 1989.

[13] L. M. Haas *et al.*, "Extensible Query Processing in Starburst," in *SIGMOD Proceedings*, pp. 377–388, ACM, June 1989.

[14] B. P. Jenq *et al.*, "Query Processing in Distributed ORION," in *EDBT*, pp. 169–187, 1990.

[15] A. Kemper, C. Kilger, and G. Moerkotte, "Function Materialization in Object Bases," in *SIGMOD Proceedings*, pp. 258–267, ACM, 1991.

[16] A. Kemper and G. Moerkotte, "Access Support in Object Bases," in *SIGMOD Proceedings*, pp. 364–374, ACM, 1990.

[17] R. S. B. Lanzelotte and P. Valduriez, "Extending the Search Strategy in a Query Optimizer," in *Proceedings of the 17th VLDB Conference*, pp. 363 – 373, 1991.

[18] R. S. B. Lanzelotte, P. 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.

[19] C. Lécluse, P. Richard, and F. Velez, "O_2, an Object-Oriented Data Model," in *SIGMOD Proceedings*, pp. 424–433, ACM, June 1988.

[20] C. G. Legaria and R. Barrera, "A Rule-Based Query Optimizer," Tech. Rep. 92, Centro de Investigacion y de Estudios Avanzados Del IPN, 1989.

[21] B. Liskov *et al.*, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, pp. 564–576, August 1977.

[22] D. Maier and J. Stein, "Indexing in an Object-Oriented Database," in *International Workshop on Object-Oriented Database Systems*, pp. 171–182, 1986.

[23] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," in *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, eds.), pp. 355–392, Cambridge, MA: MIT Press, 1987.

[24] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," in *Readings in Object-Oriented Database Systems* (S. B. Zdonik and D. Maier, eds.), San Mateo, CA: Morgan Kaufmann, 1990.

[25] S. L. Osborn, "Identity, Equality and Query Optimization," in *Advances in Object-Oriented Database Systems*, pp. 346–351, International Workshop on Object-Oriented Database Systems, September 1988.

[26] E. Sciore and J. Sieg, Jr., "A Modular Query Optimizer Generator," in *Proceedings of the 6th International Conference on Data Engineering*, pp. 146–153, 1990.

[27] P. G. Selinger *et al.*, "Access Path Selection in a Relational Database Management System," in *SIGMOD Proceedings*, pp. 23–34, ACM, 1979.

[28] G. M. Shaw and S. B. Zdonik, "Object-Oriented Queries: Equivalence and Optimization," in *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pp. 264–278, December 1989.

[29] G. M. Shaw and S. B. Zdonik, "A Query Algebra for Object-Oriented Databases," in *Proceedings of the 6th International Conference on Data Engineering*, pp. 154–162, IEEE, 1990. An early version of this paper appears as Brown University tech report CS-89-19.

[30] D. D. Straube, *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, Univ. of Alberta, December 1990.

[31] D. D. Straube and M. Özsu, "Query Transformation Rules for an Object Algebra," Tech. Rep. CS-89-23, University of Alberta, 1989.

[32] S. Zdonik and G. Mitchell, "ENCORE: An Object-Oriented Approach to Database Modelling and Querying," *IEEE Data Engineering Bulletin*, vol. 14, June 1991.

[33] S. B. Zdonik, "Query Optimization in Object-Oriented Database Systems," in *Proceedings of the Hawaii International Conference on System Science*, January 1989.

[34] S. B. Zdonik and P. Wegner, "Language and Methodology for Object-Oriented Database Environments," in *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.