Object-Oriented Query Optimization: What's the Problem?

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

Technical Report No. CS-91-41

June 1991

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

Object-Oriented Query Optimization: What's the Problem?

Gail Mitchell and Stanley B. Zdonik
Department of Computer Science
Brown University

Umeshwar Dayal
Cambridge Research Lab
Digital Equipment Corporation

Abstract

An object-oriented database model can support features such as abstract data types, methods, encapsulation, subtyping (or inheritance), complex structures, and object identity. The processing of queries in such a model also entails support for these features. Query optimization will require new techniques for supporting the object-oriented features. Although many of the problems that must be solved by an object-oriented query optimizer are similar to problems solved by relational and extensible optimizers, there are also many problems that are unique to the object-oriented model.

In this paper we explore some of the problems that are encountered when trying to optimize queries in an object-oriented database. We present these problems in the context of the modelling constructs which generate the problem. We also survey current approaches to solving some of these problems and discuss problems that are not addressed by these approaches.

1 Introduction

We are interested in efficiently processing queries in an object-oriented database. A query is an expression describing information one wants to retrieve from a database. Such an expression is normally translated, by a query optimizer, into a series of database operations that yield the desired information.

The ability to do automatic query optimization is one of the strengths of relational database systems. Relational query optimizers exploit the semantics of the model and the fixed sets of operators, storage structures, and implementation techniques for the operators. The designs of such optimizers differ and are specific to the system in which the optimizer is built [Ull89, Dat89]. Most optimization strategies for relational optimizers focus on queries involving the Select, Project and Join operators (see [Ull89] for a discussion of the theory upon which such strategies are based). Further work in relational optimization looked at extending optimization to include other operators, for example aggregators [Kim82, Day87]. Optimization results have also been extended to include more expressive models, such as nested [S$^+$89, Kor88] and network models [RR85, R$^+$83].

Object-oriented databases are extensible systems which support (among other features) abstract data types, type inheritance (subtyping), methods and late-binding, and object identity [MS87, BDK91, ZW86, MD90, B$^+$87]. The extensibility of object-oriented databases is founded on the ability to extend the data model through abstract data types and the inheritance structure. In particular, the ability to define new abstract data types provides extensibility in the database modelled by the types and places a requirement on the database system to support the new model.

In this paper we present some of the problems that arise when optimizing queries in an object-oriented database. We look at problems that arise as a result of supporting different object-oriented features; in particular, abstract data types, complex structures, methods and encapsulation, and object identity. We discuss each problem in the context of the particular feature with which it is

associated. This approach provides important information for designers of database models and languages. It is also important to understand how existing optimization approaches can be applied to object-oriented query optimization and where new approaches are required. We believe that a better understanding of the problems encountered can lead to the development of new (and better) techniques for optimizing object-oriented queries.

The application of algebraic transformations has formed the basis for the design of query optimizers for object-oriented databases [GM88, SO89, Osb88]. A recognized difficulty in applying transformations is the problem of manipulating query expressions containing references to arbitrary methods. Query languages in object-oriented databases are built from operations defined on built-in types (such as Set and Tuple) and on abstract data types, and can therefore access arbitrary methods defined for the types. The costs associated with the application of methods need to be considered when manipulating query expressions to determine database access strategies. We have also found that manipulations involving objects with identity complicate the definition of the equivalence of two query expressions [SZ89a] thus complicating the application of transformation rules.

Support for abstract data types requires an optimizer support extensibility of the model. Current extensible optimizers are based on the transformation of query expressions according to a set of rules which can be augmented as new types, operations, and access techniques are added to the database system. These optimizers provide a fixed repertoire of control strategies for finding and applying these rules to transform a query.

Support for abstract data types and complex structures can lead to query expressions that can not be transformed using context-free algebraic rules. Transformation rules for such expressions need information about the types, and possibly global information about the query expression.

Although many of the problems that must be solved by an object-oriented query optimizer are similar to problems solved by relational and extensible optimizers, there are also many problems that are unique to the object-oriented model. We have identified a number of problems that are encountered when trying to optimize a query expression in an object-oriented database and believe that there is a need for more than a single strategy for transforming queries. We also believe that support for extensibility in the object model will lead to the development of new strategies for optimizing queries involving new abstract data types in addition to new techniques for executing the query.

In the next section we review the database model and algebra we will use to illustrate the problems. In Sections 3.1 through 3.4 we give a detailed discussion of the problems raised above. In Section 4 we discuss some of the current approaches to solving these problems and in Section 5 we conclude with a brief summary.

2 An Object-Oriented Data Model and Algebra

We use the ENCORE data model and query algebra (EQUAL; Encore QUery ALgebra) [SZ89b, SZ90] as the foundation for our discussion of problems in object-oriented query processing. The model incorporates data modelling features found in other object-oriented systems, such as abstract data types, subtyping (inheritance), encapsulation, complex structures, object identity, and late-binding of methods. Thus, problems in dealing with optimization in other models also arise in this model.

The ENCORE model is based primarily on data abstraction [L$^+$77]. An ENCORE type is an atomic type (Int, String, Boolean, etc.) or an abstract data type describing an interface and an implementation for instances of the type. An instance of a type is an *object*. The interface is a logical description of access to instances of the type; it describes methods that can be applied to objects. The implementation describes a physical representation for instances of the type and includes

implementations for methods described by the interface. Abstract data types and encapsulation are synonymous in this model. Objects are encapsulated since all access is through the interface, and the representation of the data and implementation of the methods is hidden. A query is concerned with the interface of a type. However, processing of a query may need to consider the implementation.

Types are related to each other through subtyping. Subtyping requirements ensure that instances of a subtype can be substituted as instances of a supertype . All types are subtypes of type *Object*. This means that all types, and instances of types, are objects with unique identities.

Properties in ENCORE reflect the abstract state of an object. The notion of *property* is modelled by one or more of the methods defined on a type. A property value is accessed by a special observer method which is required to have no side-effects on the observable state of the object. This method for property P is called *Get_P*. For s an object of type T and q a property of T, we represent $Get\_q(s)$ as $s.q$. The Get_P method can return the value of a stored field or it may perform a more sophisticated computation based on the stored representation of the object. A property P may also support another function called *Set_P* that allows the value of P to be changed.

The ENCORE model supports the construction of parameterized types. The parameterized types Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$] and Set[T] are built in. A parameterized type is considered to be a metatype, and the type parameters are input arguments for that metatype's Create operation. For example, the metatype Set[T] defines a Create operation that takes a type T as an input argument and returns a new type as its output.

Parameterized type Set[T] declares T as the type, or supertype, of objects in a collection having type Set, and defines operations *in* and *subset_of* (among others). Type T is called the *member-type* of the set. Type Tuple associates types (T_i) with attributes (A_i) and defines methods *Get_attribute_value* and *Set_attribute_value* for each attribute.

Abstract data types in ENCORE give us the ability to define logical complex structures. The properties of an abstract data type are similar to attributes in systems such as Orion [Kim88] or O_2 [KLR91] in that they give the ability to access state information about an object. The differences between properties and attributes is important to note however. Properties in ENCORE are actually objects, with methods that access their values. The properties of an object define a logical structure for the object. Attributes in other systems are names that have an associated stored value, thus they define a physical structure for objects. Tuple attributes in ENCORE are names with associated methods Get_attribute_value and Set_attribute_value that can compute values associated with the name.

An ENCORE database is a collection of typed objects. The EQUAL query algebra [SZ90] for ENCORE provides type specific operations against collections of objects with identity. We query over collections of type Set[T], where T is some data type, and return new objects having type Set[Q], where type Q is statically determined by the query. Duplication in set membership is determined using object identity; two members of a set cannot be identical.

The algebraic operations support abstract data types. All access to objects in a collection is through the interface defined for the member-type of the collection. The operations are also concerned with object identity; new objects with unique identities can be created and we provide operators to manipulate the identities of objects.

EQUAL includes operations that are the analogs of relational algebraic operations as well as operations to manipulate the logical structure of ENCORE objects. The *Select*, *Project* and *Ojoin* (object join) operations are similar to their relational counterparts in meaning, but generalize the relational operations by applying functions (i.e. property methods or query operations) to the database objects.

The *Select* operation collects a set of database objects satisfying a selection predicate, i.e. $Select(S, p) = \{s \mid (s\ in\ S) \wedge p(s)\}$. The Project and Ojoin operations can create new relationships

not explicitly defined in the properties of the object types. These operations produce tuple objects to store the computed relationships. The *Project* operation creates one tuple for each object in the collection being queried, with the tuple storing selected relationships between components of the object or relationships involving database objects from other collections. For example,

$$Project(Employees, \lambda e\ <(E,e),(M,Select(Managers, \lambda m\ m.salary < e.salary)>) \quad (1)$$

returns, for each employee, all managers who are paid less. The result is a collection of 2-tuples with attribute E of type Employee and M of type Set[Manager]. The value of each attribute E is obtained by applying the identity function (e) to an employee, and the value of each attribute M is obtained as the result of a query over Managers. Project can also act similarly to a relational Project by extracting only properties from the objects in the queried collection.

The *Ojoin* operator is an explicit join operator used to create relationships between objects from two classes in the database. For example, the matching of employees and managers could be accomplished using

$$Ojoin(Employees, Managers, E, M, \lambda e \lambda m\ \ m.salary < e.salary) \quad (2)$$

where e and m represent objects in Employees and Managers, respectively. This result is stored as a collection of 2-tuples with one Employee type attribute and one Manager type attribute.

For sets of tuple objects, *Ojoin* is analogous to a relational *Theta-join*. When a **Set[T]** is involved in an *Ojoin* (for **T** some abstract, non-tuple type), it is treated as a set of single-attribute tuples **Set[Tuple[A:T]]**.

The *Image* operation is like a LISP mapcar; it allows the application of a function (which may be a property or an EQUAL method) to each object in a set, collecting the results in a set object. For example,

$$Image(Managers, \lambda m\ m.secretary) \quad (3)$$

applies the *secretary* property to each manager to yield a set of employees who are secretaries for some manager.

Union, *Difference* and *Intersection* are the usual set operations with set membership based on object identity. The result for all operations is considered to be a collection of objects of type T, where T is the most specific common supertype (in the type lattice) of the types of the objects in the operands.

The *Nest*, *UnNest* and *Flatten* operations work only with the structure of objects. *DupEliminate* and *Coalesce* manipulate identities of result objects. Operation *Flatten* takes a set of sets of objects (type Set[Set[T]]) and returns a set of objects (Set[T]). *Nest* and *UnNest* extend the same operators for non-first normal form relations (see [JS82]) to sets of objects with identity. Sets of tuples can be unnested to convert a set-valued attribute to single-valued, or nested to create a set-valued attribute. Operation *DupEliminate* provides the option of eliminating duplicate copies of objects from a collection. Such elimination is automatic for identical objects (a set cannot have two identical members), but operations that can create new objects (such as Project or UnNest) may create objects that are equal-valued.

Operation *Coalesce* can be applied to a set of tuples, and allows the removal of duplication in tuple attribute values; i.e. manipulates the structure of the tuples to provide aliasing of equal objects. For example, recall query 1 creating, for each employee, a set of managers who are lower paid. For each employee a new set is created to store the matching managers. So, for two equally paid employees, two set objects, each containing the same managers, will be created. Coalescing the result would ensure that employees having the same salary are paired with identical sets of managers.

```
ADT Vehicle
    Extent: Vehicles
    Properties:
        vin: string
        manu_by: Company

ADT Company
    Extent: Companies
    Properties:
        name: string
        emps: Set[Employee]
        depts: Set[Dept]

ADT Dept
    Extent: Departments
    Properties
        mgr: Manager
        emps: Set[Employee]

ADT Address
    Properties:
        street: string
        city: string
        state: string
        zip: string

ADT Person
    Extent: People
    Properties:
        cars: Set[Vehicle]
        residences: Set[Address]

ADT Employee
    Supertype: Person
    Extent: Employees
    Properties:
        employer: Company
        mgr: Manager
        department: Dept
        salary: real

ADT Manager
    Supertype: Employee
    Extent: Managers
    Properties:
        budget: real
        secretary: Employee

ADT Student
    Supertype: Person
    Extent: Students
    Properties:
        gpa: real
        comp: Company
        mgr: Manager

ADT Student-Employee
    Supertype: Student, Employee
```

Table 1: Example scheme.

We use the EQUAL operations and the ENCORE model to illustrate the discussion in the remainder of this paper. The examples in this exposition will refer to the sample scheme of Table 1. The scheme represents a car-manufacturer database (similar to [BKK88]) in which companies have departments, and vehicles are manufactured by companies. There is also a hierarchy of people which includes employees, managers (specialized employees), and students who study at a company under the direction of a manager. Some students are paid while they study (type Student-Employee).

All types are abstract data types and the schemes shown describe methods that can be applied to instances of a type to disclose state information (these are the *Properties*), the inheritance lattice (*Supertypes*), and collections of instances that are maintained by the database (the *Extent*s of the types). Abstract data types could also describe *Operations* that can be applied to objects to modify state, but we omit those here since we want to query without side-effects. The information shown is at the interface of the abstract data type. We do not know how the *Properties* or *Operations* are implemented, nor do we know the physical representation of instances of the type. This information is invisible to a user, since it is hidden by the encapsulation assumed by the model, so we do not reveal it here either.

The Properties of any type in the scheme include those explicitly listed for the type as well as all properties of its supertypes. For example, type Manager has Properties budget, secretary, employer, mgr, department, salary, cars and residences. Type Student-Employee has two different properties of type Company; Property **comp** is the Company inherited from type Student and represents the company at which the student studies, and Property **employer** is inherited from type Employee and represents the company that pays the student. Type Student-Employee also multiply inherits its **mgr** property. We are not concerned with the resolution of that conflict here.

Recall that the Properties of each abstract data type impose a logical structure on instances of the type. Since these are abstract data types, we do not know how the Properties are implemented. For example, the *dept* of a Manager m may be a stored identifier for a department or may involve a computation such as searching through the *Departments* collection to find the one whose *mgr* field is m.

3 Problems in Object-Oriented Query Optimization

We expect the model for an object-oriented database to incorporate abstract data types, subtyping (inheritance), methods, late-binding, encapsulation, complex logical structures and object identity. A query optimizer can encounter a variety of problems when supporting these features, but can also take advantage of some of these features in the optimization process. We discuss some of the specific problems encountered in object-oriented query optimization in this section.

The problems are organized by the object-oriented modelling feature which generates them. In Section 3.1 we discuss general problems that arise in supporting abstract data types and in Section 3.2 we look at problems that are specific to supporting the complex structures that can be built using an abstract type system. In Section 3.3 we discuss the problems related to supporting encapsulated methods. Finally, in Section 3.4 we examine problems that arise when supporting object identity in an object-oriented database.

3.1 Abstract Data Types

In an object-oriented model we need to incorporate optimizations for a changing variety of types. Queries may be based on operations over collections, but optimizations pertaining to sets (or multisets or lists, etc.) need to be combined with optimizations over the types of the objects contained in the sets. Similarly, these optimizations can involve optimizations with the types of objects related to objects in the set (by properties, for example) and so on. An object-oriented query optimizer must be able to apply optimizations specific to the types, and optimizations that look at relationships between objects of different types. Such optimizations are similar to those examined in the area of semantic query optimization [SO87].

For example, suppose we have the query clause

$$e.employer.name = v.manu\_by.name$$

where e is an object of type Employee and v is a Vehicle, and suppose we can define the following axiom (analogous to a functional dependency) for abstract data type Company:

$$\forall c_1, c_2 : Company \quad c_1.name = c_2.name \Longrightarrow c_1 = c_2$$

A type-specific optimization could note that e.employer and v.manu_by both refer to Company objects, and apply the axiom to the predicate to simplify it to e.employer = v.manu_by. This kind of simplification depended solely on the information about the abstract data type. We want to be able to incorporate such transformations into an object-oriented query optimizer. This also means that the optimizer must be able to be extended with new transformations when new types and rules are added to the system.

An object-oriented query optimizer could also have opportunities to apply transformations that use knowledge about the abstract data type construct. For example, consider the query "Find all GM vehicles owned by GM employees". One way to evaluate this query is represented as figure 1a. This evaluation corresponds to the following sequence of algebraic operations:

$$\begin{aligned} GMemps &:= Select(Employees, \lambda e\ e.employer.name = \text{“}GM\text{”}) \\ GMempCars &:= Flatten(Image(GMemps, \lambda p\ p.cars)) \\ Answer &:= Select(Vehicles, \lambda v\ v.manu\_by.name = \text{“}GM\text{”} \\ &\qquad \wedge\ v\ \in\ GMempCars) \end{aligned}$$

The first Select operation creates a set of all GM employees. The Image operation extracts the cars for each employee, resulting in a set of sets which is then Flattened. The final Select chooses,

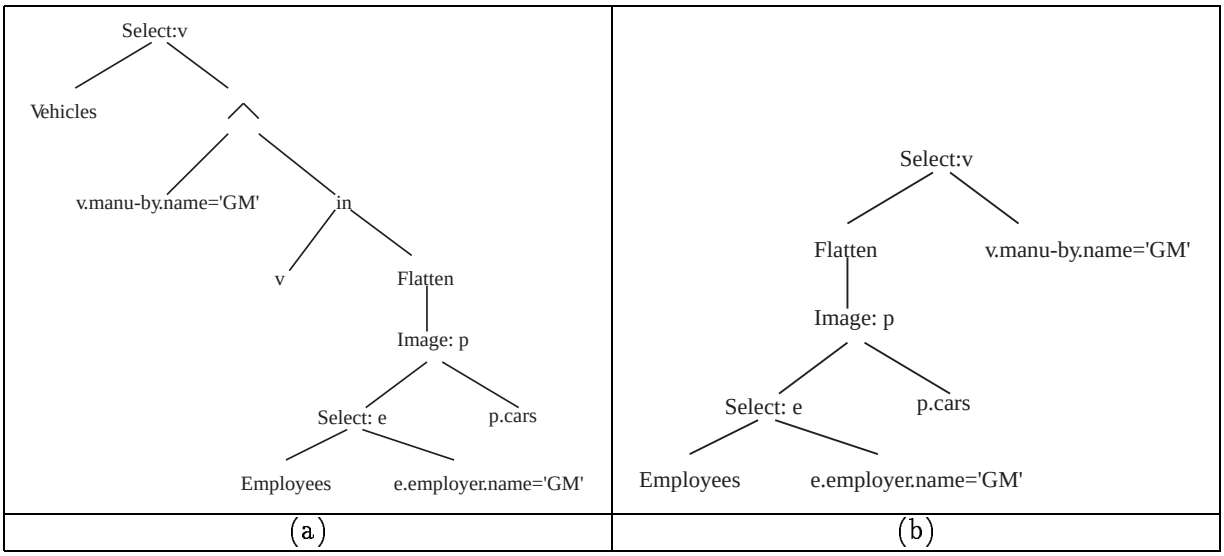

Figure 1: Query — Find all GM vehicles owned by GM employees.

from the set of all vehicles, those that are manufactured by GM and owned by GM employees. We would like the final Selection to be simplified to :

$$Answer := Select(GMempCars, \lambda v\ v.manu\_by.name = \text{“}GM\text{”})$$

This simplification, represented in figure 1b, could result from the knowledge that, in our model, all sets of with member-type T are subsets of the extent of T. Since GMempCars has type Set[Vehicle], it must be a subset of Vehicles (the extent of type Vehicle). This simplification is similar to the domain refinement technique in semantic query optimization [HZ80]. In our problem, however, the simplification is based on knowledge about set inclusion and abstract data types, not necessarily a particular ADT. Subtyping relationships give similar information about set inclusion relationships. For example, in most object-oriented models, the set Managers is a subset of the set Employees. This results because Manager is a subtype of Employee, and the sets are both extents.

Subtyping information can also affect the applicability of transformations in the presence of static type-checking. For example, consider the query Union(Students,Employees). In models without Union types, the application of this set operation, if it is even allowed, would normally result in a set having type Set[Person], where Person is the closest common supertype of types Student and Employee. As a result, only properties of type Person are applicable, as far as static type-checking is concerned, to the result set. This modelling decision means that a query transformation distributing Union (or Intersection or Difference) over other operations is not applicable if static type-checking is required. In other words,

$$Union(QueryOp(S_1, p), QueryOp(S_2, p)) \neq QueryOp(Union(S_1, S_2), p)$$

since p may not be defined for the type of Union(S_1,S_2).

For example, consider a query which asks for the managers of all Students and Employees. This query can be expressed as

$$Union(Image(Students, \lambda s\ s.mgr), Image(Employees, \lambda e\ e.mgr))$$

but not as

$$Image(Union(Students, Employees), \lambda p\ p.mgr)$$

because the *mgr* property is not defined for type Person. However, the property can be applied to the union because of late-binding of the methods to the objects. If the intersection (i.e., objects of type Student-Employee) is large, the latter expression might be the most efficient to process, since applying *mgr* after the Union would avoid applying it twice to objects in the intersection.

These examples illustrate that knowledge about abstract data types and subtyping in the object-oriented model can affect query transformations. In order to support abstract data types an optimizer must be able to incorporate semantic query optimizations, such as those discussed here, while supporting standard query optimizations. An optimizer must also support the extensibility that is inherent in abstract data type construction. The addition of new types results in new operations and transformations that must be considered in the optimization process.

3.2 Complex Structures

The complex structure of objects (whether it be physical or logical) means that languages to query the objects must have mechanisms for exploring their structure. In addition, languages which allow the creation of objects need mechanisms for building new structures. The exploration and creation of such structures can lead to the regular use of path expressions to navigate through a structure. Support for complex structures also results in languages that regularly use nested query expressions, where variables from outer expressions are referenced in nested expressions.

One difficulty with path expressions is the definition of indices for such expressions [MS86, BK89]. The presence of arbitrary methods in the path further complicates this problem. The maintenance of an index can require the application of methods and those methods could manipulate arbitrarily many objects.

Another problem with path expressions is that they imply an execution order for properties on the path. This order may not, however, be the most efficient way to process the query. A path can sometimes be more efficiently processed using Join. Consider, for example, a query involving the path *s.comp.name* where *s* is in Students. It might be more efficient (if, for example, there are very few companies) to first compute the **name** property for each company and store this result in a tuple. The path from a student to a company name would then involve joining Students with the set of tuples by matching the **comp** property of a student with the company attribute of a tuple. An optimizer should be able to make (and evaluate) transformations between path expressions and explicit joins.

A problem with nested query expressions is that query transformations may no longer be local transformations; they can involve query operators that are in different nesting levels. Consider for example a query to find the garaging locations for all Vehicles. The following algebraic expression satisfies the query by building a structure that stores a set of addresses with each vehicle.

$$\begin{aligned} Project(Vehicles, \lambda v < \quad & (V : v), \\ & (G : Flatten(Image(Select(People, \lambda p\ v \in p.cars), \\ & \qquad \lambda p\ p.residences)) >) \end{aligned}$$

This query is illustrated by the tree in figure 2a. In this query the variable representing a Vehicle is referenced in the nested Select query. The $v \in p.cars$ predicate indicates that there is a relationship between Vehicles (represented by variable v) and People (represented by variable p) that could be effected with a Join operation.

The query expression illustrated in figure 2b also satisfies the query. A difference in the solutions is that Project does a left outer join, so could result in vehicles related to empty sets of residences. These could easily be eliminated if necessary.

The two query expressions differ significantly in their use of variables. In the first query, variable reference is nested, while in the second query all variables are local to the operation using them.

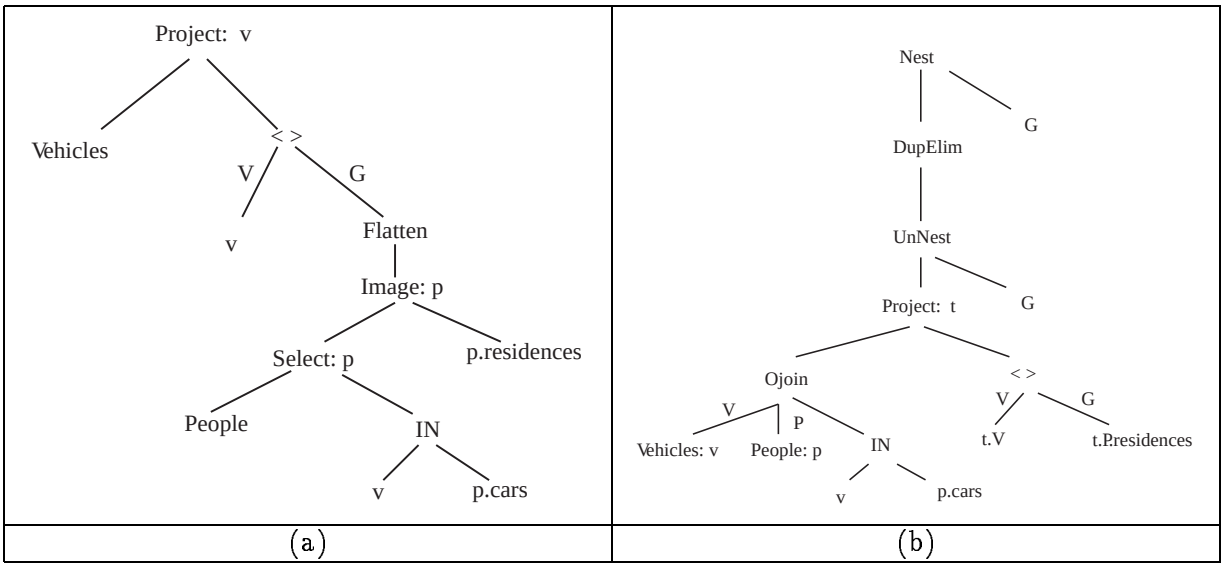

Figure 2: Query – Find all possible garaging locations for each vehicle.

The second query is similar to relational queries in which a Join of all relations assimilates the data, then a Project operation builds the result relation. In this query, the data is basically collected using the Join operation, then the structure for the result is built with the Project, Nest and UnNest operations.

The transformation between the Project and Join versions of a query expression may require more than the application of context-free algebraic transformation rules. The arbitrary nesting of query expressions means that recognition of join relationships involves global knowledge about the expression. In addition, any transformations may have to preserve the structure of result objects. Again, producing such a structure requires knowledge which includes the effects of all parts of the query expression on that structure.

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

3.3 Methods and Encapsulation

The applicability of algebraic transformations to object-oriented queries is complicated by the object model. In general we have found that there are no universally applicable transformations, and that type and storage information about objects is necessary in order to decide the utility of an algebraic transformation. For example, in the relational model, pushing a Select operation past a Join operation is generally accepted to be a useful transformation. However, in the object-oriented model, the cost of applying a Selection operation is a factor when deciding whether to push the Select past a Join [Del89]. The presence of methods in a Selection predicate means that the cost of applying that operation depends on the cost of applying the methods.

Consider, for example, a query which matches managers who have budgets of over one million dollars with students studying at the same company who have grade point averages of more than

3.5. An SQL-like version of this query is:

```
Select   [RichManager: m, GoodStudent: s]
From     m in Managers
         s in Students
Where    m.budget ≥ 1000000 AND
         s.gpa ≥ 3.5 AND
         m.employer = s.comp
```

Applying relational heuristics to this query could give an expression which joins Managers with budgets of over one million with Students who have appropriate grade point averages, then Selects over those pairs by matching the companies. Such a query expression results from strategies that push selections (on budget and gpa respectively) to the single relations to which they apply in an attempt to reduce the sizes of collections to be joined. The heuristic used is that joining is an expensive operation, but selections are straightforward or could even be done as part of the joining process.

However, consider the case where computing a manager's budget is an expensive operation. In this situation it could be more efficient to first join Students (perhaps selected by gpa, depending on the expense of that operation) with Managers by matching the companies. The join operation could reduce the number of managers to which the *budget* method must be applied.

This type of situation is certainly present in relational systems, but the presence of methods in the object-oriented model — with possibly arbitrary computations implementing the methods — means that strategies such as pushing Selection past Join will be less often applicable. We claim that a better strategy for object-oriented models is to always consider cost in the application of transformations.

The determination of the cost of a query, or query operation, is complicated by the presence of methods and encapsulation. Although method cost needs to be considered when transforming a query expression, a query optimizer needs a way to ascertain method cost in the presence of encapsulation. If the optimizer is allowed to break encapsulation and look at the implementation of a method, then it must be able to understand that implementation. For example, a method could be written in the query language understood by the optimizer. The method code could then be merged with the query and managed by the query optimizer [BK90b]. This approach, of course, limits the expressibility of methods to that of the query language.

If the optimizer gathers cost information about a method by querying the method itself [GM88] then type Method must define an interface that can provide the required information. For example, in the simple case, type Method would have a property named **cost** which, when applied to a method instance, would return an expected cost for executing the method.

Late-binding of methods also complicates the determination of method costs, since the implementation (and therefore the cost) of a method used in a query will not necessarily be known until query execution time. For example, suppose the gpa method for type Student directly accesses values in the representation of Student but the method is overridden in type Student-Employee by a more expensive method (perhaps computing an average using Student and Company information). The cost of applying the gpa method to members of Students can not be statically determined since some objects in the set could actually have type Student-Employee.

3.4 Object Identity

The identity of objects involves modelling and language decisions that can affect the optimization of queries. When objects have identities, there is a question as to what constitutes equality of two objects (see for example [KC86], [SZ89a]). This carries over to the language, where equality

operations are used in predicates and where a decision must be made concerning the creation of new objects by a query. The creation of new objects can lead to new definitions for equivalence of queries that affect the transformations available to an optimizer. An optimizer for object-oriented models must be able to deal with the creation of new objects and with alternative definitions for equivalence.

Equality of objects in a query can refer to any definition of equality for type Object (e.g., identical, deep-equality) or to equality operations defined for a particular abstract data type. In a query language a variety of equality operations could be permitted in predicates. For example consider the operation $a.cars = b.cars$. Equality here might refer to identical sets (i.e. a and b reference exactly the same set of cars), to shallow equal sets (the sets of cars referenced by a and b contain exactly the same cars), or possibly even to sets with the same cardinality (if = is defined as such for type Set[Car]). In the presence of object identity, an equality operation is actually a method, and will have to be treated as such by an optimizer.

A major language decision that affects the optimization of queries is whether a query language can create new objects and, if so, whether those objects can have identities. If the language does not create objects, new relationships between objects cannot be built by queries. If the language only creates objects without identities, then it must be able to work with those objects as well as with objects with identity. On the other hand, if the language creates objects with identity then it must have mechanisms to determine new identities and manipulate those identities. For example, the EQUAL query language provides DupEliminate as an operation to manipulate unwanted duplication in objects that may be created by a query.

The creation of new objects with identity by a query can complicate the optimizer's task. For one thing, the optimizer may want a mechanism for deciding whether to create identities for objects in intermediate stages of a query. Also, the creation of objects with identity complicates the meaning of equivalence of two queries. The result of a query can be a new collection of objects with a unique identity, and as a result even two responses to exactly the same query are not identical. The creation of new objects by a query language means that the structure of results as well as the data retrieved by a query must be considered when defining equivalence.

We have defined alternate notions for equivalence (see [SZ89a]) that must be recognized by an optimizer working with a language that builds objects with identity. For example, our weakest notion of equivalence states that two queries are equivalent if they respond with the same data, regardless of the logical structure of the objects returned. The query "For each student find all employees in the same department" could be answered by an Ojoin, returning Student/Employee pairs, or by a Project (followed by a Select to eliminate students matched with no employees) returning Student/Set[Employee] pairs.

The ability to define these alternatives might allow the optimizer to choose a more efficient method for solving a query. It also might give the optimizer more latitude when working with nested queries. For example, consider the query:

```
Select e
From   s in Students
       e in Employees
Where  e.mgr = s.mgr
```

giving all employees who work for someone who also manages a student. A straightforward translation of this query to an algebra could give:

$$Project(Ojoin(Students, Employees, S, E, \lambda s \lambda e\ s.mgr = e.mgr), \lambda t < (Emps, t.E) >)$$

Note that this translation assumes that the type to be returned is Set[Tuple[Employee] rather than Set[Employee]. The assumption implies that a given employee can be represented many times in

the result, once for each student with which it is associated in the Join. If an Image operation is chosen instead of the Project the result would not contain duplicates of individual employees. An object-oriented query optimizer should have the ability to make such decisions whenever they are appropriate since such transformations might be useful in providing further opportunities for optimization.

For example, an optimizer that can deal with weakly equivalent transformations can transform the previous query by substituting a Project operation for the Ojoin giving

$$\begin{aligned} &Project(Select(Project(Students, \lambda s \\ &\qquad < (S, s), (E, Select(Employees, \lambda e\ s.mgr = e.mgr)) >), \\ &\qquad \lambda t\ NOT\ Empty\ t.E), \\ &\quad \lambda t < (E, t.E) >) \end{aligned}$$

A further transformation could commute the outer Project and Select, since the Project retains the information needed for the Select:

$$\begin{aligned} &Select(Project(Project(Students, \lambda s \\ &\qquad < (S, s), (E, (Select(Employees, \lambda e\ s.mgr = e.mgr) >), \\ &\quad \lambda t < (E, t.E) >, \\ &\quad \lambda t\ NOT\ Empty\ t.E)) \end{aligned}$$

The two Project operations can then be collapsed into one giving:

$$\begin{aligned} &Select(Project(Students, \lambda s < (E, Select(Employees, \lambda e\ s.mgr = e.mgr)) >) \\ &\quad \lambda t\ NOT\ Empty\ t.E) \end{aligned}$$

The final result performs fewer operations than the original query, and also is concerned with fewer objects since there are no intermediate results. On the other hand, the final result has a different type than the initial query expression (Set[Tuple[Set[Employee]]] as opposed to Set[Tuple[Employee]]). The change in type information occurred when the Project was transformed to Ojoin using the weak equivalence. An optimizer should be able to apply weaker equivalences to find better solutions, but must then be able to determine when such transformations are acceptable. This might involve the ability to restore the type of a result.

4 Current Research

Query optimizers and optimization techniques are based on a particular data model and language for that model. Query optimization was important to the success of the relational model, and much of the current work in query optimization for extensible and object-oriented systems is based on relational optimization results.

Relational optimizers make extensive use of heuristics based on logical query transformation rules, such as commutativity of join and pushing select past join. A rule-based system for optimizing relational algebra expressions was proposed by Smith and Chang [SC75]. They use rules and algorithms for tree transformation that are based on heuristics for the relational model. For example, they have rules to move selection and projection to the leaves of a query tree. Much of the work in query optimization in extensible and object-oriented systems is based on such rules about algebraic transformations in the query language.

A need for extensibility in data modelling and access led to the development of databases in which such aspects as the data types, operators, and access methods can be extended [H$^+$89, SR86, B$^+$90, CD$^+$86]. Extensible database systems often use algebraic rules about query operations to provide transformations that can be applied to queries to generate expressions that will be more

efficient to process. The extensibility in the system is supported by the ability to add new rules as new data types and operators are added to the system. Optimizers for these systems are extensible in the sense that they accept the definition of new types and rules. The new rules are, however, processed by the optimizer in the same way as existing rules.

Optimization techniques and results in relational and extensible databases form the basis for current research in object-oriented query optimization. The Gemstone database, for example, focuses on indexing for processing of queries that select from a class based on a predicate [MS86]. Current research on optimization in Orion is directed toward the adaptation of relational techniques to the object-oriented database [J+90]. In particular they explore efficient alternatives to object navigation. Kemper and Moerkotte explore the use of relations to support navigation [KM90].

A variety of proposals have been made for optimization based on algebraic transformation rules for query operations[LB89, Osb88, Zdo89, SO89, BK90b]. For example, Straube and Ozsu [Str90] propose a methodology for query optimization that includes calculus-based optimization, the transformation of calculus-based queries to algebraic form, type-checking of algebraic expressions, the application of algebraic transformation axioms, and the generation of access plans for the algebraic operations. Beeri and Kornatzky [BK90b] assume there exists a rule-based optimizer and provide an extensive set of rules for bulk data types, thus generalizing existing axioms for object-oriented algebras.

Other work in object-oriented query optimization has concentrated on problems that are more specific to object-oriented models. Research on optimization in O_2 [CD90], for example, combines the Orion results with techniques that include the factorization of common subexpressions, cost-based application of query rewrite rules, and the use of indexes and clustering to guide the query rewrite process and aid in determining access plans. These techniques are concerned, in particular, with accessing complex structures and the processing of nested query expressions.

Optimization in the presence of encapsulation and methods is a recognized problem that is still being studied [BK90a]. Graefe and Ward [GW89] propose to statically generate query evaluation plans with alternatives. Information available about objects is used at execution time to choose among the alternatives and generate a final evaluation plan. Graefe and Maier submitted a preliminary architecture for an object-oriented optimizer that sends messages to methods to ask them to "reveal" information about their execution [GM88]. The revealed information is used to expand the nodes of a query tree with execution information. The query tree, when fully expanded, would be input to a rule-based optimizer such as an EXODUS optimizer [GD87].

The variety of problems encountered when optimizing an object-oriented query, and the different approaches to solving these problems, are addressed by our current work in query processing [MZD]. We propose a new approach to optimizer extensibility which supports extensions to the optimizer control strategies and techniques for query transformation in addition to extensions that support the data model. This approach is motivated by the identification of the problems discussed in this paper and the different strategies that can be employed to solve some of these problems. We believe that the extensibility of the object-oriented database model will lead to the identification of new problems in query optimization and, hopefully, new techniques for solving these problems. Our architecture for query optimization will allow the addition of new optimization strategies as they are developed.

5 Summary

The optimization of queries in an object-oriented database entails support for the features of the object-oriented data model. In this paper we have noted some of the optimization problems that accompany support for abstract data types, complex structures, encapsulation and methods, and object identity. These problems need to be considered by designers of database models and lan-

guages, since support for a feature will require solutions to the problems accompanying that feature. Solutions to some of the problems we have identified will also require optimization techniques that go beyond current relational optimization technology. An object-oriented query optimizer must be able to incorporate techniques for solving the different problems. We have presented these problems and examples as a first step toward organizing the open issues in object-oriented query optimization.

References

[B+87] Jay Banerjee et al. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[B+90] D. S. Batory et al. GENESIS: An Extensible Database Management System. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 500–518. Morgan Kaufmann, San Mateo, CA, 1990.

[BDK91] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object Oriented Database System: the Story of O_2*. Morgan Kaufmann, San Mateo, CA, 1991.

[BK89] Elisa Bertino and Won Kim. Indexing Techniques for Queries on Nested Objects. Technical Report ACT-OODS-132-89, MCC, 1989.

[BK90a] François Bancilhon and Won Kim. Object-Oriented Database Systems: In Transition. *SIGMOD Record*, 19(4):49–53, Dec 1990.

[BK90b] Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proceedings ICDT*, Paris, France, 1990.

[BKK88] Jay Banerjee, Won Kim, and Kyung-Chang Kim. Queries in Object-Oriented Databases. In *Proceedings 4th Intl. Conf. on Data Engineering*, pages 31–38. IEEE, Feb 1988.

[CD+86] Michael J. Carey, David J. DeWitt, et al. The architecture of the exodus extensible dbms. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pages 52–65, September 1986.

[CD90] Sophie Cluet and Claude Delobel, May 1990. unpublished manuscript.

[Dat89] C. J. Date. *An Introduction to Database Systems*, volume I. Addison-Wesley, Reading, MA., 5th edition, 1989.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the 13th VLDB Conference*, pages 197–208, 1987.

[Del89] Claude Delobel, May 1989. Personal communication.

[GD87] Goetz Graefe and David J. DeWitt. The EXODUS Optimizer Generator. In *SIGMOD Proceedings*, pages 160–172. ACM, May 1987.

[GM88] Goetz Graefe and David Maier. Query Optimization in Object-Oriented Database Systems: A Prospectus. In *Advances in Object-Oriented Database Systems*, pages 358–363. International Workshop on Object-Oriented Database Systems, September 1988.

[GW89] Goetz Graefe and Karen Ward. Dynamic Query Evaluation Plans. In *SIGMOD Proceedings*, pages 358–366. ACM, June 1989.

[H+89] Laura M. Haas et al. Extensible Query Processing in Starburst. In *SIGMOD Proceedings*, pages 377–388. ACM, June 1989.

[HZ80] Michael Hammer and Stanley B. Zdonik, Jr. Knowledge-based query processing. In *Proceedings of the 6th VLDB Conference*, pages 137–147. ACM, 1980.

[J+90] B. Paul Jenq et al. Query Processing in Distributed ORION. In *EDBT*, pages 169–187, 1990.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proceedings of the Symposium on Principles of Database Systems*, pages 124–138. ACM, March 1982.

[KC86] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416. ACM, September 1986.

[Kim82] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, Sept 1982.

[Kim88] Won Kim. A Model of Queries for Object-Oriented Databases. Technical Report ACA-ST-365-88, MCC, 1988.

[KLR91] Paris Kanellakis, Christophe Lécluse, and Philippe Richard. Introduction to the Data Model. In Bancilhon et al. [BDK91].

[KM90] Alfons Kemper and Guido Moerkotte. Access Support in Object Bases. In *SIGMOD Proceedings*, pages 364–374. ACM, 1990.

[Kor88] Henry F. Korth. Optimization of Object-Retrieval Queries. In *Advances in Object-Oriented Database Systems*, pages 352–357. International Workshop on Object-Oriented Database Systems, September 1988.

[L+77] Barbara Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[LB89] Cesar Galindo Legaria and Renato Barrera. A Rule-Based Query Optimizer. Technical Report 92, Centro de Investigacion y de Estudios Avanzados Del IPN, 1989.

[MD90] Frank Manola and Umeshwar Dayal. PDM: An Object-Oriented Data Model. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, CA, 1990.

[MS86] David Maier and Jacob Stein. Indexing in an Object-Oriented Database. In *International Workshop on Object-Oriented Database Systems*, pages 171–182, 1986.

[MS87] David Maier and Jacob Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[MZD] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. An architecture for query processing in persistent object stores. In preparation.

[Osb88] S. L. Osborn. Identity, Equality and Query Optimization. In *Advances in Object-Oriented Database Systems*, pages 346–351. International Workshop on Object-Oriented Database Systems, September 1988.

[R+83] Daniel R. Ries et al. Decompilation and Optimization for ADAPLEX: A Procedural Database Language. Technical Report CCA-82-04, Computer Corporation of America, Sep 1983.

[RR85] Arnon Rosenthal and David S. Reiner. Querying relational views of networks. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 109–124. Springer-Verlag, 1985.

[S+89] G. Saake et al. Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype. In *Proceedings of the 15th VLDB Conference*, pages 307–316, 1989.

[SC75] John Miles Smith and Philip Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Database Interface. *Communications of the ACM*, 8(10):568–579, Oct 1975.

[SO87] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. A System for Semantic Query Optimization. In *SIGMOD Proceedings*, pages 181–195. ACM, 1987.

[SO89] Dave D. Straube and M. Tamer Özsu. Query Transformation Rules for an Object Algebra. Technical Report CS-89-23, University of Alberta, 1989.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *SIGMOD Proceedings*, pages 340–355. ACM, 1986.

[Str90] Dave D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, Univ. of Alberta, December 1990.

[SZ89a] Gail M. Shaw and Stanley B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 264–278, December 1989.

[SZ89b] Gail M. Shaw and Stanley B. Zdonik. An Object-Oriented Query Algebra. In *Proceedings of the 2nd International Workshop on Database Programming Languages*, pages 103–112, June 1989. Reprinted in *IEEE Data Engineering Bulletin*,12,3, September 1989.

[SZ90] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pages 154–162. IEEE, 1990. An early version of this paper appears as Brown University tech report CS-89-19.

[Ull89] Jeffrey D. Ullman. *Principles of Database And Knowledge-Base Systems*, volume II. Computer Science Press, 1989.

[Zdo89] Stanley B. Zdonik. Query Optimization in Object-Oriented Database Systems. In *Proceedings of the Hawaii International Conference on System Science*, January 1989.

[ZW86] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.