Experiences Building the Open OODB Query Optimizer

José A. Blakeley, Texas Instruments
William J. McKenna, University of Colorado at Boulder
Goetz Graefe, Portland State University

Abstract

This paper reports our experiences building the query optimizer for TI's Open OODB system. To the best of our knowledge, it is the first working object query optimizer to be based on a complete extensible optimization framework including logical algebra, execution algorithms, property enforcers, logical transformation rules, implementation rules, and selectivity and cost estimation. Our algebra incorporates a new *materialize* operator with its corresponding logical transformation and implementation rules that enable the optimization of path expressions. Initial experiments on queries obtained from the object query optimization literature demonstrate that our optimizer is able to derive plans that are as efficient as, and often substantially more efficient than, the plans generated by other query optimization strategies. These experiments demonstrate that our initial choices for populating each part of our optimization framework are reasonable. Our experience also shows that having a complete optimization framework is crucial for two reasons. First, it allows the optimizer to discover plans that cannot be revealed by exploring only the alternatives provided by the logical algebra and its transformations. Second, it helps and forces the database system designer to consider all parts of the framework and to maintain a good balance of choices when incorporating a new logical operator, execution algorithm, transformation rule, or implementation rule. The Open OODB query optimizer was constructed using the Volcano Optimizer Generator, demonstrating that this second-generation optimizer generator enables rapid development of efficient and effective query optimizers for non-standard data models and systems.

1. Introduction

Query processing remains one of the most important challenges to researchers and developers of Object-Oriented Database Management Systems (OODBs) [1]. Most research efforts on object query optimization have concentrated on the design of system components only, e.g., object algebras [17, 19, 21], query rewriting techniques [3], global query processing architectures [4, 11, 19], indexing techniques [8-10], and new execution algorithms to efficiently traverse complex object structures such as pointer-based joins [18] and complex object assembly [7]. However, little has been reported on the development of complete working object query optimizers [14].

This paper reports our experiences in the design and development of the object query optimizer for the *Open OODB* system [22], an modular and extensible OODB system being built at Texas Instruments. The lack of a standard object data model and application program interfaces has not only slowed acceptance of OODB technology by potential users but also slowed the development of standardized OODB components such as the query optimizer. The Open OODB team attempts to overcome these shortcomings by describing the design space of OODBs and their modules, building a data model-independent architectural framework that allows system developers to configure independently useful modules to form an OODB, verifying the suitability of this open approach by implementing an OODB to these specifications, and determining areas (e.g., module interfaces) where consensus among the OODB research community exists or is possible. Our approach to the design of the Open OODB query optimizer has been influenced strongly by the extensibility and modularity goals of the Open OODB project. The philosophy used in the design of the Open OODB optimizer can be summarized by the following goals.

- *Extensibility.* We believe that an object query optimizer must be extensible to facilitate experimentation with (i) new algebraic operators, (ii) new algebraic transformation rules, (iii) new execution algorithms, (iv) improved statistics and cost models, (v) physical formats and structures (e.g., data compression), (vi) enforcer algorithms for physical properties (e.g., sort order, presence in a particular address space), (vii) new state space search strategies, and (viii) improved quality of plans (e.g., thoroughness of search) [20]. An extensible object query optimizer will give us a powerful research workbench on which to try new ideas emerging in the object query processing area.

- *Cost-effective, rapid development.* We wanted to develop a near-production-quality optimizer with a reasonable amount of time and resources. While our optimizer is not at production quality yet, the crucial pieces are in place and are described here. The missing functionality can be viewed as refinements of existing pieces, e.g., more accurate selectivity and cost estimation.

- *Performance.* The optimizer should be able to carry out optimization tasks efficiently. Moderately complex queries should be optimized on today's workstations in less than 1 sec.

- *Effectiveness.* The optimizer should produce plans that substantially improve the efficiency of queries. One of our goals is to ensure that queries over large data collections are executed by the Open OODB at least as fast as by relational systems.

The Open OODB team chose to build the query optimizer using the Volcano Optimizer Generator [6] for four reasons. First, Volcano provides a fairly complete optimization framework extensible in most of the dimensions described above, except in the search strategy. Second, since there is currently no widely accepted formal basis for object query optimization, we wanted to be able to "borrow" and experiment with algebraic operators from several existing object algebras, execution algorithms and their transformations to discover the most effective combination of ideas. Third, the development effort using an extensible system is reduced because some important components are provided, most importantly a complete search engine, and the extensible system provides a framework for dividing the large software development effort for a database query optimizer into defined modules. Fourth, we wanted to validate the suitability of the Volcano optimizer generator as a tool for the development for non-standard database systems, in particular for object-oriented database systems.

Our approach to designing the Open OODB query processor is eclectic in that it tries to leverage existing results and components available in the object query processing community. Our long term goal is to develop an extensible and robust query processing framework for the Open OODB system that will allow us to improve this important module of our system as the field evolves. The main contributions of the work reported in this paper are:

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0287...$1.50

• A description of the design and development of a working query optimizer for an OODB based on a complete optimization framework including logical algebra, execution algorithms, logical and physical properties, selectivity and cost estimation, logical transformation rules, and implementation rules.

• An effective algebraic transformation and optimization framework that separates a "user" algebra for specifying queries (consisting of arbitrarily complex operations and arguments) from an algebra in which the query optimizer performs transformations (consisting of simpler operations with simpler arguments).

• A novel logical operator, *materialize*, that indicates where path expressions exploit references embedded in the data. In a sense, it brings an object component "into scope." Its corresponding transformation and implementation rules enable algebraic optimization of path expressions. In our framework, path expressions become algebraic expressions in the form of compositions of materialize operators that allow the exploration of alternative reference resolution strategies within an algebraic framework.

• The importance of physical properties in the search process, not for heuristic pruning as suggested by other researchers, e.g. [3], but for goal-directed search. We show how our extensible optimization framework allows the definition of a physical property *presence in memory*, and that inclusion of this property allows the search engine to discover plans that cannot be revealed by exploring only the logical algebra and its alternatives.

• Experimental evidence that set matching operations such as join and intersection and their corresponding algorithms developed in the relational context remain relevant in object-oriented database systems, not only for operations on sets and for value-based matching, but also as alternative implementation methods for object-oriented path expressions. Even if precomputed access paths (stored references) exist, naive traversal of such references ("goto's on disk") may result in suboptimal performance. Algebraic equivalence transformations for these operations, including commutativity and associativity, permit significant optimizations. In the case where only uni-directional links exist between objects, we show the need to consider processing operators that resolve the object references in the opposite direction.

• The validation of the Volcano optimizer generator for the generation of an object query optimizer.

The remainder of this paper is organized as follows. Section 2 compares previous designs with our techniques. Section 3 describes our query optimization framework and discusses some specific issues in object query optimization. Section 4 presents some of our initial experimental results, including comparisons with optimization strategies found in the literature. Section 5 reflects on lessons learned about query optimization in object-oriented database systems and about the Volcano optimizer generator. Section 6 offers our conclusions and directions for future research.

2. Previous Work

There is currently no working commercial or experimental OODB query optimizer that satisfies our design philosophy. This section compares our work with previous query optimizer proposals. While not all designs discussed in this section have been implemented, they represent the body of work most closely related to ours.

Mitchell et al. [11] have proposed an extensible object query processing architecture similar to the one by Sciore and Sieg [15]. Both proposals argue for a new dimension of query optimization extensibility, namely the support of multiple optimizer control strategies and the capability to add new control strategies. Mitchell et al.'s envisioned optimizer design consists of a collection of optimization "regions," each of which can transform queries according to a particular control strategy, a set of algebraic transformations, and a cost model. A global optimizer control coordinates the movement of a query among these regions.

One of the motivations for the use of optimization regions is that query transformations may involve operators at various levels of an expression tree. We believe that providing specialized region optimizers is an intriguing research problem. However, another way to interpret the multi-level query transformation problem is to separate the rich algebra with which users communicate with the database system from the simpler algebra (although computationally equivalent) that specifies a query to the optimizer. Algebraic operators with semantically rich arguments allow a more succinct representation of complex queries and also serve as a better communication tool among database system designers. However, our experience indicates that it is substantially more difficult and cumbersome to design and build algebraic transformations for operators with complex arguments than for operators with simpler arguments. Since the design and development of transformation rules represents a major portion of the work involved in building a query optimizer, we believe that this makes a strong argument for separating the "user interface" algebra (which might permit very complex operator arguments) and an algebra with simple arguments suitable as input to an algebraic optimizer. The logical algebra from which query optimization starts in the Open OODB optimizer is an example of the second algebra.

Orenstein et al. [14] describe the design of the query optimizer used in the ObjectStore OODB. An interesting feature of the ObjectStore optimizer is its dynamic plan selection capability whereby the optimizer generates multiple execution strategies at compile time and makes a final plan selection at run-time based on the availability of indices. This dynamic capability permits users to modify some of the physical characteristics of the objects being queried (e.g., adding and deleting indices) without having to recompile their applications. Unfortunately, that paper does not report results on the optimizer's efficiency nor the effectiveness of the plans it generates.

The ObjectStore optimizer has three major disadvantages. First, it appears to handle only a small set of fixed optimization strategies, namely the use of indices for path expressions. In addition to using the conventional sequential scan on collections, the optimizer seems to use only index-based scan and not to consider other efficient execution algorithms for joins or assembly of complex objects. Hence, it is unclear how the optimizer can be easily extended to incorporate new optimization strategies or execution algorithms. Second, it is not cost-based – as a result, the optimizer may miss the optimal plan even on queries especially crafted to demonstrate the optimization strategies it supports as illustrated later in Section 4. Third, the optimizer appears not to be based on a query algebra, which makes it difficult to enumerate and explore equivalent execution expressions.

Cluet and Delobel [3] describe a type-based rewriting technique to be used as a basis for a new query optimizer being implemented in O2 [13]. Their approach "unifies" algebraic and type-based rewriting techniques, permits factorization of common subexpressions, and supports heuristics to limit rewriting. However, Cluet and Delobel's technique ignores cost estimation and execution algorithm selection and mixes *some* physical (index availability and object clustering) and logical (extent availability) concerns during rewriting.

Cluet and Delobel exploit type information to decompose initial complex arguments of a query into a set of simpler operators and to rewrite path expressions ("pointer chasing") into joins. While rewriting complex arguments into joins is a transformation on the *logical* level, Cluet and Delobel heuristically prune their search space based on *physical* information about clustering and indices. In our approach, on the other hand, path expressions are represented by the materialize operator, and the decision to transform a materialize operator into a logical join is based on logical rather than physical information. Execution algorithms such as assembly and hybrid hash join are treated uniformly as physical implementations of the logical operators materialize and join. Open OODB's execution algorithm selection, which is based solely on anticipated execution costs, yields optimal strategies for resolving logical references without violating the clean separation of logical and physical optimization concerns.

Cluet and Delobel point out that common subexpression factorization is an important issue in object-oriented query optimization, and propose a technique for performing exhaustive factorization. Global common subexpression factorization is one of the features that we obtain for free by using the Volcano optimizer generator.

Straube and Ozsu [19] propose a query processing methodology that includes a formal object calculus and algebra, calculus to algebra translation, type-checking of algebraic expressions, algebraic optimization based on the application of algebraic transformation axioms, and access plan generation. Their object algebra includes the operators union, difference, select (which is really a combination of select and Cartesian product), generate (a combination of unnest and method application collector), and map (a special case of generate). To the best of our knowledge, Straube and Ozsu's query optimizer design has not been validated in an implementation.

3. Extensible Query Optimization Framework

In this section, we discuss how optimization works in the Open OODB query optimizer. As mentioned in the introduction, it is based on the Volcano optimizer generator, which translates specifications of algebra operators, transformation rules, and implementation rules into source code (C or C++), combines the generated code with an algebraic search engine, and links it with the support functions provided by the optimizer implementor such as cost functions and with the other DBMS code [6]. The essential components of Volcano-generated query optimizers are the logical algebra, the set of execution algorithms, logical and physical properties, the cost model, and the optimization rules. We describe these components for the Open OODB optimizer in turn as well as the user query language and the simplification step from the user-level complex algebra to our optimizer input algebra.

User Query Language

We use ZQL[C++] [2] as a representative user query language in our examples. ZQL[C++] is an SQL-based object query language designed to be well-integrated with C++. ZQL[C++] is a strongly-typed query language that assumes the C++ type system as object data model. ZQL[C++] supports queries on type extents as well as on any user-defined, collection-valued expressions (currently, sets and lists). ZQL[C++] predicates can be composed by mixing arbitrary Boolean-valued functions defined by the user.

The query in Figure 1 illustrates the syntax and some of the features of ZQL[C++]. The example represents a query expressed inside a C++ program to obtain the employee name and department of all employees who are at least 32 years old, work in a department on the third floor, and have received a salary increase on or after January 1, 1992. The query illustrates (i) the use of a set-valued program variable (i.e., result) to contain the result of the query, (ii) path expressions, (iii) join and projection (i.e., generation of objects of type Newobject with new identity), and (iv) abstract data type operators (e.g., ">=" for Date), and (v) data abstraction (i.e., all predicates are expressed in terms of the types' public interface). The last clause in the query represents a comparison of department objects based on their OID's.

```
Set<Newobject> *result;
Date lr(01,01,1992);
result = SELECT Newobject( e.name(), d.name() )
FROM Employee e IN Employees, Department d IN Departments
WHERE d.floor() == 3 && e.age() >= 32 && e.last_raise() >= lr
&& e.department() == d ;
```

Figure 1. Example ZQL[C++] Query.

The formulation of this query also illustrates the use of various C++ expressions in each of the SELECT, FROM, and WHERE clauses. It uses a call to the Newobject class constructor operator in the SELECT clause. It defines range variables e and d as Employee and Department instances, respectively, using C++'s syntax in the FROM clause. It uses C++'s path and conditional expressions in the WHERE clause.

Since the input to the Open OODB query optimizer is an algebraic query graph, our choice for the user query language does not limit the generality of the Open OODB query optimizer. In fact, our optimizer could easily be adapted to work with other proposed object-oriented user query languages such as the ones offered by the ObjectStore [14] and O2 [3] systems.

Logical Algebra

In our model of query processing, a logical algebra expression is the input into the query optimizer. The set of execution algorithms defines the query evaluation environment and is discussed in a later subsection. The goal of our optimizer is to map a logical algebra expression to a combination of execution algorithms, optimized by transformations of the logical algebra expression and by choosing the most cost-effective implementation algorithms.

A large number of algebras have been proposed for object-oriented databases; most of those are too complex to be manipulated efficiently by an algebraic optimizer, because much of the query semantics is expressed in operator arguments rather than in the actual algebra operators. In fact, the operator arguments (e.g., selection predicates) frequently are more reminiscent of a (non-procedural) calculus language than of an algebra, which is inherently a procedural language. Instead, we have designed our logical algebra so that as much as possible of the query semantics is captured in the algebraic operators and in the algebra expression while the operator arguments are as simple as possible. The optimizer design presumes, therefore, that a preprocessor exists that translates a query's parse tree into an initial algebra expression. This translation, called *simplification*, is very straightforward because there is no need for optimality and therefore for choices in this translation.

Since the basic paradigm of Open OODB's query language ZQL[C++] is selection from collections of objects, the foundation of our algebra are the traditional set and relation operators. *Select*, *project*, *join*, *intersection*, and *union* are defined as in relational systems. An *unnest* operator is used to manipulate set-valued components. In addition, we have defined a new logical operator, called *materialize* or *Mat*, that represents each link of path expressions such as *city.country.president.name*. The purpose of this operator is to explicitly indicate the use of inter-object

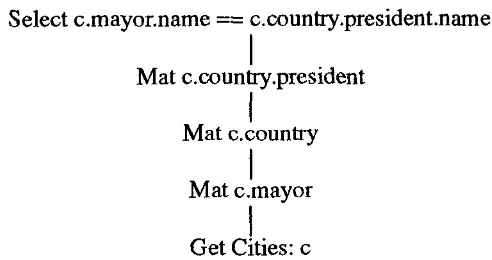

Figure 2. A Logical Algebra Expression Using the Mat Operator.

references. A single *Mat* operator can materialize multiple components, or multiple *Mat* operators can be used. Another way to think of this operator is as a "scope definition," because it lets elements of a path expression come into scope so that these elements may be used in later operations. The scoping rules in the optimizer input algebra are very simple. An object component gets into scope either by being scanned (captured using the logical *Get* operator in the leaves of expressions trees) or by being referenced (captured in the *Mat* operator). Components remain in scope until a projection discards them.

As an example for the materialize operator, the query:
SELECT City c in Cities
WHERE c.mayor().name() == c.country().president().name();
is translated into the logical algebra expression shown in Figure 2. The purpose of the materialize operator is to indicate to the optimizer where path expressions are used and where therefore algebraic transformations can be applied. In the example query, the materialize operators can trade their positions in the query expression, with the condition that "country" must be materialized before "president." We presume that the "name" instance variables are similar to record fields that need not be explicitly materialized.

Query Simplification

The Open OODB query processing model uses a *query simplification* stage to transform ZQL[C++] parse trees into an equivalent algebraic operator graph with simple arguments suitable as input to the Open OODB optimizer. *Complex arguments* are any predicate terms containing path expressions that have not been brought into scope or collection-valued operations (e.g., an existentially quantified subquery in the argument of a select).

Currently, simplification has been defined for select-from-where ZQL[C++] queries whose condition involves arbitrary conjunctive Boolean expressions with existentially quantified nested subqueries, but no aggregates. Because of space limitations, we omit a formal description of this simplification. Interesting simplifications include simplifications for single- and set-valued path expressions as well as existentially quantified subqueries. A simplification involving a single-valued path expression was illustrated in the previous subsection when we introduced the materialize operator. Simplification involving nested subqueries uses tactics similar to the ones used for nested subqueries in SQL [12]. Consider a query involving a set-valued path *task.team-members* denoting all employees working in a particular project task. This path is translated into a logical algebra expression shown in Figure 3. Since the team-members component of a task is a set of references to employee objects, we need to first reveal (unnest) all references to employee objects that are members of this set. Calling a reference to an employee team member m, the output of the unnest operator is a set of pairs [t, m]. The materialize operator resolves all employee references to employee objects present in memory, producing [t, e] pairs that can be used as the input to subsequent select or other operators. We will discuss later the optimization of a query subsuming the one shown in Figure 3.

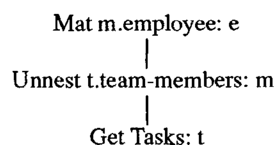

Figure 3. Algebra Expression for Set-Valued Path Expression.

Execution Algorithms

Since our algebra includes all traditional set operators, our execution engine also includes (or shortly will include) the traditional set processing algorithms, namely file (extent) and index scan as well as value-based matching (e.g., intersection, union, join) based on hybrid hash join. This algorithm also supports equality of a reference attribute on one side and object identifiers on the other side.

We are currently implementing (and optimizing for) two algorithms that promise to have a significant performance impact in object-oriented database systems. *Pointer-based joins*, recently analyzed by Shekita et al. [18], are sometimes superior to other join methods; therefore, an optimizer should consider them among the alternative implementations for join.

The second algorithm our optimizer considers is *assembly*, developed in the REVELATION project [7]. Assembly generalizes pointer-joins as it permits multiple references, even recursive object references, i.e., transitive closures of sub-component relationships. It achieves higher performance than sequences of traditional operators by maintaining a window of open, unresolved references in order to exploit multiple objects located on a single disk page and to sequence disk read operations into an elevator pattern over physical disk locations.

Properties and Property Enforcement

In order to determine whether a transformation is applicable or whether an algorithm can implement a given logical expression, it is often necessary to inspect the logical and physical properties of intermediate results. *Logical properties* are properties of an expression determined by the logical operators before execution algorithms are chosen (e.g., type or size of intermediate results). *Physical properties* depend on execution algorithms selected. The standard example for a physical property in relational query optimization is the sort order [16]. In object-oriented query processing, an important property is *presence in memory*. Our optimizer currently does not use merge-join for value-based matching, therefore it supports only presence in memory.

Physical properties have no role in the logical algebra but are important in the realm of query execution algorithms. In our framework, execution algorithms implement a logical operator, *enforce* some physical property, or both. For instance, the assembly algorithm is used to enforce the present-in-memory property and to implement the logical materialize operator.

Cost Model

Currently, our cost model is very traditional. We consider both CPU and I/O costs, and "charge" less for sequential than for random I/O. Assembly's I/O cost captures the fact that seek distances are minimized by charging less than for a random I/O operation. Actual assembly performance including the effects of buffer hits can only be studied in the context of a real, working system; therefore, we delay validating and refining assembly's cost

function until the query plan executor becomes operational.

While our cost model is not precise yet, the important points are that cost is integrated into our optimizer framework and that query evaluation plans are transformed and compared based on anticipated execution costs, not purely on heuristics. Very little research has been done to-date on cost models and formulas in object-oriented database systems[1]; however, as such research evolves, we will incorporate it swiftly. Cost is encapsulated in an abstract data type (ADT) and tuning an algorithm's cost formula is a very localized change.

Transformation Rules

Since our logical algebra is based on the relational algebra, our transformation rules include known relational transformations plus some new ones pertaining to the materialize operator. These transformations move materialize operators above and beneath ("through") selection, join, and set operators, provided none of the other operators depends on a scope defined by materialize.

One rule that we believe can be very important and effective in query optimization in object-oriented database systems transforms materialize operations into joins, not because joins are always a good choice but because joins are an alternative execution strategy that should be chosen or rejected based on anticipated execution costs. If the scope introduced by a materialize operator is actually a scannable object (a set object, file, etc.), the materialize operator can be transformed into a join. For example, if there is a set or file of countries in Figure 2, the materialize operation bringing c.country into scope can be replaced by a join operation. Figure 4 shows the resulting query plan.

Once a materialize operator has been transformed into a join, all transformation rules and implementation rules for join apply. Join associativity is closely related to the commutativity of multiple materialize operators. Join commutativity permits exploring query plan alternatives that are usually ignored in object query optimization, e.g., traversing single-directional inter-object links (pointers) in their opposite (not pre-computed) direction. We discuss physical algorithms and their specification for the optimizer generator in the next subsection.

Implementation Rules

The implementation rules establish the correspondence between logical algebra expressions and execution algorithms. Algorithm selection is the second important aspect of query optimization, beyond logical equivalence transformations. Query evaluation

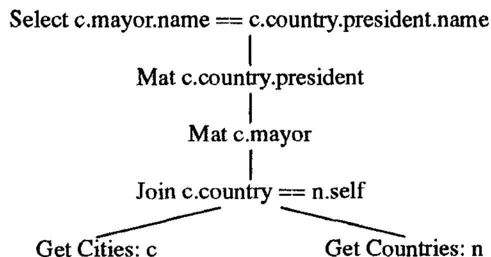

Figure 4. Transforming a Mat Operator into a Join.

[1] In our optimizer, the issues requiring immediate attention are object clustering, adaptive reclustering, and statistical summary data about clustering to enable more accurate cost estimation.

algorithms are important and still continually improved in the "simpler" relational model, and will have an even larger effect in object-oriented database systems. The optimizer chooses algorithms based on implementation rules, an algorithm's ability to deliver a logical expression with the desired physical properties, and cost estimations.

Summary

The Open OODB optimizer leverages the Volcano optimizer generator for algebraic query optimization comprising operator transformations, algorithm selection, and enforcement of properties. As the basic query paradigm in ZQL[C++] is selection from collections, we built on previous work in relational query processing by using well-known transformation rules and implementation algorithms for set operations. Algebraic operators with complex arguments are translated into equivalent expressions with "simpler" operations before optimization. Optimization of path expressions is facilitated by a new materialize operator that brings object components "into scope" and indicates where the optimizer can apply transformation rules. Using the Volcano optimizer generator permitted and forced us to consider the entire spectrum of issues in query optimization, from the logical algebra design via algorithm choices to the cost model. Thus, rather than considering isolated optimization issues, our work resulted in a complete query optimizer. In the next section, we evaluate the optimizer using a few representative examples.

4. Experimental Results

This section presents the optimization of selected queries using actual runs of the Open OODB optimizer. The queries show interesting optimizations realized by the Open OODB optimizer and allow us to compare and contrast its effectiveness against other object-oriented optimization approaches. Specifically, our examples show

(1) the need to consider links traversals between objects in both directions (even if only one direction is supported with physical pointers), recognizing the utility of set matching algorithms developed for relational join,

(2) the effectiveness of using physical properties in guiding the search process,

(3) and the effectiveness of cost-based search over heuristically-guided search.

All queries were optimized on a DEC Station 5000/125, which has a 25 MHz RISC CPU and 32 MB of memory. Model description file and support functions were developed during a summer internship in about 10 weeks, including transformation rules, implementation rules, cost estimation and property functions.

In all example queries, we assume the catalog information shown in Table 1. Objects in user-defined *sets* and *type extents* are assumed to be densely packed on pages. If no index can be used to assist in selectivity estimation, selectivity of selection predicates is assumed to be 10%, which is naive and will later be replaced by a more accurate selectivity estimation method.

Path Expressions and Inter-Object References

This section illustrates the use of the novel materialize operator to facilitate algebraic optimization of path expressions and the contribution of relational algorithms to the performance of object-oriented database query processing. Consider a ZQL[C++] query to retrieve the name, department name, and job name of all employees who work in a plant in Dallas, Texas:

Type	Set Name	Set Card.	Obj. Size [bytes]	Type Extent?	Extent Card.
Capital	Capitals	160	400	No	
City	Cities	10000	200	No	
Country			300	Yes	160
Department			400	Yes	1000
Employee	Employees	50000	250	Yes	200000
Information			400	Yes	1000
Job			250	Yes	5000
Person			100	Yes	100000
Plant			1000	No	
Task	Tasks		12	Yes	10000

Table 1. Catalog Information.

```
SELECT Newobject (e.name(), e.dept().name(), e.job().name())
FROM Employee e in Employees
WHERE e.dept().plant().location() == "Dallas";
```

After the simplification step, this query is presented to our optimizer as shown in Figure 5. The materialize operators with arguments job(), dept(), and plant(), represent the logical references in the path expressions in the select and project operator arguments.

Optimized with all transformation and implementation rules enabled, this query results in the query execution plan shown in Figure 6. Since all rules are enabled and exhaustive search is performed, the access plan in Figure 6 is the optimal plan for Query 1. There are several interesting observations in comparing the query in Figure 5 and the optimal plan in Figure 6. First, two materialize operators in Figure 5 are transformed to join operations, which permits the selection of hybrid hash join in Figure 6. This transformation is possible because the materialize operators in Figure 5 explicitly represent each of the path expressions' links that need to be traversed to establish the relationships between object components. Second, object components are linked together in different orders (compare where the employee-job relationship is established in Figure 5 and Figure 6). Reversing the order in which the links are traversed is possible because the materialize operator makes their use explicit. Third, the plan traverses some links in a direction opposite to that of the physical pointers between the objects. This interesting and initially counterintuitive choice stems from the small extent size for the *Job* type and the small cardinality of filtered and assembled Department/Plant objects, which permit very efficient executions of hybrid hash join using only in-memory hash tables and no overflow files. Fourth, the placement of the assembly algorithm in Figure 6 attempts to minimize the number of plant components that have to be assembled (in this case, 1,000, the number of Department objects). A particularly unfortunate choice would have been to assemble first departments and then plants for each of the 50,000 employees, although it might be considered the most "natural" execution of Figure 5.

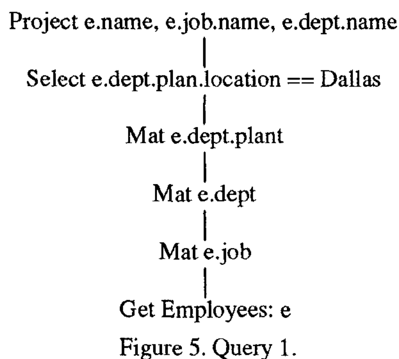

Figure 5. Query 1.

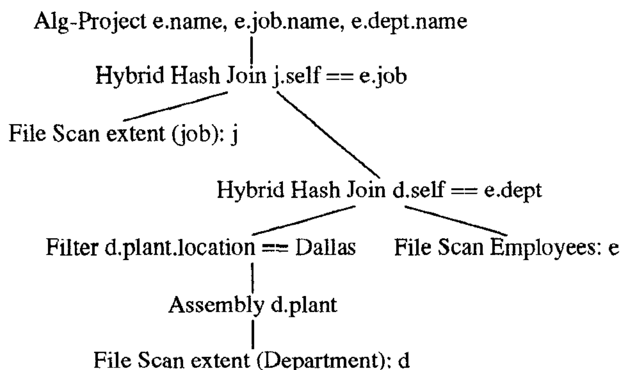

Figure 6. Optimal Execution Plan for Query 1.

To assess the value of including a value-based hybrid hash join in our suite of algorithms, we disabled the join commutativity rule to force the optimizer to consider a more "naive" query execution strategy (i.e., one using pointer-chasing algorithms). The resulting optimized plan is shown in Figure 7, which is more than four times as expensive as the optimized plan in Figure 6. The slight decrease in optimization effort is not worth the large increase in anticipated query execution time.

Table 2 summarizes optimization and expected execution times required to optimize this same query with different optimizers (simulated by disabling various rules in our optimizer). This increase is due to the assumption that the Plant type does not have an extent, and the optimizer estimates that 50,000 page faults may result from assembling the plant components of each of the employee objects (if the plant objects are clustered with information about the products manufactured at each plant, for instance). Of course, there are fewer plants, but as indicated in Table 1, and consistent with the current prototype implementation of the Open OODB, we assumed that cardinality information is kept only with extents and set instances and that the Plant type does not have a separate extent. However, this example indicates

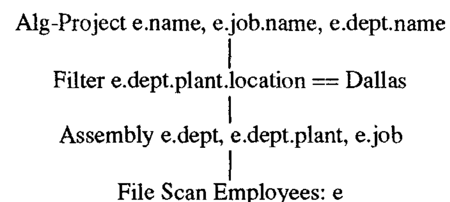

Figure 7. Query 1 Plan w/o Join Commutativity.

	Optim. Time [sec]	% of Exh. Search	Est. Exec. Time [sec]	% of Optimal Exec. Time
All Rules	0.21	100	161	100
W/o Comm.	0.12	57	681	422
W/o Window	0.11	52	1188	737

Table 2. Optimization Results for Query 1.

that additional cardinality information should be maintained whether or not the objects belong to a set or extent, and we may revisit this issue in a later version of the system. For the department and job components, on the other hand, there is an explicit extent and the optimizer can place an upper bound on the number of I/O operations needed to assemble the department and job components of an employee object. Thus, the optimizer can fairly accurately estimate the cost of assembling these components.

While the first and second lines in Table 2 show the value of join algorithms in object-oriented database systems, the second and third lines show the value of assembly using a window of open references [7]. Restricting assembly's window size to one (leaving the join commutativity rule disabled) forces the operator to assemble one object at-a-time and thus prevent it from optimizing disk seeks. The assembly algorithm becomes similar to the lookup component of an unclustered index scan.

Summarizing, optimization time decreases as rules are disabled in the optimizer. However, the large increase in estimated execution time demonstrates that investing more time in the optimization effort is a profitable choice, and that much better plans are indeed found by considering a larger number of alternatives.

Our optimizer captures exactly the tradeoffs explored by Shekita and Carey [18]. Naive pointer chasing ("goto's on disk") is insufficient for high-performance query execution engines. In addition, optimizers for object-oriented database systems must consider a variety of algorithms for each logical operation as well as transformations that permit the use of these algorithms in the search process.

The need for powerful, accurate, and flexible optimization tools increases with the advent of object-oriented database system; it does not decrease as has been claimed elsewhere [14]. In particular, optimizing path expressions and inter-object references does not become a simpler problem due to precomputed joins (stored references) and path indices; instead, it becomes a more complex problem because the number of competing execution strategies grows with the number of alternative access paths. In other words, naive pointer chasing of precomputed relationships is one of many possible methods, but not the best in all situations.

Physical Properties and Goal-Directed Search

In this section, we consider two versions of an example query and use optimization results for these two queries to demonstrate the value of modeling physical properties in the optimization process. Consider Query 2 shown in Figure 8, which selects cities whose mayor is called "Joe." Suppose the set Cities is indexed on the path "mayor().name()" using a path index. The optimizer will then choose the query execution plan shown in Figure 8, because the index scan can determine the answer set without actually retrieving any "mayor" objects from disk.

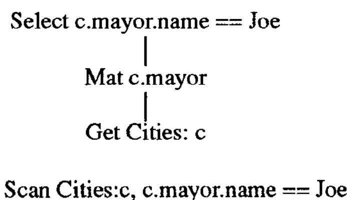

Figure 8. Query 2 and its Optimal Execution Plan.

The crucial optimization rule for this query is an implementation rule that allows collapsing the select-materialize-file scan sequence into a single index scan with a predicate (this rule is subsequently called the *collapse-to-index-scan rule*). In this case, the mayor component objects are never read into memory, and the estimated execution cost of this plan is 0.08 sec, with an optimization time of 0.05 sec.

If the collapse-to-index-scan rule is disabled (or no index on this path exists), the optimizer returns the plan shown in Figure 9. In this plan, the execution engine must assemble the "mayor" component for each city before the selection predicate can be applied. The cost of this plan is 119.6 sec, with an optimization time of 0.05 sec – a substantial increase in execution time (about four orders of magnitude) with no appreciable decrease in optimization effort.

Let us now consider Query 3, shown in Figure 10, which is a small variant of Query 2 by requiring mayor's ages in the result. In other words, the mayor component must be retrieved for this query. In order to evaluate the projection, the assembled city and mayor objects satisfying the selection predicate must be present in memory. While it is necessary for the optimizer to consider all possibilities in order to guarantee the best plan, including the collapse-to-index-scan rule, the plan in Figure 8 does not retrieve the mayor components; thus, it is not applicable for the query in Figure 10 without change. The most promising plan for the query is to only assemble those mayor components for cities satisfying the selection predicate. Yet, in order to maintain a strict separation between issues pertaining to the logical algebra and to the execution algorithms, we do not want to introduce a logical transformation rule to allow a selection to be pushed below a materialize operator if a path index is present. The solution to this optimization problem relies on the use of physical properties to guide the search process, a solution that is not possible in query optimizers that consider only logical transformations, but not query evaluation algorithms and physical properties.

The Volcano optimizer search engine uses physical properties to drive the search top-down. In other words, the search process

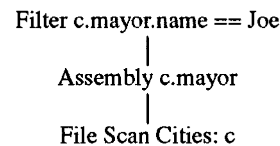

Figure 9. Query 2 Plan w/o Collapse-to-Index-Scan.

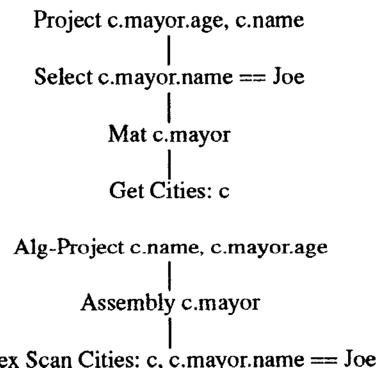

Figure 10. Query 3 and its Optimal Execution Plan.

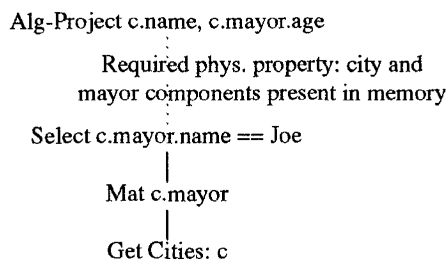

Figure 11. Search State while Optimizing Query 3.

considers only those subplans that can deliver the physical properties that are required by the algorithm of the containing (larger) plan. This is unlike other optimizers, which construct the plans bottom up and keep all subplans that deliver "interesting" properties (defined a priori by the optimizer implementor) without knowing whether these subplans might ever participate in a larger plan. For example, the algorithm that implements projection, Alg-Project, requires that its inputs deliver assembled city/mayor objects present in memory. The state of the search while considering Alg-Project to implement the project operator is shown in Figure 11. Notice that in Figure 11, there is one physical operator (Alg-Project) and three logical operators (Select, Mat, Get). At this point in the search, the optimizer searches for the best plan to produce the logical expression comprising the three logical operators such that the resulting plan's output satisfies the physical property vector of Alg-Project's input, i.e., city and mayor components are present in memory.

For the expression consisting of the three logical operators in Figure 11, the collapse-to-index-scan rule cannot be used, because the index scan only delivers city, not mayor, objects present in memory. The search engine therefore has two alternatives when optimizing the select subquery, namely (1) to use filter to implement select, and request that the filter input deliver city/mayor objects present in memory (and, hence, the filter itself will deliver these properties); or (2) to introduce the assembly algorithm to enforce the delivery of the requested objects "in memory."

When the first alternative is evaluated by the search algorithm, the execution plan found for the select subquery will be the same as the execution plan for Query 2 with the collapse-to-index-scan rule disabled, because filter will require that its input deliver assembled city/mayor objects present in memory, and an assembly-file scan algorithm sequence is the only choice in this case. The result is a total cost of 119.6 sec for processing Query 3.

When the second alternative is evaluated, the input to the assembly enforcer is the same execution plan as that for Query 2 with the collapse-to-index-scan rule enabled, since the index scan delivers only city, not mayor, objects present in memory. With the city objects in memory as the result of the index scan, the assembly enforcer delivers the referenced mayor objects in memory. Having the city and mayor objects in memory is precisely the physical property required of the Alg-Project input. The plan in this case has a cost of 0.12 sec (the optimizer estimates that only 2 cities have mayors named "Joe"). Thus, the optimal plan found for Query 3 is the plan shown in Figure 10. Comparing the plans in Figure 9 and Figure 10, three orders of magnitude in performance were gained.

Physical properties, in particular sort order, have been recognized as important in relational query optimization [16]. In this example, we show the utility of using the physical property "presence in memory" to drive the search process in the Open OODB optimizer. This example also indicates that an optimizer should be extensible enough to incorporate new physical properties and their enforcers.

Heuristic- vs. Cost-Based Optimization

In this section, we show the advantages of a strictly cost-based approach to OODB query optimization and compare it to approaches that rely on heuristically guided optimization [14]. In particular, ObjectStore's query optimizer uses a fixed, greedy strategy designed to exploit any available indices. We show that such a greedy strategy will not always lead to the optimal plan.

Consider the query shown in Figure 12, taken with a slight modification from [14], which selects tasks with a completion time of 100 hours and a team member called "Fred." Figure 12 also shows the optimal query evaluation plan found by the Open OODB optimizer assuming the catalog information given in Table 1. Figure 13 shows the query evaluation plan that uses both indices, i.e., the plan found by a greedy optimizer. Table 3 shows the expected execution times for this query with full optimization and with greedy index use. The columns represent different assumptions about the existence of indices. As is immediately obvious, the optimal plan uses only one index, the index on time, not both indices as the greedy algorithm would choose. The greedy plan is slower than the optimal plan by more than a factor of 5. Thus, the greedy algorithm is too simplistic to permit effective query optimization in object-oriented database systems.

Summary

In this section, we have used queries selected from the literature to demonstrate that the Open OODB optimizer framework based

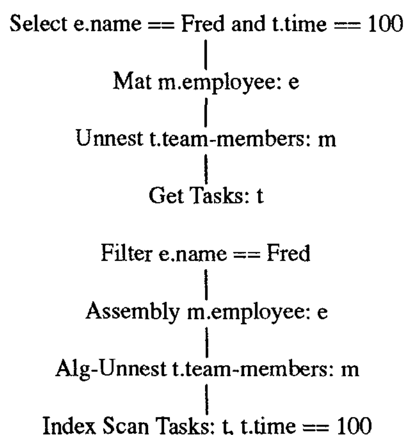

Figure 12. Query 4 and its Optimal Execution Plan.

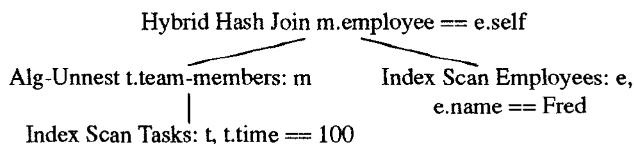

Figure 13. Greedy Evaluation Plan for Query 4.

Indices	None	Time only	Name only	Both
All rules	108	1.73	28.4	1.73
Greedy use	108	1.73	28.4	10.1

Table 3. Anticipated Execution Times for Query 4.

on the Volcano optimizer generator is significantly more powerful than previous optimizers for object-oriented database systems. In the next section, we summarize the lessons we have learned during the design, development, and preliminary evaluation of the Open OODB optimizer.

5. Retrospection on using the Volcano Optimizer Generator

In this section we summarize some of the lessons we learned while building the Open OODB query optimizer using the Volcano optimizer generator tool.

• Lesson 1: Having a comprehensive yet extensible query optimization framework that includes not only an object algebra and its transformations, but also the algorithms to implement the logical operators, selectivity and cost estimation, and enforcement of properties is very important for the development of a working query optimizer. In other words, it is very seductive for a designer to incorporate new powerful logical algebra operators without considering their implications on the complexity of the corresponding logical transformations or implementation algorithms. The optimization framework forced us to visit all stages of the framework every time we considered adding a new logical operator, execution algorithm, and transformation or implementation rule.

• Lesson 2: The fact that the Volcano optimizer generator provides such a framework was very helpful in building the Open OODB query optimizer. The optimizer generator provides a complete framework for query optimization that considers all critical aspects of this task: logical operators, execution algorithms, logical transformations, physical and logical properties, cost estimations, and property derivation functions that encapsulate schema manipulation, statistical descriptions of intermediate results, and selectivity estimation. Query optimizer generator technology has become mature enough to substantially help DBMS designers build efficient and effective optimizers that include non-standard operators such as our materialize operator and algorithms such as pointer-join and complex object assembly.

• Lesson 3: Using an optimizer generator allowed us to focus immediately on issues directly related to OODB query optimization and ignore some aspects not specific to object query optimization such as plan enumeration and search efficiency.

• Lesson 4: Designing and implementing transformation rules for logical operators with simple arguments is substantially easier than designing rules for operators with complex arguments. The following intuitive design rule served us well: if an argument can encapsulate some form of collection-valued operation or might otherwise result in costly I/O operations, consider representing that part of the argument as a separate logical operator in the algebra and in the optimizer. If this rule is followed, possibly expensive iterative operations are exposed as operators for rewrite by the algebraic optimizer.

• Lesson 5: Supporting property enforcement allows exploration of strategies not covered by exclusively algebraic optimization frameworks (see Section 4 above).

• Lesson 6: Using the Volcano optimizer generator allows us to leverage earlier experience accumulated in this tool, which is also being used by other database researchers to develop new query optimizers. This is important not only with regard to functionality, but also with regard to the robustness of the generated code. The more database system designers use a tool, the more trustworthy the tool will become. Moreover, as soon as a bug is fixed or an enhancement to the optimizer generator is made, all optimizers generated with the tool will benefit.

• Lesson 7: A comparison of hash join using a hash table of the referenced objects and an equivalent assembly algorithm with a large window suggests a new "warm-start" assembly algorithm, i.e., the ability to scan a scannable object into main memory before the normal complex object assembly operation commences. We plan on studying this algorithm variant and its effectiveness in future research.

In addition, the Open OODB optimizer implementation resulted in the following suggestions for improving the optimizer generator.

• Lesson 8: A fair amount of time was spent building support functions such as catalog accesses, operator graph manipulation, selectivity estimation, and cost calculations. Future versions of the optimizer generator should provide facilities to automatically generate the majority of these functions.

• Lesson 9: We found it sometimes necessary to transform logical operator arguments in a way that is similar to the algebraic operator transformations. These logical argument transformations may be subject to rules completely different than the algebraic operator transformations. Although the Volcano Optimizer Generator supports separate transformation rule groups and reuse of the logical expression enumerator, we found its rule language cumbersome for performing the types of transformations we needed. We are considering enhancing the rule language to make it powerful and flexible enough to support these transformations. Notice that the distinction between argument and operator rule matching sounds similar to the region-based optimization proposals of Sieg and Sciore and of Mitchell et al. applied to two regions; we have begun to explore designs that permit using the Volcano optimizer generator and its parameterized, rule-driven search engine to generate both the global control among multiple regions as well as the local optimizer within each region.

• Lesson 10: The Volcano optimizer generator's rule language and the requirements for support functions are far from user-friendly and require redesign and better documentation. We view the Volcano optimizer generator effort as part of a continuum: The EXODUS optimizer generator contributed the paradigm and basic system architecture; the Volcano effort contributed an efficient search strategy plus better abstractions for properties and costs, thus making extensible query optimization based on optimizer generators practical; a new effort will focus on more convenient packaging for the Volcano optimizer generator. Two such efforts are currently under-way, one at the Oregon Graduate Institute and one at the University of Texas at Austin.

6. Conclusions and Future Work

The Open OODB optimizer presented in this paper is the first working query optimizer for an OODB to utilize a comprehensive yet extensible optimization framework including logical algebra, execution algorithms, logical and physical properties, property enforcers, selectivity estimation, cost model, logical transformation rules, and implementation rules. Having a complete optimization framework is crucial for two reasons. First, it allows the optimizer to discover plans that cannot be revealed by exploring only the alternatives provided by the logical algebra and its transformations. Second, it forces the database system designer to visit all aspects of the framework before incorporating any new logical operator, transformation rule, or implementation algorithm.

The Volcano optimizer generator allowed us to develop the Open OODB optimizer in a very short amount of time. The resulting Open OODB optimizer code is very modular. It consists of transformation and implementation rules and a variety of support functions. Each support function is associated with one of the logical operators, implementation algorithms, enforcers, or the abstract data types for logical properties, physical properties, or

costs. We believe that the modularization prescribed by the optimizer generator will enable us and other developers to extend and refine the Open OODB query optimizer in the future.

Our initial experiments on queries obtained from the object query optimization literature show that our optimizer derives as efficient and frequently more efficient plans for these queries than other optimizers. Thus, by separating the user-level algebra (with arbitrarily complex operations and arguments) and the optimizable algebra (using only simple operator arguments) and by introducing the novel *materialize* operator, we have been able to extend algebra-based query optimization to a wide variety of queries in object-oriented database systems. Moreover, the optimization times required in our optimizer are very modest, demonstrating that exhaustive search and therefore truly optimal plans are feasible for moderately complex queries over object-oriented databases.

Our future plans with the Volcano optimizer generator and the Open OODB optimizer include several directions of research. First, we will evaluate and refine the "rougher" modules, in particular selectivity and cost estimation. Second, although the Volcano optimizer generator provides mechanisms for heuristic guidance and pruning, we have not evaluated them for object-oriented query optimization yet. Third, we will investigate automatic generation of support functions. Fourth, we will generate code to transform operator arguments. Finally, we will transfer query execution concepts and algorithms from the Volcano query execution module [5] to the Open OODB system in order to complement our optimization work.

Acknowledgements

Suggestions by Craig Thompson, David Wells, Ben Zorn, and David Maier's weekly reading group at the Oregon Graduate Institute (Winter 1993) have significantly improved the presentation of this paper. – This paper is based on research partially supported by DARPA with contracts DAAB 07-90-C-B920 and DAAB 07-91-C-Q518 and by NSF with grants IRI-8996270, IRI-8912618, and IRI-9119446.

References

[1] F. Bancilhon and W. Kim, Object-Oriented Database Systems: In Transition, *ACM SIGMOD Record, Special Issue on Directions for Future Database Research and Development 19*, 4 (December 1990), 49.

[2] J. A. Blakeley, C. W. Thompson and A. Alashqur, A Strawman Reference Model for Object Query Languages, *Computer Standards & Interfaces 13*(1991), 185.

[3] S. Cluet and C. Delobel, A General Framework for the Optimization of Object-Oriented Queries, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 383.

[4] G. Graefe and D. Maier, Query Optimization in Object-Oriented Database Systems: A Prospectus, in *Advances in Object-Oriented Database Sys.*, vol. 334, K. R. Dittrich (ed.), Springer-Verlag, September 1988, 358.

[5] G. Graefe, Volcano, An Extensible and Parallel Dataflow Query Processing System, *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1993.

[6] G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, *Proc. IEEE Conf. on Data Eng.*, Vienna, Austria, 1993.

[7] T. Keller, G. Graefe and D. Maier, Efficient Assembly of Complex Objects, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.

[8] A. Kemper and G. Moerkotte, Advanced Query Processing in Object Bases Using Access Support Relations, *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, 290.

[9] W. Kim, K. C. Kim and A. Dale, Indexing Techniques for Object-Oriented Databases, in *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky (ed.), ACM and Addison-Wesley, 1989.

[10] D. Maier and J. Stein, Indexing in an Object-Oriented DBMS, *Proc. Int'l Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, September 1986, 171.

[11] G. Mitchell, S. B. Zdonik and U. Dayal, An Architecture for Query Processing in Persistent Object Stores, *Proc. Hawaii Conf. on System Sciences*, 1993.

[12] M. Muralikrishna, Improved Unnesting Algorithms for Join Aggregate SQL Queries, *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, 1992, 91.

[13] O. Deux et al., The Story of O_2, *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 91.

[14] J. Orenstein, S. Haradhvala, B. Margulies and D. Sakahara, Query Processing in the ObjectStore Database System, *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 403.

[15] E. Sciore and J. Sieg, A Modular Query Optimizer Generator, *Proc. IEEE Conf. on Data Eng.*, Los Angeles, CA, February 1990, 146.

[16] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[17] G. M. Shaw and S. B. Zdonik, A Query Algebra for Object-Oriented Databases, *Proc. IEEE Conf. on Data Eng.*, Los Angeles, CA, February 1990, 154.

[18] E. J. Shekita and M. J. Carey, A Performance Evaluation of Pointer-Based Joins, *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 300.

[19] D. D. Straube and M. T. Ozsu, Queries and Query Processing in Object-Oriented Database Systems, *ACM Trans. on Inf. Sys. 8*, 4 (1990), 387.

[20] C. Thompson, ed., *Report on DARPA Open OODB Workshop II: Preliminary Module Interface Specification Workshop*, Texas Instruments, Inc., CS Labs., Dallas, TX., October 1991.

[21] S. L. Vandenberg and D. J. DeWitt, Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance, *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 158.

[22] D. Wells, J. A. Blakeley and C. W. Thompson, Architecture of an Open Object-Oriented Database Management System, *IEEE Computer 25*, 10 (October 1992), 74.