Query Optimization in Object-Oriented Databases

Stanley B. Zdonik

Brown University

Abstract

The use of data abstraction in object-oriented databases places a burden on the ability of the system to perform query optimization. This paper discusses a framework for query specification and optimization that is applicable to object-oriented database systems that take a strict view of data abstraction. It examines techniques that preserve much of the optimization potential of relational languages by limiting the query language. It further examines techniques for query optimization that involve type-specific rewrite rules.

1. Introduction

Relational database systems have been very successful at providing ad hoc query languages. These languges allow users to write associative retrievals which the query processor is able to optimize. The ability to optimize queries derives largely from the fact that the relational model is very simple and uniform. Studies of relational algebra have led to an understanding of the formal properties of relational operators. For example, it is well-known that the select and join operations commute. This leads to the standard optimization strategy of pushing selections past joins.

In object-oriented databases, we are faced with a data model that allows for extension of the basic types by some form of data abstraction. This is a powerful modeling facility, but it has severe implications regarding how we might do query optimization. No longer is there a single simple model with a limited set of carefully studied operators. Each user-defined type effectively introduces a brand new algebra. How are we to understand the properties of these algebras in order to allow the same degree of optimization that we had in the relational world?

This paper discusses some of the issues involved in query optimization in the context of object-oriented databases. We feel that the subject requires additional study. We will argue that it is possible to provide query facilities including optimization in limited ways for object-oriented databases.

2. Brief Description of the Model

The ENCORE object-oriented database system [ZW86] provides for type definitions that are based on the notion of abstract types. A type definition T consists of a representation R, a set of operations Op, and a set of properties Pr. R is any other previously defined type. Op is a set of operations that have privileged access to R, and Pr is the definition of the abstract state of the object. A property p that relates an object x to an object y is a pair of the form (x,y) and as such is considered to be an object.

Two types A and B can be related to each other by means of an *is-a* property that requires that they be behaviorally compatible. If B is-a A, then instances of B can be used in any context in which instances of A were expected. Types also induce set objects that we will call *classes*. A type T has a corresponding class Ts that contains all current instances of T. T will be a collection of all objects that have been created as an instance of type T. There may be classes that do not correspond to types. We typically create subtypes to introduce additional operational behavior. A subclass may be created even if members of the subclass have no additional operations than those of the superclass. For example, we might have the type Car and the corresponding class Cars with an additional subclass called Blue_Cars. Blue cars have exactly the same operations as Cars, but can be usefully distinguished for reasons of the application.

It is classes over which we will form queries. A *query* is a function that returns selected members of some class. Classes might have implementations that differ. For example, some classes might be implemented by a B-tree, while others might be a hash table or a heap. The implementations that are chosen for class objects will have a large impact on the efficiency of some searches. Query optimization will select a query processing strategy that will use the different implementaions of classes in the most advantageous way.

3. Query Language

The type Aggregate (and all its subtypes) define two operations called Select and Image that take Aggregates as input and return Aggregates as results. Select returns those members of a given aggregate that satisfy a given predicate. Image takes an Aggregate A and a function f as input and returns an Aggregate that contains the result of applying f to each element of A. A Query is therefore modeled as an application of the Select or Image operations. The language that is used to form queries and query predicates P will be described in this section.

We have chosen to restrict the predicate language to only allow predicates over the abstract state of objects (i.e., its properties). That is, we can build expressions that rely on the values of properties, but we do not allow them to contain calls

to arbitrary type-defined operations. As an example, consider the following query expression:

Select [Stacks, lambda (s) pop (s) = 3]

It selects those stacks from the set of all known stacks that if we were to apply the *pop* operation to them would return 3. Although it should be possible to form queries involving user-defined operations, we will not make any attempt to optimize these expressions. This makes query optimization more managable. We will return to this point later.

The query language consists of the following functions:

Select (A: Aggregate, P: Predicate) -> Aggregate
Image (A: Aggregate, F: Function) -> Aggregate
Union (A1: Aggregate, A2: Aggregate) ->
Aggregate (also, Intersection)

For_all (A: Aggregate, P: Predicate) -> T/F
For_some (A: Aggregate, P: Predicate) -> T/F
Contained_in (A1: Aggregate, A2: Aggregate) -> T/F
Member (x: Object, A: Aggregate) -> T/F
(alternatively: x: Object **in** A: Aggregate)

Select returns those elements of the given aggregate A that match the given predicate P. That is, Select (A, P) = { a | a is in A and P(a) }. *Image* returns an aggregate defined as Image (A, f) = { f(a) | a is in A }. *Union* returns the set union of the two aggregates A1 and A2. The types of the objects that are members of these aggregates must be of a comparable type (i.e., of the same type or of types that are in a type-subtype relationship with each other).

The last four functions are predicate formers. *For_all* returns **true** if every P(x) is true for all x in A. *For_some* returns **true** if there exists an x in A such that P(x) is true. *Contained_in* is **true** if A1 is a subset of A2, and Member is **true** if x is an element of A.

Other operations could be added, such as the standard aggregate (not to be confused with Aggregate types) operations of *average*, *count*, *max*, and *min*. We will ignore these for now to simplify the rest of the paper.

It is worth considering whether a query language such as this one is as powerful as the relational algebra. The relational algebra consists primarily of the three operations *Select*, *Project*, and *Join* . Select is supported directly by our query language. The other two are somewhat more problematic.

Join is normally used to compute associations between records of the same or different types. In an object-oriented world associations are normally computed by predefined properties or attributes. One applies a property to an object x (much like a function) to return the object or objects to which x is related. The Join operation produces a new relation of an as yet undefined type. Computing the Join automatically introduces the new relation type. Although it is possible in many object-oriented systems to crete new types dynamically, it is uncommon and is somewhat antithetical to the object-oriented style in which types tend to be predefined. Computing types on the fly could prohibit, or at least complicate, the ability to do static type checking. As long as all possible relation types that we will ever use are predefined, we have the abilty to simulate Join. We therefore, conclude that within the stylistic limits of the object-oriented paradigm, Join-like operations are possible.

It is entirely possible to define an operation called *Match* with the following form:

Match (t: T, P (y : T1)) returns Set [T: Type]

on a type T. The second argument to *Match* is a predicate that might have the following structure:

P (x: T1) returns Set [T2]
Select [T2s, lambda (t2) t2.p = x.q]

where T2s is a set of objects of type T2 (some other type against which we wish to do the matching). Here the match is performed by finding those elements of T2s that have a property value that matches the q property of x (an object of type T1). Repeated use of the Match operation allows us to compute the associations between objects based on the current states of the objects in the same way that *Join* does.

Projection is an operation that discards information, largely for reporting purposes. In an object-oriented world, reporting would be handled by separate operations that would be used to print specific aspects of an object. Projection, like Join, also creates a new type. Object-oriented databases could incorporate operations that produce such new types, but dynamically creating types could have undesirable effect on our ability to do static type checking.

4. Approaches to Query Optimization

One way to approach the optimization of queries in an object-oriented database is to perform syntactic query transformations similar to those used in the relational algebra. With the syntactic approach, it is sometimes possible to perform transformations that always produce better results than the original query expression. An example from relational optimization is to always push selection past joins. Although each type provides a new algebra, if we restrict query expressions to use only the abstract state (i.e., properties) of the new types and generic operations over Aggregate types (e.g., the query language defined above), we can write down syntactic rules that are true for any type since the basic semantics of properties and sets are fixed and well-known. The next section gives some examples of this approach.

A second technique requires restricting the legal representations for types and the language that is used to access them. We can construct methods from these languages whose bodies can be used in the optimization process. The method body can be substituted in the query expression, and this new expression can be optimized by using techniques that are available on the more restricted representations. Relations are a good candidate for one of these representations.

A third technique is based on the use of type-specific rewrite rules that are specified as a part of the interface of a type. These rewrite rules capture algebraic properties of user-defined operations and the ways in which they combine. All three of these techniques are discussed in more detail in the remainder of the paper.

The way in which transformations are applied depends on our cost functions that are used to estimate the cost associated with evaluating a query. Often the cost functions need to know how a type is implemented. We then apply transformations selectively to arrive at expressions that make use of underlying access methods. For example, we would like to evaluate expressions early if their computation can take advantage of an index. Indexing in an object-oriented database is discussed at the end of the paper.

Several of these optimization techniques require that the optimizer have knowledge about how things are implemented. We find that this assumption is reasonable in the object-oriented world even though this seems like it could be a breach of the basic principle of encapsulation. The optimizer is a trusted part of the system and can be allowed to access the implementation as long as the and user code cannot.

Another interesting approach involves the use of type-level semantics to transform the query in fundamental ways to a wholly different query that is provably equivalent to the original. Wheres the syntactic methods apply rules that are true in all domains, these techniques apply additional information that is dependent on application-level semantics. This kind of optimization has become known as semantic query optimization [HZ80] and seems well-suited to object-oriented models. Many of the transformations in semantic query optimization rely on inclusion relations between sets. The object hierarchies that appear in object-oriented systems supply exactly the kind of information that is required. A more detailed examination of these methods is beyond the scope of this paper.

4.1 Query Transformations

It is possible to discover transformation rules that can generally be applied to query expressions. Like query optimization in relational systems, we can use these transformations to get new query expressions that have lower expected costs. The following five rules are examples:

(1) Select [S1, lambda (s) s.p in S2] =
Image [S2, inverse (p)]

(2) Select [Select [S1, P1], P2] =
Select [Select [S1, P2], P1]

(3) Union [Select [S, P1], Select [S, P2]] =
Union [Select [S, P1 or P2]]

(4) Union [Image [S1, f], Image [S2, f]] =
Image [Union [S1, S2], f]]

(5) Select [S1, lambda (s) s.p1.p2 in S3] =
Select [S1, lambda (s) s.p1 in
Select [S2, lambda (s) s.p2 in S3]]

The following example illustrates the use of the first rule. Consider the following query that retrieves the Employees that work in the shoe department.

```
Select [Employees, lambda (e) e.dept.dname = "shoe"]
```

Application of rule (1), will produce the following equivalent query:

```
Image [
    Select [Departments,
            lambda (d) d.dname = "shoe"],
    inverse (dept)]
```

This optimization technique relies on the fact that the inverse property exists for the property that was originally used. Although this is not true in the general case, if the aggregates in the query are being implemented by relations, the computation of the property values and their inverses is all done with a single application of the relational *join* operation. *Join* is inherently bi-directional. Inverses might also exist if two properties p and q are defined to always obey the constraint

$$(x.p = y) \Rightarrow (y.q = x)$$

The first rule is analogous to the relational query optimization technique that pushes selections past joins.

Query optimization requires that we also have a mechanism for determining an estimate of the relative costs of two expressions. For the purposes of this paper we will make the simplifying assumption that any set that can be retrieved by using an index is cheaper than one that cannot. Indexing will be discussed in a later section.

4.2 Optimization Based on Method Bodies

Object-oriented databases can unify value-based and identity-based models by using either approach as an implementation of a more abstract concept. This concept is the property (as in the ENCORE data model [ZW85]) or relationship. If there is a property p that relates objects of type A to objects of type B, this simply implies that there potentially exists a relationship between any object of type A and some objects of type B. The expression x.p simply means "find the objects related to x by the p relationship." There is no implementation implied. This relationship can be implemented in many ways including by pointers (i.e., by identity) or by a join-like operation (i.e., by value).

The implementation of methods can be expressed in terms of different languages. They can be written in terms of lower-level programming languages like C, or they can be written in terms of higher-level languages like a relational query language (e.g., SQL). An implementation in terms of a higher-level language leaves open the possibility for more significant query optimization.

An object-oriented data model provides the ability to write arbitrary new methods in a general-purpose programming language and the ability to encapsulate the implementation of this new method via an abstract interface specification. When a new method is implemented in this way, we have ultimate flexibility in the behavior of that method allowing us to define very complex relationships, but we might lose the ability to use the implementation in the optimization of queries. In order to perform any optimization on methods of this type, we must supply additional information about their algebraic properties (see next section).

If, however, we implement a type (and, therefore, its methods) in terms of a more restricted set of data types and operations like relations and the relational algebra, we have the ability to use the method definition in the optimization process. This assumes that the query optimizer has access to the internals of a type. We believe that this does not compromise *the principle of encapsulation within our system.* Encapsulation is a contract between implementors of an abstraction and users of that abstraction. The users are other programs. We must not rely on the internals of a data abstraction in the bodies of other code. The query optimizer is a trusted part of the system. Moreover, if we allow it to access the internals of a data abstraction T, it will not require that that the optimizer be rewritten when the implementaion of T changes.

The rest of this section presents an example of how using the internals of a data abstraction that has been implemented in terms of relations can be useful for query optimization.

Assume that we have two abstract types S and T defined as in the following. Further assume that these two types are implemented as specified to their right.

Define Type S	**Relation** S (... , p: Pid, ...)
Properties:	**Method** get_p (x)
p: P	Project_T (Join (x, T))
Define Type T	**Relation** T (... , p: Pid, q: Q, ...)
Properties:	**Method** get_q (x)
q: Q	x.q

Type S has a property called p that maps objects of type S to objects of type P. Type T has a property called q that maps objects of type T to objects of type Q. Type S is implemented by a relation in which there is an attribute called p that contains values in the set Pid. Type T is also implemented by a relation which contains an attribute called p whose value is drawn from the same set of identifiers Pid. Relation S can be joined with relation T on their p attributes. This fact is used in the implementaion of the method get_p. We use the convention that Join of a tuple x and a relation R has the value that we would get by joining a relation that contains the single tuple x with R.

Suppose that we had the following query, expressed in terms of the abstract types S and T and a constant c.

```
Select [S, lambda (x) c in x.p.q]
```

Using the definition of the method get_p, we can transfer this query into

```
Select [S,
   lambda (x)
      c in Project_q (
         Project_T (Join (x, T)))]
```

We can then make use of the following identity

```
Select [S,

   lambda (x)
      c in Project_q (
         Project_T (Join (x, T)))] =
Project_S (
   Select [ Join (S, T),
      lambda (x) x.q = c])
```

to get the new query

```
Project_S (
   Select [ Join (S, T),
      lambda (x) c in x.q])
```

This query can be optimized by conventional techniques by pushing the selection past the join to arrive at the following optimized query

```
Project_S (
   Join (S,
      Select [T, lambda (x) c in x.q]))
```

which could then be given to a relational algebra based query processor.

4.3 Type-specific rewrite rules

We can also obtain rewrite rules like those described above that involve expressions over the operations of an arbitrary abstract type. Each new abstract type forms a many-sorted algebra [BW], and the relationships between these operations can be expressed by a set of axioms as in LARCH [Gu85]. These axioms (or rewrite rules) are equations that relate terms over abstract operations to other such terms or to terms over previously defined types. The classic example of such an axiom for an abstract IntStack type that defines the standard push and pop operations is

```
S : IntStack
x : Int
pop (push (x, S)) = x
```

A rule such as this one can be used to simplify queries. The above rule is a statement of the fact that push and pop are inverse operations. That is, any query expression that includes a subexpression of the form pop (push (e, S) can be simplified to simply the expression e.

We can also use axioms for new Aggregate types. These are often more useful for simplifying queries. Consider the type EmpArray which represents an array of Employees. This type introduces several new operations including *Insert*, *Delete*, *Is_in* , and *Find* . The signatures of these operations are:

```
Insert: EmpArray, Int, Emp -> EmpArray
Delete: EmpArray, Emp -> EmpArray
Is_in: EmpArray, Emp -> Bool
Find: EmpArray, Emp -> Int
```

These operations are understood by rewrite rules (i.e., axioms) like the following:

```
b: EmpArray; r, s: Emp; i: Int;

Find (Insert (b, i, r), s) equiv
   if r=s then i else Find (b, s)
Find (Delete (b, r), s) equiv
   if r=s then 0 else Find (b, s)
```

For the following simple query:

```
Select [b, lambda (e) e = r]
```

we can show that it is equivalent to the operation invocation:

```
Find (b, r)
```

If there exists an access path such that we can determine that the EmpArray contains the record r, we can write the equivalent expression:

```
Find (Insert (b, i, r), r)
```

which we know must be equal to i.

5. Indices on Abstract Types

We will propose the use of indices as a way to implement efficient access to aggregate types. However, there are a few problems that must first be addressed.

The first involves what can be indexed. An index is really nothing more than a precompiled search. In other words, we store lists of all objects that would be returned if we were to search a set for all occurences of objects that match some criterion. Usually this criterion is that a key value is equal to a given constant. By using the index, we avoid having to do the search. With abstract types, we could potentially index on the result of applying methods. That is, we could keep a list of all objects x that would return a given value v if a method m were to be applied to x. A set of these lists for all possible v would be an index.

$$I_m = \{ (v_i, (x_1, \ldots, x_n)) \mid v_i \text{ is a possible value of m } \& \; m(x_j) = v_i \; (1 \le j \le n) \}$$

Notice that m is a function of a single argument. Restricting our attention to methods of one argument that have no side-effects simplifies our approach to query processing. The get_value method that is available for all properties is an example of this type of function. For this reason, we restrict our query expressions to use only properties instead of more general operations.

Our data abstraction facilities allow property values to be computed in arbitrary ways by the get_property_value operation. In many cases, this operation simply looks up the value of the property from a stored piece of the representation. However, it is possible for this operation to compute a complex function based on several pieces of the representation. This conveniently allows a straightforward way to incorporate derived properties into the model. It potentially complicates our ability to analyze query expressions since we cannot always transform them directly onto simple manipulations of the underlying storage.

The existence of an index short-circuits this problem. If we have the value of a property stored in an index, it becomes irrelevant at retrieval time how this value is derived. It is available by doing a lookup on the index.

Another problem is a result of the fact that abstract types can be built on top of other abstract types. An aggregate object type can be built on top of another aggregate type, and so on. It is possible to provide indices at any of these levels. For example, some level in such an implementaion hierarchy can be impemented by a B-tree which is itself an index. Even though a query expression is written in terms of the highest level interface, we would like to be able to take advantage of indices at deeper levels.

5.1 Index Maintenance

An index on property p over a set S captures the current state of the objects in S. An update to p for any object in S would require that the index be changed to reflect this update. When creating an operation Op for an abstract data type T, we would like to avoid having to worry about all of the indices that Op might disturb. We solve this problem by using triggers on the *set_p* operation for the appropriate properties.

The model defines triggers as a triple T=(O, C, P) where O is an operation, C is a condition, and P is a program. The semantics of T is that when operation O is invoked on an object x, if C(x) is true P is executed. This view of triggers is less general than some proposals, but it localizes the decision about whether or not a trigger is applicable to the current operation. It is unnecessary to check the condition C unless the associated operation has been invoked.

The only way that an index I on a property p can be invalidated is if a *set_p* operation is performed on the property p for some instance x. Abstract types can be built on top of other abstract types. This produces multiple levels of abstraction. As an example, a stack type can be built on an array type which in turn can be built on top of a bytestring type.

If an index is provided on a property p at any level in an abstract type, we attach a trigger to the set operation on each piece of state (abstract or stored at the next level) on which the property p depends.

In **Figure 1**, we have shown a Set [T] type being implemented by a Set [R] type. The implication in this diagram is that an object of type R has a property r with a value that is indexed by the data structure on the left. An object of type T has a property p that is based on the property of the object of type R (that is acting as the representation for the T type). In this example, the property on the T type is computed as two times the value of the property on the corresponding object of type R. Both sets (Set [R] and Set [T]) have an index. The query optimizer should be able to make use of either of these. For example, if the index over Set [T] was deleted, it still might be possible to use the index on the Set [R].

If we had a query of the form:

```
Select [ST: Set[T], lambda (s) s.p = n]
```

the get_p_value(s) operation would be implemented by 2*(rep(s).r) where rep(s) returns the representation of the object s. Here rep(s) would produce an object of type R. In this case, we can view the query from the point of view of the implementation as:

```
Select [SR: Set[R],
        lambda (s) 2*(rep(s).r)]
```

This transformation is possible since we are essentially substituting the representation type for the abstract throughout the expression. This allows us to substitute Set[R] for Set[S] and the implementation of the body of the predicate in terms of the implementation. The second query expression can now make use of the index on the underlying data structure which is Set[R].

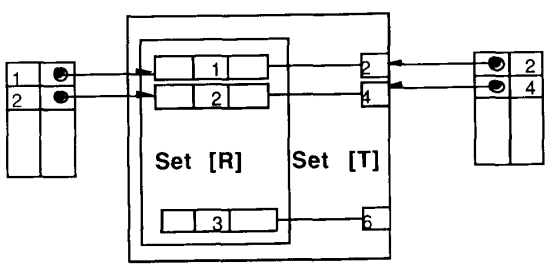

Figure 1 - Indexing at Different Levels of Abstraction

5.2 Indexing User-Defined Types

Normally, indices are mappings between primitive values (e.g., integers, strings) and objects that have these values as values of one of their attributes. How can we build indices in a system in which values are typically of abstract types?

Our approach treats abstract values as indexable by their surrogates or oid's (i.e., object identifiers). This leads to a data structure that might be pictured as in **Figure 2.**

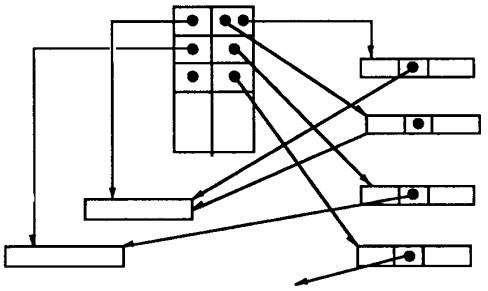

Figure 2 - Index on Structured Objects

The central table in this figure represents an index over a set S for a property p. It contains an entry for each possible value of the property p for each member of the set S. The two rectangles in the lower right represent the values of property p. The rectangles on the right represent the objects in set S. The dark dots represent oid's or pointers to other objects. The entries in the index are all oid values. For a given entry, the left-hand column contains the oid of a property value v (which is a structured object), and the right-hand column contains a list of the the oid's of objects that point to v.

Notice that this structure essentially provides us with a complete set of back pointers for the object set that is indexed. That is, for each object on the left, the index tells us which objects are currently pointing to it. This kind of data structure can be useful for other things in an object-oriented database. For example, it might be useful to know which objects S are pointing to a given object x, so that the objects in X can be updated whenever x is updated. Suppose that x is a component of a complex object. If a new version of x is created, we might want to create a new version of all objects of type T of which x is a part. This index makes this possible.

5.3 An Example of Index Maintenance

Figure 3 is meant to illustrate a situation in which a type T is represented by a type R. R is a record type that contains two fields x and y. T has two properties p and q which are computed as functions of x and y. p is computed as x+y and q is computed as the value of x. The two ovals are instances of the type T showing their R representation inside. The index on the left is computed over the property p for some set containing at least these two instances.

Call the top instance a. If we attempt to update the value of the q property of a to the value 3, we must use an operation invocation of the form *set_p (a.q, 3)*. This would of course translate to an update of the x field of the representation of a. Since the p property also depends on the value of the x field, the derived property p will change as well.

The new value of a.p might not be computed until a *get_p* operation is invoked; however, since there is an index on the p property, we must update that structure immediately so that it will be correct next time it is used. It might be possible to store expressions in the index so that evaluation of the derived

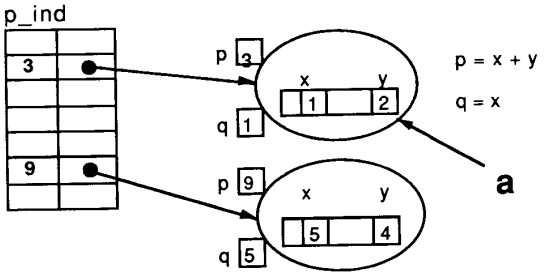

Figure 3 - An Example

properties is delayed until the value is requested. This form of lazy evaluation is beyond the scope of this paper.

Our solution involves placing a trigger T on the x field of the representation. When this field is updated, the following code will be executed.

```
set_x (down (a), 3)              // causes T to fire
T(a) = update_index (p_ind, a, x+y)
```

The set operation on x requires that we convert to the representation type by means of the down operation. The trigger T is attached to this operation. The body of the trigger is given in the second line. It updates the index with the value of x+y.

6. Summary

Although data abstraction presents potential difficulties to the process of query optimization, we feel that it does not preclude it. This paper has presented three different approaches to the problem of query optimiation in an object-oriented database that includes data abstraction. If there are uniformities in the model around the basic notions of aggregation and properties, this can be exploited to produce several generic query transformations. It has further been argued that for types whose methods are written in general-purpose programming languages, we must include additional type-specific semantics and algebraic properties if we are to use them in our optimizations. However, it was shown how methods that are written in very high-level languages like relational algebra can be optimized in conventional ways. More study of the kinds of techniques that were sketched in this paper are needed to better understand what the opportunities and tradeoffs are.

Indexing and auxilliary access paths have played an important role in traditional query processing and will continue to be important in object-oriented databases. This paper has shown how some of these techniques can be extended for use in object-oriented databases with user defined abstract types.

Techniques to allow for more general forms of indexing on general operations is an important area for further study.

7. Acknowledgements

The author wishes to thank David Maier, Gail Shaw, and Michael Stonebraker for interesting comments about the general problem of query optimization in an object-oriented setting. The referees made many useful suggestions for improving the content of the paper. This work was partially supported by grants from Apple Computer, Inc., International Business Machines, Inc., The Office of Naval Research, and US West.

8. References

[BB84] Batory, D. and A. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework," Proceedings of the Conference on Very Large Databases, 1984.

[BKK88] Banerjee, J., W. Kim, and K.C. Kim, "Queries in Object-Oriented Databases",Proceedings of the Fourth International Conference on Data Engineering, Los Angeles, CA, February, 1988.

[BW] Bruce, K. and P. Wegner, "An Algebraic Model of Subtypes and Inheritance," Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September, 1987.

[GD87] Graefe, G. and D.J. DeWitt, "The EXODUS Optimizer Generator," Proceedings of the ACM SIGMOD Conference, May, 1987.

[GHW85] Guttag, J.V., J.J. Horning, and J.M. WIng, "Larch in Five Easy Pieces," Digital Equipment Corporation, Technical Report #5, Systems Research Center, Palo Alto, CA, July, 1985.

[Gu85] Guttag, J., J.J. Horning, and J.M. Wing, "Larch in Five Easy Pieces," Digital Equipment Corporation Technical Report, July, 1985.

[HZ80] Hammer, M. and S.B. Zdonik, "Knowledge-Based Query Optimization," Proceedings of the Conference on Very Large Databases, Montreal, Canada, 1980.

[KRB85] Kim, W., D.S. Reiner, and D.S. Batory, "Query Processing in Database Systems," Springer-Verlag, 1985.

[MS86] Maier, D. and J. Stein, "Indexing in an Object-Oriented Database System, " Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September, 1986.

[MD86] Manola, F. and U. Dayal, "PDM: An Object-Oriented Data Model," Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September, 1986.

[SBG83] Stonebraker, M., B. Rubenstein, and A. Guttman, "Applications of Abstract Data Types and Abstract Indices to CAD Databases," Engineering Design Applications, Proceedings from SIGMOD Database Week, May, 1983.

[ZW86] Zdonik, S.B. and P. Wegner, "Language and Methodology for Object-Oriented Database Environments," Proceedings of the Nineteenth Annual International Conference on System Sciences, Honolulu, Hawaii, January 1986.