Query Processing Issues in Object-Oriented Database Systems — Preliminary Ideas*
(Invited Paper)

M. Tamer Özsu

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
ozsu@cs.ualberta.ca

Abstract

Object-oriented database technology is being developed to provide support for advanced applications such as engineering databases, office information systems, and knowledge bases. Successful deployment of these systems requires the provision of full set of features and capabilities found in traditional database managers (e.g., relational systems). An important such feature is a declarative query language and its associated query processor. Development of query models and design of query processors is well understood in traditional database management systems. However, the task is complicated in object-oriented systems due to the richness of the data model. The query models necessarily become more complex and their optimization significantly more difficult. In this paper we discuss the issues that need to be considered in the development of query models and in the implementation of query processors in object-oriented database systems.

1 Introduction

Object-oriented database (OODB) technology is expected to have a significant impact on advanced database applications such as engineering databases, office information systems, etc. Early research in the area has concentrated on the modeling capabilities resulting in the development of a number of data models [15, 30, 35, 43, 45]. Unfortunately there is no commonly accepted data model formalization. Each model differs in its formalism, support for object identity [27], encapsulation of state and behavior [46], type inheritance [8] and typed collections.

To be successful, an object-oriented database management system (OODBMS) has to provide all the functionality found in traditional database managers. These functions include a declarative query language, transactions, view management and so on. This paper concentrates on one of these issues: the design of query languages, their formalization, and their processing in OODBMSs. We should immediately note that it is too early for an exhaustive discussion of these issues. Work on query models and query processing in OODBMSs is quite recent and all of the relevant issues have not yet been uncovered. In this paper we restrict ourselves to the issues that have been addressed in our own research [50, 51]. We also heavily rely on [56] which provides a framework for evaluating query models, specifically object algebras. The ideas in this paper should be treated as preliminary, requiring significant more research to work out the details.

Our previous research investigated the feasibility of a query processing methodology as depicted in Figure 1, which is an extension of relational query processing strategies. The steps of the methodology are as follows. Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first reduced to a normalized form by eliminating duplicates, applying identi-
ties and rewriting [24]. The normalized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the extents of types in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested function. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence preserving rewrite rules [16] to the type consistent algebra expression. Lastly, an access plan which takes into account object implementations is generated from the optimized algebra expression.

The above methodology forms the framework of this paper. In the next section we discuss object model issues. In Section 3, we discuss query model issues including the definition of a calculus and an algebra. Optimization of queries, which covers query rewrite rules, the generation of alternative execution plans and the selection of the “best” execution plan are topics of Section 4. Finally, in Section 5, we provide some concluding remarks on the suitability of the methodology and the remaining open problems. Our presentation assumes some familiarity with object-oriented concepts and terminology.

2 Data Model Issues

The power and flexibility of object-oriented systems introduce considerable complexity into their data models. Even the feasibility of defining an “object-oriented data model” in the same sense as the relational model is in question [33]. A comprehensive discussion and treatment of all these issues is beyond the scope of this paper. The discussion in this section is restricted to those data model issues that relate directly to the definition of query languages and their processing.

The following aspects of the data model have a direct bearing on the query model and the capabilities that need to be included in a query processor. These issues are not entirely independent of each other. We illustrate how one design decision can affect others.

Nature of an “object”. There are different definitions of an “object”. Some data models consider objects simply as complex data structures [4, 30, 40], somewhat similar to the nested relation models that permit relation-valued attributes. This approach is common to those models that are developed to deal with complex object structures as they exist, for example, in engineering applications. Other data models consider objects to be instances of abstract data types (ADTs) [3, 45], which encapsulate the representation of the objects together with a set of public methods that can be used to access them. In this case, the type is a template for its instances.

The variation in the level of encapsulation enforced by data models affects query models in the following sense. The query model must fully describe the visible components of objects which can be accessed by query primitives. For example, if objects are tuple-valued as in [6], then query expressions can directly access tuple fields by name. Furthermore, the allowable query primitives are dependent upon this decision. In principle, maintaining the data abstraction paradigm would require querying the database based on object behaviors, not their structure. Complete encapsulation, therefore, would require that the comparison operators in the query language be based only on identity (“are two objects the same?”) not on structure. There are query models that provide a relaxed form of encapsulation by enabling some sort of structure-dependent equality check.

Strong versus weak typing. If the data model treats objects as instances of ADTs, then does it require that each object be of a given type? Most object data models require objects to belong to predefined types, but there are others which permit one-of-a-kind
objects (also called prototypical objects) that define their own types and are not associated with any predefined type in the system. Some models also permit variant objects that belong to a type, but vary from the template defined by the type in some manner. In systems that allow variant and prototypical objects (e.g., [31, 53]), it is no longer possible to specify the full behavior of each object based on its type and this influences the types of optimizations that can be performed.

Furthermore, the definition of the "schema" when prototypical and variant objects are supported needs to be clearly worked out. When variant objects are allowed in the data model, the type is no longer a template for its instances as we claimed before; it is only a minimal template. Therefore, it only defines the minimal behavior of the objects that belong to this type. This is important because if a query language "takes as input a schema (and a database) and generates as output another schema (and another database)" [5] then the definition of the schema determines part of the query input.

Uniformity of the model. Is everything in the system an object? Some data models (e.g., OODAPLEX [12], FUGUE [21], FROOM [36]) treat objects such as methods, and anything else that can be defined in the system. Such models bring uniformity to the treatment of objects. This is in contrast to other models where concepts such as methods and types are treated as meta-information separate from objects.

The uniformity of the object model affects the query model in various ways. It clearly provides more flexibility in querying what is basically meta-information. This could provide possibilities for their manipulation via the query formalism. Furthermore, if, for example, methods are objects, then the query model should be able to handle them (i.e., invoke them, access their bodies, etc). As observed in [30], this requires language capabilities that allow method invocation, supplying parameters to these methods and being able to deal with the returned results. There requirements make the query formalism more powerful, but also more complex.

Single versus multiple types. Objects can either belong to a single type or to multiple types. If two types such as Employee and Student are defined, then data models which restrict objects to be of a single type would force a Teaching Associate to belong to one of these types. In other data models, objects can be of multiple types at once, allowing a Teaching Associate to be both an Employee and a Student at the same time. This may be a more flexible and, in some sense, more natural representation of the real world, but it is also more difficult to handle in a query model since it is necessary to deal with objects that have multiple behaviors.

We should note that belonging to multiple types in this context does not imply a subtype/supertype relationship. Type A is said to be a subtype of type B if the behavior of A (as defined by its methods) is included in the behavior of B. B, in this case, is called the supertype. Subtyping establishes an "IS-A" relationship between A and B: "A is a B." Thus, by definition, any object of type A is also an object of type B. The subtype/supertype relationships between types form a hierarchy (actually a lattice) such that a parent type in the lattice is a supertype of all its children types. This is not what we mean by belonging to multiple types. This relationship can be used, however, to model objects which need to be of more than one type. Returning to the example, Employee and Student types are not in a subtype/supertype relationship. To model a Teaching Associate as both an employee and a student a TeachingAssociate type is created with two supertypes: Employee and Student.

The difference between a data model that allows objects to belong to multiple types and one which "simulates" the same effect by creating subtypes is subtle. In the latter, there is a new type in the type lattice which becomes part of the schema; in the former, no such type definition exists in the schema. If we assume the existence of a system-defined hyType function that maps an object to its type(s), in the example that we are considering, the result of applying this function to an object which represents a Teaching Associate would be different in the two cases. hyType would return {Employee, Student} if objects can belong to multiple types, but its result would be {TeachingAssociate} if explicit subtypes are created.

Classes versus collections. Some data models have the concept of a "class" as an extension of a type. Therefore, a class represents the set of instances of a type. In these models, the distinction between a class and a type become fuzzy. In other data models, there is no explicit notion of a class (or class is not the only collection of objects of a given type) and objects are grouped in arbitrary collections. Therefore, queries can be specified on any one (or more) of these collections. The difference from the perspective of query models is the following. Let us assume that there is a type Employee with properties Name,
Salary, Department. In the first case, there is an Employee class over which queries are defined. Thus, a query which asks for names of employees in the shoe department will retrieve all employees who satisfy the predicate. In the second case, however, no Employee class exists. Maybe there are explicit collections such as BostonEmployees, KansasEmployees, TorontoEmployees and the query either has to be defined over one of them (resulting in the retrieval of only the employees in that collection) or the user has to explicitly specify the query on all three collections and union the results. Note that this distinction is important in the context of defining what the schema is. If the data model treats these collections uniformly as objects, and a relationship is established between these collections and the type lattice, then the schema can be defined as consisting of the type lattice and an object collection lattice which are connected at the root via a system defined Object object. This would enable the specification of queries on types (as well as on individual collections), which are then executed on all the collections that are of that type.

It has been pointed out that data models that employ "class as the extent of type" approach make it easier to conceptualize a query [28]. Furthermore, it is possible in these models to exploit the subtype/supertype relationship [56]. If, by definition, an object of type A is also of type B when A is a subtype of B, then the result of a query which asks for all objects of in a class of type B can also include objects of a class of type A. This can be extended to multiple levels of subtype/supertype lattice. Thus, the query result is the union of all objects that are in all the classes of types in the subtree of the type lattice rooted at the type (class) at which the query is posed. We call this the deep extent of a class (type) in [51].

Mechanism for sharing. One of the strengths of object-oriented models is that they provide mechanisms for sharing among objects. Two types of sharing are possible: sharing of implementation and sharing of behavior. Behavioral sharing is what we called subtype/supertype relationship above. It is important to differentiate this from implementation sharing. As noted by Nierstrasz [39], "many of the 'problems' with inheritance arise from the discrepancy between these two notions." He goes further and associates subtyping with types and inheritance with classes. This association is certainly valid in models that support "classes as the extent of type" approach. Unfortunately, early object-oriented languages such as Smalltalk [17] bundle these two concepts.

There have been two proposals for implementing sharing, one based on inheritance (e.g., Smalltalk) and the other based on delegation [31]. In inheritance, the sharing is based on a lattice of types. Thus, type A which is a child of type B in the hierarchy, inherits (behavior or implementation) from B. In delegation, sharing is achieved by an object explicitly delegating its behavior or implementation to another object. By and large, the choice of the sharing mechanism has been tied to the choice of the type system. Those models which allow variant and prototypical objects typically implement delegation as the sharing mechanism, while those that enforce strong typing implement inheritance. There has been some recognition that the two mechanisms are similar [47, 48], but the semantics of a data model that permits prototypical and variant objects, but implements inheritance as the sharing mechanism is quite complicated. This is especially the case if the data model does not treat types uniformly as objects in which case the specification of inheritance between types and the inheritance between instances of those types (which are different due to the existence of variant objects) and the link between the two is bound to be quite complex. These complications affect query processing very directly since the optimization of queries in an object-oriented system can and should take advantage of the semantics of sharing.

3 Query Model Issues

There are too many issues to consider in designing query models. In this paper, we cannot discuss all of these issues. We will concentrate on a few of the more important issues and direct the reader to [56] for a more comprehensive discussion, especially of object algebras.

We concentrate on three key trade-offs of OODB query facilities: (1) formal vs ad hoc query languages, (2) predicates based upon structure vs behavior, and (3) object-preserving vs object-generating operations. We also discuss a number of other design issues.

Formal versus ad hoc query languages. Formal query languages [40, 45, 51] have several properties not found in ad hoc query languages [15, 35] making them more suitable for formal analysis. Most importantly, their semantics are well defined which simplifies formal proofs about their properties. Common types of formal query languages are a calculus or an algebra. A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in an algebra are procedural in nature but
can be optimized. Algebras provide a sound foundation for rule-based transformation systems [16, 18, 20] which allow experimentation with various optimization strategies. A large body of work exists on algebras for other data models (see, for example, [1, 23]). Defining OODB query requirements formally in terms of an algebra facilitates comparisons with these other models.

An important aspect of formal query languages is whether or not they support a calculus definition (in the sense of the relational calculus). If declarative languages are to be provided at the user interface, there is a need to define a formal object calculus. We have defined such a calculus in [51], but calculus definitions are typically lacking in object-oriented query research.

Definition of a calculus raises a number of interesting issues. The notion of completeness (in the same sense as relational completeness) has to be worked out, since it influences the set of algebraic operators. Completeness requires the calculus and the algebra to be equivalent. Safety of calculus expressions is also an issue that needs to be worked out. Safe expressions guarantee that queries retrieve a finite set of objects in finite amount of time [41]. Finally, efficient algorithms need to be developed to translate safe calculus expressions to algebraic ones. Our work reported in [51] defines a "restricted" calculus (therefore is only partially complete) and gives a translation algorithm to an object algebra which we also define.

Predicates based upon structure versus behavior. As we discussed in the previous section, some object models implement complex objects whose internal structure is visible while others view objects as instances of abstract data types. Access to objects which are instances of an ADT is through a public interface. This interface defines the behavior of the object. Although the two views of objects appear incompatible, the ADT approach can effectively model complex objects by including set and put methods for each of the components of the internal structure [57]. Thus, a query language which supports predicates based on object behavior is more general while still allowing knowledge of object representations to be introduced in a later stage of query processing.

Object-preserving versus object creating operations. A distinction can be made between object-preserving and object-creating query operations [43]. Object-preserving query languages [2, 3, 35] return objects which exist in the original database. Object-creating languages [28, 30, 40, 45] answer queries by creating new objects from other objects. The new objects have a unique identity and some criteria is used to appropriately establish their supertype/subtype properties. In one sense this violates the integrity afforded by objects with identity as objects with no apparent relation to each other can be combined and presented as a new object which encapsulates some well defined behavior. But the requirement for combining objects into new relationships does exist; either for output purposes or for further processing as in knowledge bases where knowledge is acquired by forming new relationships among existing facts.

Notice that any OODB query language must have a complete object-preserving query facility independent of whether it additionally creates new objects. The ability to retrieve any object in the database independent of whether it additionally creates new objects. The ability to retrieve any object in the database utilising relationships defined by the inheritance lattice or defined by ADT operations on objects is a fundamental requirement. The addition of object-creating operations adds to the power of the language, but also raises a number of issues such as the type of the created objects and the operations that they support.

Closed versus open algebras. One of the strengths of the relational algebra is that it is closed so that the output of one operation can become an input to the next. Extension of this concept to object algebras is considered "highly desirable" [5]. Closure is somewhat more complicated in OODBMSs, however. The simplifying factor in relational systems is that the operand(s) as well as the result of any algebraic operation are relations. Thus, all operators have one type of input and generate one type of output: relation. In object-oriented systems, the schema consists of many types. Thus, closure property has to be redefined to handle the multiplicity of types. A closed object algebra consists of operators each of which operates on set(s) of objects of a type in the type system and outputs a set of objects that is of an existing type in the type system. As observed in [5], most object-oriented languages are "able to map structured objects into other structured object. However, the objects returned do not necessarily belong to any of the existing types."

Note that the existence of object-creating algebra operators, by definition, complicates closure. The provision of heterogeneous collections as outputs of queries is also difficult to reconcile with closure. The issues relate to the determination of the type of objects in the collection which we address next.

Type of the objects in the query result. It is possible for some object algebra operators to produce results that consist of a set of heterogeneous objects.
(more precisely, a set of identifiers of objects which belong to different types). If the object algebra is closed, then this heterogeneous set may be an input to another operation. Thus, it is important to be able to determine the set of methods that are applicable to all the objects in this heterogeneous set. To put it another way, the type that all these objects belong to have to be found. This is only possible by defining a type system which specifies rules for going up and down the inheritance lattice to determine the type conformance of objects.

A related issue is when to do type checking. Static type checking has the advantage of identifying errors early and without the potentially harmful results which could occur at run time. However, it also hampers dynamic binding of objects, which is a commonly advantage of object-oriented languages.

Finally, almost all applications have the requirement that a program variable be iteratively bound to

to determine the set of methods that are applicable (more precisely, a set of identifiers of objects which could occur at run time. However, it also hampers dynamic binding of objects, which is a commonly advantage of object-oriented languages.

There is no agreement on the set of operators or their semantics. As we indicated before, disagreements exist on whether object-creating operators should be included, what the proper level of encapsulation should be and so on. It has been suggested that object algebras should extend relational algebra [56], requiring the definition of project and Cartesian product operators. However, these operators, by definition, deal with components of objects, thereby violating strict encapsulation. As we indicated above, there is probably a need to include these operators in the language, but their exact relationship to encapsulation needs to be worked out.

As an example, and to aid the discussion in the next section, we will provide a formal definition of a closed object algebra consisting of four operations that are typically available in most object algebras. These operators are selection, difference, union, and intersection. Our definition of the operators assumes a data model that enforces strict object encapsulation and strong typing. We also assume that classes maintain the only collections of objects. However, operators generate sets of objects which may be inputs to other operators (since the algebra is closed). Therefore, operators are defined on sets of objects.

Some of the operators accept more than two operands. Let \( \theta \) be an operator in the algebra. We will use the notation \( P \theta (Q_1 \ldots Q_k) \) for algebra expressions where \( P \) and \( Q_i \) denote sets of objects. In the case where \( k = 1 \) we will use \( P \theta Q \) and where \( k = 0 \) we will use \( P \theta () \) without loss of generality.

Some of the algebra operators are qualified by a predicate. Such operators will be written \( P \Theta_F (Q_1 \ldots Q_k) \) where \( F \) is a formula consisting of one or more atoms connected by \( \land \), \( \lor \), or \( \lnot \) using parentheses as required. Atoms reference lower case, single letter variables which range over objects in the input set named with the corresponding upper case letter. For example, the object variables \( p, q_1 \) and \( q_2 \) in the predicate of \( P \Theta_F (p, q_1, q_2) \) \( (Q_1, Q_2) \) range over the sets of objects denoted by \( P, Q_1 \) and \( Q_2 \) respectively.

Select \( (P \sigma_F (Q_1 \ldots Q_k)) \): returns the objects denoted by \( p \) in each vector \( <p,q_1,\ldots,q_k> \in P \times Q_1 \times \ldots \times Q_k \) which satisfies the predicate \( F \). An equivalent expression for select is \( \{ p | P(p) \land Q_1(q_1) \land \ldots \land Q_k(q_k) \land F(p,q_1,\ldots,q_k) \} \).

Select permits multiple operands which simulates explicit joins as described in [28]. An explicit join is a join between arbitrary classes which support (a sequence of) method applications resulting in comparable objects. The result of the expression \( P \sigma_F (Q_1 \ldots Q_k) \), however, is a subset of \( P \), not sets of \( <P,Q_1,\ldots,Q_k> \) objects. In this sense, the select is most like the traditional semijoin operator.

Difference \( (P - Q) \): gives the set of objects which are in \( P \) and not in \( Q \). An equivalent expression for difference is \( \{ o | P(o) \land \lnot Q(o) \} \).

Union \( (P \cup Q) \): returns set of objects which are in \( P \) or \( Q \) or both. An equivalent expression for union is \( \{ o | P(o) \lor Q(o) \} \).

Intersection \( (P \cap Q) \): produces the set of objects which are in both \( P \) and \( Q \). An equivalent expression for intersection is \( \{ o | P(o) \land Q(o) \} \). The definition of the intersection operator is not strictly necessary since it can be derived by \( P - (P - Q) \).

The operators defined above are not that much different than those in relational algebra (except for the somewhat different semantics). We have chosen operators similar to their relational counterparts exactly because we can present them without much background discussion. Our aim is to use them as the basis of some of our discussions in the next section.

If the reader is left wondering about the special features of object algebras, let us simply indicate
that other algebra definitions include operators to take Cartesian product and project [40], checking the equality of complex object structures down to a certain level of composition [45], and operators that iterate over a class, applying a sequence of methods to each object in that class [45, 51]. As we stated earlier, our objective in this paper is not to perform an exhaustive survey or critique (see [56] for that), but to expose and highlight the various issues.

4 Query Optimization

What we have discussed so far are the better understood aspects of object-oriented query models and languages. Issues related to their optimization are much less understood. Optimization, as we use the term here, correspond to the last two steps of the methodology given in Figure 1, namely query rewriting to generate alternative query trees and the generation of alternative low-level execution plans for each of these trees. The logical separation of these two steps is a common one [20]. Query rewrite is a high level process where general purpose heuristics drive the application of transformation rules. Execution plan generation is a lower level process which identifies alternative execution plans based on low-level data manipulation operations. A further step that is not depicted in that figure represents low level optimization to select the "best" query tree and the "best" execution plan for that tree based on a specific cost model that incorporates the efficiency of storage structures and access paths.

Some implementations such as Iris [15] and POSTGRES [49] use relational databases as their storage subsystems. Such systems only use the semantics of the data model to translate queries into a form which reflects the implementation of the object model in the relational subsystem. They then rely on relational optimization techniques for overall query optimization. Banerjee et al. [6] provide an example of the object-oriented model to relational translation process for a database whose methods are restricted to returning internal object state information. In this restricted case, they show that a query in the object-oriented world has a semantically equivalent representation in the relational world. This implementation method has two drawbacks. First, it does not take advantage of data model semantics during query optimization. Second, relational databases typically support only simple data types such as numbers, booleans and strings. The object model permits the database implementor to provide basic types of arbitrary complexity, e.g., bitmaps. The inefficient representation of complex data types in relational systems make the relational model unsuitable for implementing the object-oriented paradigm.

Query rewrite rules. The overall goal of expression transformation is to reduce the cost of query evaluation by recasting the original algebraic expression as a more desirable one. This is accomplished by means of two sets of rules, algebraic and semantic, for the algebra operations defined in the language. Algebraic rules create equivalent expressions based upon pattern matching and textual substitution. Semantic rules are similar, but they are additionally dependent on the semantics of the database schema as defined by the type definitions and behavioral inheritance (subtype/supertype relationship).

Let us consider some example rules that one may develop for the object algebra operators that we have defined in the previous section. Our aim is not to be exhaustive, but to simply demonstrate what these rules look like and what is involved in defining them1. In the following, $P, Q_i$, and $R_i$ denote sets of objects and $(QSet)$ is a shorthand notation for $(Q_1, \ldots, Q_k)$. Similar definition holds for $(RSet)$.

\[
\begin{align*}
(P \sigma_{F_1} (Qset)) \sigma_{F_2} (Rset) &\Leftrightarrow (P \sigma_{F_3} (Rset)) \sigma_{F_4} (Qset) \\
(P - Q) \sigma_{F} (Rset) &\Leftrightarrow (P \sigma_{F} (Rset)) - Q \\
(P \cup Q) \sigma_{F} (Rset) &\Leftrightarrow (P \sigma_{F} (Rset)) \cup (Q \sigma_{F} (Rset)) \\
(P \cap Q) \sigma_{F} (Rset) &\Leftrightarrow (P \sigma_{F} (Rset)) \cap (Q \sigma_{F} (Rset)) \\
P \sigma_{F} (Q_1, \ldots, (Q_x \cup Q_y), \ldots, Q_k) &\Leftrightarrow (P \sigma_{F} (Q_1, \ldots, Q_x)) \cup (P \sigma_{F} (Q_1, \ldots, Q_y) \ldots, Q_k) \\
(P \sigma_{F_1} (Qset)) \sigma_{F_2} (Rset) &\Leftrightarrow (P \sigma_{F_1} (Qset)) \cap (P \sigma_{F_3} (Rset)) \\
P \sigma_{F_1 \cup F_3} (Qset, Rset) &\Leftrightarrow (P \sigma_{F_1} (Qset)) \cup (P \sigma_{F_3} (Rset)) \\
P \sigma_{F_1 \cup F_3} (Qset, Rset) &\Leftrightarrow (P \sigma_{F_1} (Qset)) \cup (P \sigma_{F_3} (Rset))
\end{align*}
\]

where:
\[
c : ref(F_1, (p, q_1 \ldots q_k)) \land res(F_1, p) \land \text{ref}(F_3, (p, r_1 \ldots r_l)) \land res(F_2, p) \\
\text{ref}(F, (v_1, \ldots, v_n)) \text{ is true when } v_1, \ldots, v_n \text{ are the only variables referenced in predicate } F
\]

\footnote{In defining rules, we use the notation $E_1 \Leftrightarrow E_2$ which specifies that expression $E_1$ is equivalent to expression $E_2$. We also use restricted rules of the form $E_1 \Leftrightarrow E_2$. Restricted rules are applicable only when the condition $c$ is true. Conditions are a conjunction of functions which determine properties of argument sets, predicates and variables used in a rule.}
res \((F, v)\) is true when predicate \(F\) restricts values of \(v\).

Rule (1) captures commutativity of select. Rules (2)–(4) show that difference, union and intersection commute with select. Rule (5) captures the commutativity of union in a trailing argument. Rule (6) is an identity which utilizes the fact that selection merely restricts its input and returns a subset of its first argument. The first selection, \(P \sigma_{F_1} (Q \sigma_{F_2} \text{sect})\), returns a subset of \(P\) (call it \(P'\)). The second selection can then be reduced to \(P' \sigma_{F_2} (R \text{sect})\) which is merely a smaller subset of \(P\). The same final subset of \(P\) can be obtained by applying predicates \(F_1\) and \(F_2\) separately and taking the intersection of the results. Rules (7) and (8) recognize that subformulas \(F_1\) and \(F_2\) each reference only a subset of the arguments. Operand sizes are minimized by breaking \(F_1\) and \(F_2\) into separate select operations and intersecting \((F_1 \land F_2)\) or taking the union \((F_1 \lor F_2)\) of the results.

Notice that the transformation rules that we have specified above are all algebraic. An advantage of object-oriented query languages is that one can also take advantage of the rich semantics of the data model in defining these transformations. The database schema captures many relationships which can be used to simplify object algebra expressions. For example, let \(T_1\) and \(T_2\) represent types, \(C_i\) represent the class corresponding to \(T_i\) (where class \(C_i\) is the only collection of objects in the extent of type \(T_i\)). We can show that the expression \(C_1 \land C_2 = \phi\) when \(T_1 \neq T_2\) by noting that the data model restricts each object to membership in a single class.

The reason for going through this discussion is not to explain the rules that we have listed here. There's very little that is special about these algebra operators and their transformation rules. The point, though, is that the development of a query processor requires the identification of all these algebraic and semantic rules. Intuitively, one might argue that if the object data model is richer, then more semantic information may be available that can assist in the identification of transformation rules. On the other hand, a richer data model may also be significantly more complex, restricting the cases in which these rules are applicable. The exact nature of this tradeoff is hard to determine at this stage awaiting more work on formal query models and the identification of their transformation rules.

Optimization of method executions. The transformations discussed above are specified only for the data model's query primitives. Ideally, query optimization should be possible for queries which utilize user defined methods. But this is highly dependent on the language used to define those methods. In the worst case, the only optimizations possible are those provided by the compiler of the method implementation language. Examples of such optimizations are inline subroutine expansion, removal of loop invariants and efficient pipeline and register usage.

One approach assumes that behavioral abstraction is maintained at the logical level, while a structural object-oriented system exists at the lowest implementation level [19]. Objects and classes involved in a query are requested to reveal structural information by the query processor. Revealed expressions which still contain encapsulated behavior are recursively requested to reveal their equivalent (sequence of) structural expressions. When the revealing process bottoms out, the structural manipulation primitives are optimized by an extended relational query optimizer.

Another approach would be to use a purely functional language for user defined methods. Expressions in such languages can be recursively decomposed to sequences of primitive data manipulation operations. These decomposed sequences can then be optimized using the techniques described earlier.

Clearly, optimization of user defined methods is closely tied to the ability to reason about expressions in the method implementation language and is a significant area for future research.

Generation of execution plans. Execution plan generation is the process of mapping algebraic expression of queries to sequences of data manipulation operators which are present in the physical system. In the case of the relational data model, there is a close correspondence between algebra operations and the low level primitives of the physical system [44]. The mapping between relations and files, and tuples and records may have contributed to this strong correspondence. However, there is no analogous, intuitive correspondence between object algebra operators and physical system primitives. Thus any discussion of access plan generation must first define the low level object manipulation primitives which will be the building blocks of access plans.

We call this low level object manipulation interface the Object Manager (OM) interface. Object managers have received attention lately in the context of distributed systems [7, 11, 38, 54], programming environments [13, 25, 55] and databases [9, 10, 14, 22, 29]. These object managers differ in terms of their support for data abstraction, concurrency and object distribution. In addition, they are typically oriented towards "one-at-a-time" object access which is an inefficient
paradigm for query processing. There is a need to define an OM interface which maintains many features of previous object managers but supports "set-at-a-time" processing.

Once such an interface is defined, then execution plan generation can be thought of as creating a mapping from object algebra expression trees to trees of object manager operations. This mapping is neither trivial nor computationally inexpensive. There are many possible mappings and the search space for selecting the "best" alternative can be quite large. The choice is also dependent upon the physical implementation characteristics of each object, which we discuss next. An area of research should the development of equivalence preserving rewrite rules for trees of object manager operations. Such rules are required to generate alternative execution plans for each algebra tree and would allow global optimization of the entire process.

Physical optimization. This section is largely taken from Chapter 15 of [42]. Object storage and access is the responsibility of the object manager. In addition to providing a suitable interface for the generation of execution plans (as we discussed above), the object manager performs two other functions: physical clustering of objects, and localization of objects. In general, the object manager is also responsible for transaction management, but that's beyond the scope of this paper. Object clustering is the grouping of objects in the same memory extent, according to common properties, for example, the same value of an attribute or subobjects of the same object. By minimizing the number of memory extents to examine, fast access to clustered objects can be provided. Object localization gives the location of an object based on its identifier or content (e.g., an attribute value). It exploits object clustering information, possibly augmented with some form of indexing. The object manager, based on the clustering and localization of objects, has to provide efficient algorithms for implementing the interface operations.

Relational databases can be used as object managers, but they are only efficient at managing simple objects. The problem is made significantly more difficult in object-oriented databases due to large atomic objects and complex objects. Large atomic objects are quite frequent in new database applications. For instance, a digitized image in an image database can require a few megabytes of storage. The object manager should be able to deliver only useful portions of a large atomic object to the application program or ADT operation that needs it. Complex objects may also be large because objects can be nested within each other using set and tuple constructors to an arbitrary degree. The typical example, from CAD applications, is a VLSI chip object that consists of several sections (e.g., 10), each consisting of many cells (e.g., 100), each containing more than 1000 transistors. Although the number of atomic objects of a VLSI chip (cells) is small (e.g., 100 bytes), the complex object may require several megabytes of storage. The object manager must be able to access an object and its subobjects rapidly if the entire complex object is needed. It must also provide efficient access to collections of subobjects without having to read the large complex object. The management of complex objects is also made difficult by object sharing, which permits each subobject to have more than one parent.

Storage techniques for relational databases may well be extended to support complex objects. The philosophy of this approach is to retain as much of the relational model and its underlying technology as possible. It applied initially to System R for CAD application support [32] and more recently to PostgreSQL [49], an extension of INGRES. With the relational model, complex objects are decomposed into tuples (the subobjects). By treating tuple identifiers (TID) as attribute values, the object manager can maintain the links between the subobjects composing an object. An atomic object can be stored as a tuple (TID, atomic value). The nesting of a tuple $t_i$ within a tuple $t_p$ is represented by storing the identifier of $t_i$ as an attribute of $t_p$. The nesting of a set of tuples $\{t_1, t_2, \ldots, t_n\}$ within a tuple $t_p$ can be represented by a binary relation containing the pairs (TID of $t_i$, TID of $t_p$), with $i = 1, \ldots, n$.

This storage approach brings out the benefits of the relational model. The access to subobjects stored in the same relation can be efficient if the clustering is appropriate. Furthermore, traditional indexing on attribute values is possible. Object query processing may be simplified with this approach. The conceptual query is first mapped into a relational query, which is expressed on the stored relations by replacing path expressions with the corresponding joins. The relational query can then be optimized using any relational query optimization technique. However, the disadvantage of this approach is that access to an entire complex object requires joins on tuple identifiers. Furthermore this approach is not sufficiently general since object identity is restricted to tuples and atomic objects. Therefore, sharing of set objects at the conceptual level is difficult to map at the physical level. Furthermore, whether or not all object-oriented query primitives may be mapped to relational ones efficiently
is a question.

The alternative approach is to develop special complex object storage techniques. These must provide the capability of storing a complex object with its subobjects in the same memory extent. The early hierarchical and network database systems partially provided this capability. In CODASYL, restrictions are that a complex object must fit in a page and that records can only be shared using their physical identifiers (called database pointers). These techniques have recently been generalized to support nested relations and object-based models [26]. Special attention has also been paid to the storage of atomic objects of arbitrary size. In EXODUS [9], an atomic object is a long byte sequence, which can be accessed in parts through a byte index. The storage of arbitrarily complex objects is more involved because of object sharing. The main difficulty of complex object storage with sharing is when the parent containing a shared object is deleted. In this case, the shared subobject must be relocated with another parent, which can be an expensive operation. A simpler solution would be to use the relational storage approach whenever objects are shared. Another difficulty with this approach is indexing in order to access entire objects or subobjects, since objects may be nested within other objects. A solution is to have path indexes [34] that associate attribute values with paths to the objects.

With complex object storage, complex objects may be mapped more directly at the physical level so that an object and its subobjects may be clustered in the same memory extent. In this case the conceptual query is mapped into a query expressed on the stored objects. The query processing algorithm could be similar to the exhaustive search approach by commuting all joins of stored objects, and for each one, selecting the best access method to the stored object. The only difference here is in the choice of the best access method in the complex object. Since the access to a complex object may involve path expressions and predicates on nested objects, the availability of path indexes is critical for efficiency.

5 Conclusions

In this paper we discussed the issues that need to be considered in the development of query models and in the implementation of query processors in object-oriented database systems. The framework for the presentation is a query processing methodology that is depicted in Figure 1. One point that is evident from the foregoing discussion is the necessity for significant more research in the development of query models and languages for object-oriented database systems. The ideas in this paper are only preliminary points that arise from our work in this area.

Since we relied heavily on the methodology of Figure 1, it is important to comment on its feasibility. The fundamental criticism has to be the linearity of processing. The methodology gives the impression that the steps can be followed one after the other to arrive at an execution plan which is "optimal." This certainly is not true. The transformation step will generate a number of different algebra trees and the plan generation step will produce more than one execution plan for each of these trees. It is important to note that a strategy has to be followed which cycles back and forth between the logical algebra optimization phase and the access plan generation phase. This would allow interleaving transformations which change the shape of the query with the introduction of access plan subtrees possibly resulting in more efficient plans.

References


In *Proc. 12th International Conference on Very Large Databases*, pages 101-110, August 1986.


[57] S. Zdonik. Why properties are objects or some refinements of “is-a”. In *ACM/IEEE Fall Joint Computer Conference*, pages 41-47, November 1986.