

OBJECT-ORIENTED QUERIES: EQUIVALENCE AND OPTIMIZATION*

Gail M. Shaw and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, R.I. 02912

We are interested in efficiently accessing data in an object-oriented database. We have developed a query algebra which fully supports object identity and abstract data types, and have identified a variety of algebraic query transformations. The equivalence of two queries is complicated by the presence of object identity. In this paper we define a hierarchy of notions of equivalence for queries, and present examples of equivalent query transformations for each level of the hierarchy.

1. INTRODUCTION

A major issue in the development of query algebras is the potential for optimization. We have developed a query algebra that synthesizes relational query concepts with object-oriented databases[1]. The algebra supports an object-oriented model with abstract data types, encapsulation, type inheritance, and object identity. Unlike other languages proposed for object-oriented databases [2, 3, 4, 5, 6, 7] our algebra fully supports these object-oriented concepts and still provides full associative access to the database, including a join capability that respects the discipline of data abstraction. The similarities between the structure of our algebra and the relational algebra lead us to believe that relational optimization results will prove useful in object-oriented query optimization.

Query algebras can support optimization through syntactic transformations (e.g., [2, 7]) and we have found our algebra to be conducive to such transformations. However, in order to identify transformations producing equivalent results it is necessary to define the requirements for equivalence. An object-oriented query creates new objects in which to store the results of the query. These objects have well-defined types and unique identities, but can be structurally complex. The structural aspects of the results lead to new requirements for defining equivalence of query transformations. We have identified three notions of equivalence for queries and use these definitions when producing query transformations.

*Support for this research is provided by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under contract N0014-88-K-0406, by Apple Computer, Inc., and by US West.

In the next section, we briefly describe our object-oriented data model and our algebra for querying over that model. In section 3 the notion of equivalence of object-oriented queries is defined. We discuss query optimization in section 4 and present a variety of transformations satisfying our different definitions of equivalence.

2. THE ENCORE QUERY ALGEBRA

Our query algebra is based on the ENCORE object-oriented data model [8, 9, 10]. The model includes abstract data types, type inheritance, typed collections of typed objects, and objects with identity. We query over collections of objects using the type of objects in the collection as a scheme for the collection. The collections are considered to be homogeneous, although the objects in the collection may have a type which is a subtype of the collection member type.

A type is an abstract data type, and consists of an interface and implementation. The interface of an abstract type includes a set of *Properties* for instances of the type and a set of *Operations* which can be applied to instances of the type. Instances of a type are objects. Properties reflect the state of an object while operations may perform arbitrary actions on objects. Evaluation of a property of an object returns an object (or atomic value) and we denote such an evaluation using dot-notation (e.g. $s.P$ where s is an instance of type T and P is a property for type T).

In addition to the user defined abstract data types, we assume a collection of atomic types (Int, String, Boolean, etc.), a global supertype Object, and parameterized types Set[T] and Tuple[$\langle (A_1, T_1), \dots, (A_n, T_n) \rangle$]. These types have fixed sets of properties and operations, and they allow the creation of new, strongly-typed objects.

Types are related (in a lattice) by subtyping. A subtype S of a type T inherits properties and operations from T . Subtyping is limited to the addition of behavior (not constraints) to ensure substitutability of instances of type S in contexts expecting instances of T . All types are subtypes of type Object, thus all instances of types are objects in the database.

Type Object defines a family of operations for equality we call *i-equality* (an extended version of deep-equality from Khoshafian and Copeland [11]). Atomic objects are identical if they have the same values (i.e. they are equal as defined by their type) and non-atomic objects are identical when they are the same object (i.e. an object can only be identical to itself). Objects are *0-equal* ($=_0$) when they are identical and, for $i > 0$, two objects X and Y are *i-equal* ($=_i$) when they both have the same type (call it T) and

- 1) if T is a collection type, then X and Y have the same cardinality and there is a one-to-one correspondence between the collections such that corresponding members are $=_{i-1}$ (i.e. there is a bijection $f: X \rightarrow Y$ such that for all x in X , $x =_{i-1} f(x)$)
- or 2) if T is not a collection type then, for all properties P defined by type T , $X.P =_{i-1} Y.P$

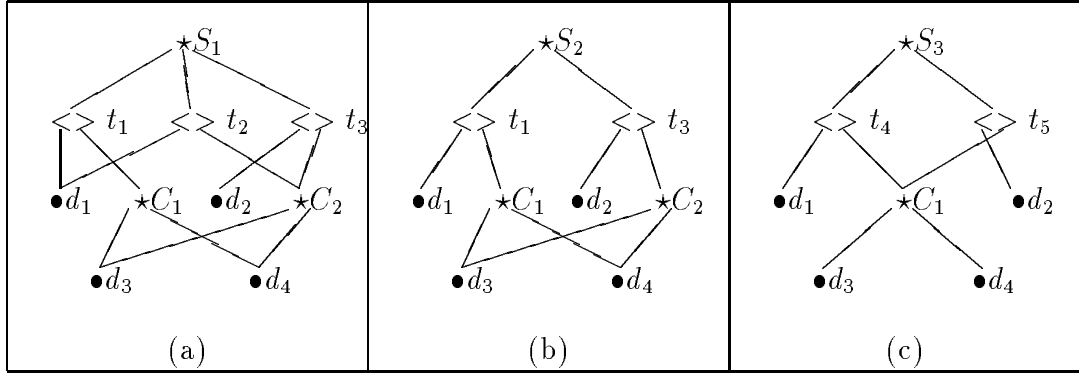


Figure 1: Structural representation of objects

Abstract data types can also define equality operations. Such operations are type-specific and will not be considered in this paper.

Objects can be represented graphically, where nodes are objects or atomic values, arcs connect collection type objects to all objects that are members of the collection, and arcs connect non-collection type objects to the values of all properties of that object (similarly to [11] for example). Nodes representing atomic values are leaves and are labelled with the atomic value. Nodes representing non-atomic objects are labelled with an (artificial) object identifier as well as the object type. Types can be labelled symbolically, using \star for set types, $\langle \rangle$ for tuple types, and \bullet for other abstract data types. In Figure 1 three set objects are represented graphically. Note that graphs b and c represent 3-equal set objects: C_2 and C_1 are 1-equal, making t_3 and t_5 2-equal (t_1 and t_4 are 1-equal, therefore 2-equal), thus S_2 and S_3 are 3-equal. Object S_1 (graph a), on the other hand, is not related by i-equality to either S_2 or S_3 since it is a set with different cardinality.

The query algebra provides type specific operations against collections of objects with identity. We assume collection objects have type $\text{Set}[T]$, where T is some data type. (We can use the algebra to query over other kinds of collections than sets by applying a coercion operator to produce a related set. For example, we would convert a list to a set before applying query operators.) Queries return new objects having type $\text{Set}[Q]$, where type Q is determined by the query. Duplication in set membership is determined using object identity; two members of a set cannot be identical.

A query result will often be a collection of existing database objects, in which case Q is an existing type. However, the properties of existing types may not explicitly reflect all relationships desired by a query. Thus we also provide algebraic operations that create new objects to store relationships between objects. These result objects are created using the parameterized type Tuple , so their data type is also well-defined.

Select:	$Set[T], p \longrightarrow Set[T]$
Image:	$Set[T], f : T \rightarrow Q \longrightarrow Set[Q]$
Project:	$Set[T], \langle (A_1, f_1 : T \rightarrow T_1), \dots, (A_n, f_n : T \rightarrow T_n) \rangle \longrightarrow$ $Set[Tuple[\langle (A_1, T_1), \dots, (A_n, T_n) \rangle]]$
Ojoin:	$Set[T], Set[R], A_1, A_2, p \longrightarrow Set[Tuple[\langle (A_1 : T), (A_2 : R) \rangle]]$
Union, Difference, Intersection:	$Set[T], Set[R] \longrightarrow Set[S]$ where S is the most specific common supertype of T and R
Flatten:	$Set[Set[T]] \longrightarrow Set[T]$
Nest:	$Set[Tuple[\langle (A_1, T_1), \dots, (A_i, T_i), \dots, (A_n, T_n) \rangle]], A_i \longrightarrow$ $Set[Tuple[\langle (A_1, T_1), \dots, (A_i, Set[T_i]), \dots, (A_n, T_n) \rangle]]$
UnNest:	$Set[Tuple[\langle (A_1, T_1), \dots, (A_i, Set[T_i]), \dots, (A_n, T_n) \rangle]], A_i \longrightarrow$ $Set[Tuple[\langle (A_1, T_1), \dots, (A_i, T_i), \dots, (A_n, T_n) \rangle]]$
DupEliminate:	$Set[T], e \longrightarrow Set[T]$
Coalesce:	$Set[Tuple[\langle (A_1, T_1), \dots, (A_i, T_i), \dots, (A_n, T_n) \rangle]], A_i, e \longrightarrow$ $Set[Tuple[\langle (A_1, T_1), \dots, (A_i, T_i), \dots, (A_n, T_n) \rangle]]$

Figure 2: Algebraic Operations

The algebraic operations can be divided into two categories:

- 1) operations that retrieve data: Select, Image, Project, Ojoin, Union, Intersection, and Difference
- 2) operations that support data retrieval through manipulation of result structure and object identity: Flatten, Nest, UnNest, DupEliminate, and Coalesce

The signatures of these operations are given in Figure 2. In the figure, p represents a predicate and e represents some equivalence relation (e.g. $=_i$). The operator set shown is not minimal. The redundancies in the operator set allow transformations which provide opportunities for query optimization.

The *Select* operation creates a collection of database objects satisfying a selection predicate, i.e. $Select(S, p) = \{s \mid (s \text{ in } S) \wedge p(s)\}$. The *Image* operation is used primarily to return a single value, or object, for each object in the queried class ($Image(S, f) = \{f(s) \mid s \text{ in } S\}$). In some ways, this is a selection of objects from a different collection than the one over which the query is defined. For example, suppose type *Stack* defines a read-only property called *top* which returns the object at the top of a stack instance. $Image(Stacks, \lambda s s.top)$ then returns a set of objects that are the tops of stacks (i.e. Select all objects which are at the top of some stack). The set will not contain two identical objects; if an object is the top of two stacks it will be represented once in the result set.

The Project and Ojoin operations can create new relationships not explicitly defined in the properties of the object types. These operations produce tuples to store the computed relationships. We define $Project(S, \langle (A_1, f_1), \dots, (A_n, f_n) \rangle) = \{\langle A_1 : f_1(s), \dots, A_n : f_n(s) \rangle \mid s \text{ in } S\}$. 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, $Project(Stacks, \lambda s \langle (S, s), (L, Select(Queues, \lambda q q.len = s.len)) \rangle)$ returns, for each stack, all queues having the same length. The result is a collection of 2-tuples with attribute S of type *Stack* and L of type $Set[Queue]$. Project

can also act similarly to a relational Project by extracting only properties from the objects in the queried collection. $Project(Stacks, \lambda s < (Top, s.top) >)$, for example, acts as the Image operation in the previous paragraph, except that it returns one tuple for each stack. As a result some tuples may be 1-equal (if their *Top* values are identical).

The *Ojoin* operator is an explicit join operator used to create relationships between objects from two classes in the database. The matching of stacks and queues could be accomplished using $Ojoin(Stacks, Queues, S, Q, \lambda s \lambda q s.len = q.len)$. This result is stored as a collection of 2-tuples with one Stack type attribute and one Queue type attribute (compare this to the Set[Queue] attribute of the Project result). For two input collections containing objects having abstract data types we define $Ojoin(S, R, A_s, A_r, p) = \{ \langle A_s : s, A_r : r \rangle \mid s \text{ in } S \wedge r \text{ in } R \wedge p(s, r) \}$. However, if one set is a collection of n-tuples, Ojoin creates (n+1)-tuples (n attributes from the set of tuples and one attribute to hold an object from the other set). This definition preserves the associativity of the operation. Although Ojoin is a special case of Project (see Figure 5) the redundancy could prove useful for optimization.

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 [12]) to sets of objects with identity. Sets of tuples can be unnested on a single set-valued attribute, or nested to create a set-valued attribute. More specifically, $Nest(S, A_i) = \{ \langle A_1 : s.A_1, \dots, A_i : t, \dots, A_n : s.A_n \rangle \mid \forall r \exists s (r \text{ in } t \wedge s \text{ in } S \wedge s.A_i = r) \}$ and $UnNest(S, A_i) = \{ \langle A_1 : s.A_1, \dots, A_i : t, \dots, A_n : s.A_n \rangle \mid s \text{ in } S \wedge t \text{ in } s.A_i \}$. The UnNest operation creates a new tuple for each object in the designated attribute. For example, a tuple $t = \langle A : o_1, B : s_1 \rangle$, where object s_1 is a set containing o_2 and o_3 , is unnested on the B attribute to create two new tuples: $t_1 = \langle A : o_1, B : o_2 \rangle$ and $t_2 = \langle A : o_1, B : o_3 \rangle$. On the other hand, a Nest on the B attribute of a set containing tuples t_1 and t_2 would create a tuple 1-equal to t .

Operation *DupEliminate* provides the option of eliminating duplicate copies of objects from a collection, where the type of equality used to determine duplication is a parameter of the operation. 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 i-equal. In figure 1, S_1 is a collection containing 2-equal objects (t_1 and t_2). Graph b of the figure represents the result of executing the operation $DupEliminate(S_1, =_2)$.

Operation *Coalesce* can be applied to a collection 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. $Coalesce(S, A_k, =_i)$ applied to a collection S of tuple objects, eliminates i-equal duplication in the A_k components of the tuples. For example, recall the query $Project(Stacks, \lambda s < (S, s), (L, Select(Queues, \lambda q q.len = s.len) >)$ relating each stack in a set to all queues having the same length as the stack. For each stack a new set is created to store the matching queues so, for two equal length stacks, two 1-equal (*not* identical) sets will be created. Coalescing the result would ensure that stacks having the same length are paired with identical sets of queues. This situation is illustrated in Figure 1, where structure *c* represents the result of a Coalesce operation

applied to structure b (i.e. apply $\text{Coalesce}(S_2, C, =_1)$). The set-valued attributes of tuples t_1 and t_3 in S_2 are 1-equal ($C_1 =_1 C_2$) thus are Coalesced to give identical set-valued attributes in t_4 and t_5 ($t_4.C =_0 t_5.C$).

The algebraic operators preserve the typing of the object-oriented data model and maintain the identities of objects in the database. The operations can create new objects in the database, but the types of those objects are statically defined. The creation of new objects leads to a variety of notions of result equivalence which we address in the next section. In addition, the creation and manipulation of these objects affect query optimization, since some result structures might be more efficient for querying than others.

3. QUERY EQUIVALENCE

We want to identify transformations of queries that produce equivalent queries. Such transformations, combined with a cost measure for the results, would provide a basis for object-oriented query optimization. However, in order to apply transformations, we need to first define the notion of equivalent results. The result of a query is a new collection of objects with a unique identity, thus two responses to even the same query cannot be identical. The creation of new objects by the algebraic operations means that we can consider the structure of results as well as the data retrieved by a query when defining equivalence.

The structure of a query result is reflected by its graphical representation. Let R be the result of some query Q (on some database D), and $G(R) = (V(R), E(R))$ be the graphical representation of object R . $V(R)$ is the set of nodes in the graph and $E(R)$ is the set of arcs connecting elements of $V(R)$. We consider two disjoint subsets of $V(R)$:

- 1) $V_Q(R)$ — the set of objects in $V(R)$ that were created by query Q
- 2) $V_{Qdata}(R)$ — the set $V(R) - V_Q(R)$

The set $V_Q(R)$ will only contain objects having type Set or Tuple, since these are the only types created by queries. Also, $V_Q(R)$ will always contain at least one element, since a query always returns a new set object. $V_{Qdata}(R)$ contains, intuitively, the set of data objects returned by query Q .

Our first notion of query equivalence considers two queries to be equivalent if they return the same objects from the database (i.e. their results have the same Qdata sets):

Definition 3.1. *Two queries, Q_1 and Q_2 , are weakly equivalent (\approx) if, when applied to an arbitrary database, they return results R_1 and R_2 , respectively, with corresponding graphical representations $G(R_1)$ and $G(R_2)$ such that $V_{Q_1data}(R_1) = V_{Q_2data}(R_2)$.*

Note that the Qdata sets are mathematical sets, not objects, thus equality of the Qdata sets is set equality.

Weak equivalence disregards the types of the objects containing the results, i.e it disregards the objects built by a query to hold results. Although both result objects are sets, they could be sets of different types or different cardinalities. For example, one could be a set of tuples containing objects of type T (e.g., $\text{Set}[\text{Tuple}[A:T,B:T]]$) and the other a set of objects having type T (i.e., $\text{Set}[T]$), where the objects of type T are retrieved from the database. These two results cannot be compared by our standard notions of object equality since they have different types.

Weak equivalence ignores the structure built by a query to store the results. Thus an application of any of the formatting types of instructions (UnNest , Nest , DupEliminate , etc.) will not affect weak equivalence.

We use i -equality to define a stronger notion of equivalence requiring results to have the same type:

Definition 3.2. *Two queries are equivalent at depth i (or i -equivalent, represented \cong_i) if, when applied to an arbitrary database, they return i -equal objects.*

The equivalence depth will depend on the query. Nested queries that build tuples will increase the depth of the structure built by the query. The objects created by the query operation will have unique identities, and this definition implies an identity-oblivious search through the structures to find the data retrieved from the database. For example, any two non-set objects are compared property-wise, thus two distinct objects in one set that have identical values for a given property could match with two distinct objects in a different set with i -equal values for the same property. The fact that the two objects in the first set have aliases for the property value is lost with i -equivalence.

The transparency of identifiers in i -equivalence means a loss of information about the structure of the results. Two results that are i -equivalent may have non-isomorphic graph representations (see Figure 3, b and d, for example). By ignoring object identities, information about aliasing of objects may be lost. Thus we define an even stronger notion of equivalence:

Definition 3.3. *Two queries are structurally equivalent at depth i (also called i -equivalent, and represented \equiv_i) if, when applied to an arbitrary database, they return objects that are i -equal and have isomorphic graph representations.*

The three definitions of equivalence are exemplified in Figure 3. In that example, all d_i 's are database objects (they have arbitrary data types) and S_i 's, t_i 's and C_i 's are objects created by the execution of query operations. S_1, S_2, S_3 and S_4 are result collections from four queries. All four results are weakly equivalent; they all return d_1, d_2, d_3 , and d_4 from the database. However, S_1 is not i -equal to any of the other results since it is a set of 2-tuples where both tuple attributes (properties) are abstract data types. All of the other sets contain 2-tuples where one attribute is known to have type Set . Sets S_2 and S_3 are structurally equivalent at depth 3; the assignments $S_2 \leftrightarrow S_3, t_4 \leftrightarrow t_6, t_5 \leftrightarrow t_7, C_1 \leftrightarrow C_3$, and $C_2 \leftrightarrow C_4$ illustrate the isomorphism with the

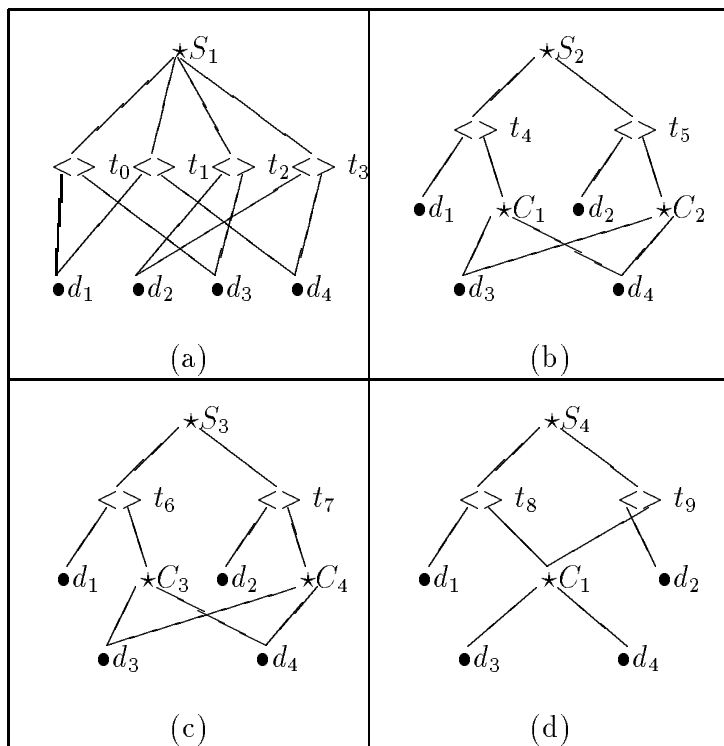


Figure 3: Equivalent query results. All results are weakly equivalent; S_2 and S_3 are id-equivalent to each other and i-equivalent to S_4 .

corresponding C 's being 1-equal. S_4 is i-equivalent at depth 3 to S_2 and S_3 , but not id-equivalent. In particular, in S_4 both t_8 and t_9 have identical values for their set-valued attribute (i.e. C_1) but in S_2 (for example) the corresponding set-valued attributes in t_4 and t_5 are 1-equal. In S_4 , changes in $t_8.C$ can affect t_9 . That is, the set-valued attribute is aliased.

The definitions of equivalence illustrate the strengths, and difficulties, of querying when objects have identity. A query not only returns data, but defines some relationships between objects in the database. The operators allow the specification of those relationships and the description of how the relationships will be stored in objects. The different definitions of equivalence allow differences in the relationships stored (weak equivalence vs. i-equivalence) and in the internal structure of the objects storing those relationships (i-equivalence vs. structural-equivalence).

4. QUERY TRANSFORMATIONS AND OPTIMIZATION

The similarities between our algebra and the relational algebra, and the flexibility of the operator set in handling data types and identities, allow us to apply a variety of transformations to the algebra to produce results that are equivalent by the definitions presented in the last section. In Figures 4 and 5 we identify some transformations

$$\begin{aligned}
Ojoin(As, Bs, A, B, \lambda a \lambda b p(a, b)) &\approx Select(Project(As, \lambda a < (A, a), \\
&\quad (B, Select(Bs, \lambda b p(a, b))) >), \\
&\quad \lambda t NOT Empty(t.B)) \tag{1} \\
Flatten(S) &\approx UnNest(Project(S, \lambda a < (A, a) >), A) \tag{2} \\
Image(As, \lambda a Select(Bs, \lambda b b.prop = a.prop)) &\approx Select(Bs, \lambda b \exists a (a in As \wedge b.prop = a.prop)) \tag{3} \\
Image(As, \lambda a f(a)) &\approx Project(As, \lambda a < (F, f(a)) >) \tag{4} \\
UnNest(NSTupleQuery, A) &\approx NSTupleQuery \tag{5} \\
AnyQuery &\approx DupEliminate(AnyQuery, =_i) \tag{6} \\
AnyTupleQuery &\cong_{i+2} Coalesce(AnyTupleQuery, A, =_i) \tag{7}
\end{aligned}$$

Figure 4: Some query transformations

satisfying the different definitions of equivalence. An optimizer can use the different equivalences to produce transformations that are structurally equivalent and to produce transformations that are not necessarily structurally equivalent but may provide more efficient query or program processing. We discuss these possibilities in this section, after first examining some transformations in each equivalence class.

4.1. The Nature of Query Transformations

Weakly equivalent queries often return data in different formats. For example, in identity 1 of Figure 4 we see that a Project operation can return the same data as an Ojoin, although the Project operation returns a nested tuple whereas Ojoin returns “flat” tuples. The type of a result can also be manipulated by the inclusion or exclusion of the formatting types of instructions in query expressions. The operations Flatten, Nest and UnNest modify the type, but not the data, returned by a query. Identity 5, for example, indicates that a query operation producing tuples with a set type attribute, followed by an UnNest, will produce a result that is weakly equivalent to the same query without the UnNest. The query results with and without the UnNest have different types.

I-equivalence maintains the type of queries, and weakly equivalent queries may have the same type. In particular, DupEliminate and Coalesce manipulate the identities of objects but not their types. Any query is weakly equivalent to the same query followed by an application of DupEliminate (identity 6). The result sets will contain the same types of objects, but are not i-equivalent because they may have different cardinalities. Similarly, Coalesce eliminates duplication in an object’s structure and thus, when applied after any other algebraic operation, produces a result that is not id-equivalent but i-equivalent to the operation without the Coalesce (identity 7).

Weak equivalence does not guarantee that relationships between data objects will be maintained. For example, in identity 3 assume that As are Stacks, Bs are Queues, and *prop* is the length property for stacks and queues. The Select operation creates a collection containing all queues for which there is some stack having the same length. The Image operation returns the same queues, but they are collected into sets of queues

$$\begin{aligned}
Ojoin(As, Bs, A, B, \lambda a \lambda b p(a, b)) &\equiv_2 Select(UnNest(Project(As, \lambda a < (A, a), (B, Bs) >), B), \\
&\quad \lambda t p(t.A, t.B)) \tag{8} \\
Ojoin(As, Bs, A, B, \lambda a \lambda b p(a, b)) &\equiv_2 UnNest(Select(Project(As, \lambda a < (A, a), \\
&\quad (B, Select(Bs, \lambda b p(a, b))) >), \\
&\quad \lambda t NOT Empty(t.B)), B) \tag{9} \\
Flatten(S) &\equiv_1 Image(UnNest(Project(S, \lambda s < (A, s) >), A), \\
&\quad \lambda a a.A) \tag{10} \\
Coalesce(S, A_k, =_i) &\equiv_{i+2} Project(Ojoin(S, NoDupA_k, \lambda a \lambda b a.A_k =_i b), \\
&\quad \lambda t < (A_1, t.A_1), \dots, (A_{k-1}, t_{k-1}), \\
&\quad (A_k, t.B), \\
&\quad (A_{k+1}, t.A_{k+1}), \dots, (A_n, t.A_n) >) \tag{11} \\
\text{where } NoDupA_k &:= DupEliminate(Image(S, \lambda s s.A_k), =_i) \\
Select(As, \lambda a \exists b (b \text{ in } Bs \wedge p(a, b))) &\equiv_1 Image(Ojoin(As, Bs, A, B, \lambda a \lambda b p(a, b)), \lambda t t.A) \tag{12} \\
Select(Ojoin(As, Bs, A, B, p), \lambda t p_s(t.A)) &\equiv_2 Ojoin(Select(As, p_s(a)), Bs, A, B, p) \tag{13} \\
Select(Ojoin(As, Bs, A, B, p(a, b)), \lambda t p_s(t.A, t.B)) &\equiv_2 Ojoin(As, Bs, A, B, p(a, b) \wedge p_s(a, b)) \tag{14} \\
Ojoin(As, Bs, A, B, \lambda a \lambda b p(a) \wedge p'(a, b)) &\equiv_2 Ojoin(Select(As, p(a)), Bs, A, B, \lambda a \lambda b p'(a, b)) \tag{15}
\end{aligned}$$

Figure 5: Some structurally-equivalent query transformations.

having the same length (for each stack, Image creates one set of queues having the same length as the stack). Each set in the Image result represents a relationship (of length) between queue objects that is not explicit in the Selection result.

Operations that create new objects can permit duplication in the data returned. For example, consider identity 4 relating the Image and Project operations. The queries are only weakly equivalent since they return different types (Set[Tuple[<F:ftype>]] vs. Set[ftype]). In addition, the Projection result may have a different cardinality than the Image result since the former may contain tuples that are 1-equal (function f could return identical objects for different input objects). Operation DupEliminate could be applied after the Project to eliminate any duplication giving, again, a weakly equivalent result. The Project and DupEliminate-Project results have the same type, but are not i-equivalent since they contain different numbers of objects.

Some id-equivalent transformations are displayed in Figure 5. Identities 8 and 9 in Figure 5 are id-equivalent versions of identity 1 (Figure 4). A Project, with Selection and UnNesting for formatting, is structurally equivalent to Ojoin. Similarly, identity 10 is the id-equivalent version of identity 2. The Image operation in the former “converts” the single-attribute tuples into single objects having the attribute type.

Id-equivalent transformations are useful in identifying redundant operators within our algebra. Identities 8 - 11 illustrate the redundancy of operators Ojoin, Flatten and Coalesce, respectively. Identity 12 illustrates that some first order predicates can be replaced by algebraic operations using propositional predicates. The redundancy in the operator set may offer opportunities for optimization. We would expect, for example, that a Flatten operation would be more efficient than the simulation of that operation with Project, UnNest and Image, since the Flatten operation more succinctly describes the required result.

Relational transformation results can also be applied to the object-oriented query algebra, resulting in id-equivalent transformations. For example, one relational optimization strategy is to push Selection past Join. That same strategy is available in our algebra (see identity 13 Figure 5). Similarly, when a Select operation is composed with an Ojoin, it may be possible to instead compose the two predicates to produce a single operation (Figure 5 identity 14). These two ideas are combined in identity 15 of Figure 5. If an Ojoin predicate contains a conjunct involving only one of the operand collections, that conjunct can be extracted from the Ojoin predicate to form a Select predicate on the appropriate operand.

4.2. Using Weaker Equivalences in Query Transformation

A rule-based query optimizer (e.g. [13]) can use transformations like the ones illustrated here to assist in determining efficient methods for processing a query. We expect id-equivalent transformations to be particularly useful in optimization, but transformations that are less than structurally equivalent can also be useful when an optimizer tries to generate new transformations.

For example, consider the query

$$Select(Stacks, \lambda s \exists q (q \text{ in } Queues \wedge q.len =_0 .len)) \quad (1)$$

selecting all stacks that are the same length as some queue. Transformation 12 can be applied to the query to give the following structurally-equivalent query.

$$Image(Ojoin(Stacks, Queues, A, B, \lambda s \lambda q q.len =_0 s.len), \lambda t t.A) \quad (2)$$

Transformation 9 could then be applied to produce

$$Image(UnNest(Select(Project(Stacks, \lambda s < (A, s), (B, Select(Queues, \lambda q q.len =_0 s.len)) >), \lambda t NOT Empty(t.B)), B), \lambda t t.A) \quad (3)$$

At this point, the optimizer has no further id-equivalent transformations to apply and may backtrack to query 2. Transformation 8 could be applied to query 2, but will also lead to a query which cannot be further transformed using id-equivalence.

A comparison of queries 2 and 3 by the optimizer might find that query 3 is efficient to process when there is an index on *len* of *Queues*. However, if we examine query 3 carefully we find that it might be possible to achieve even more efficient execution of the query by eliminating the superfluous *UnNest* operation. In query 3 the *UnNest* operation modifies the type of the *B* attribute of the tuple (from *Set[Queue]* to *Queue*), but the *B* attribute is not needed in the final result. In order to eliminate the unnecessary operation we want to apply identity 5 of Figure 4 to get

$$Image(Select(Project(Stacks, \lambda s < (A, s), (B, Select(Queues, \lambda q q.len =_0 s.len)) >), \lambda t NOT Empty(t.B)), \lambda t t.A) \quad (4)$$

Although the transformation uses weak equivalence, the final result is id-equivalent. This occurs because the Image operation extracts the A attribute values for the final result, thus does not care about the formatting of the rest of the tuple. The fact that this particular operation does not require structural equivalence of the nested query could be inferred by the optimizer through examination of the Image function (i.e., noting $\lambda t.t.A$ does not reference the B attribute) or could be explicitly specified by a user.

We could have also obtained query 4 by applying the weakly equivalent transformation of identity 1 to query 2. Clearly, we could have also defined an id-equivalent transformation that would transform query 1 or 2 directly into query 4 instead of using weak equivalence to achieve the transformation. However, the use of weak equivalence allowed us to make the transformation without resorting to defining a more specific id-equivalence. In other words, the use of weak equivalence gave the power of a larger set of id-equivalent transformations.

4.3. Using Weaker Equivalences in Query Optimization

Id-equivalent results are required to maintain all structural associations between data objects defined by a query. We might expect to require an optimizer to maintain this level of equivalence when transforming queries, since id-equivalence ensures that a transformed query returns a result that is not only the type of the original query but is isomorphic to the result described by the query. Weakly equivalent queries might have different types than those described by a query being optimized, and i-equivalent queries might produce results that have unwanted aliases for objects.

However, a user may not require that a query transformation maintain structural equivalence. For example, in section 4.2 the Image operation did not require structural equivalence for transformations involving its nested query. Some users may view their expression of a query as a description of the data desired, and not care about the types returned by the query to store the data. Other users might require that a transformation maintain the type of a result, but not necessarily the graphical structure. By specifying the minimum requirements for equivalence the user gives the optimizer more freedom in choosing applicable transformations. This might allow the optimizer to use information about the database that may be hidden from the user.

For example, consider the query which matches stacks with queues by length:

$$Project(Stacks, \lambda s < (S, s), (L, Select(Queues, \lambda q q.len = s.len) >) \quad (5)$$

Suppose there is an index on the length of queues and that the index is maintained as a collection of set objects, i.e. an index entry for some length l references a set object containing all queues of length l . An efficient implementation of the operation $Select(Queues, \lambda q q.len = c)$ (call it Index-Select) could return the set object associated by the index with length c . This object could be extracted directly from the index. However, using this implementation of Select when executing query 5 would give a result that is i-equivalent to the the result described by the query. That is, using

Index-Select is the same as applying identity 7 to query 5. If a user requires structural equivalence this transformation could not be applied. However, if the user only requires that the result type be maintained, the optimizer could use information about the index and apply i-equivalence to produce a more efficient query.

Using weaker equivalences might also allow an optimizer to improve the efficiency of operations following a query. For example, consider again query 5 and suppose a subsequent operation changes the length of a queue q . If the query result was not Coalesced on the L attribute, all tuples whose L attribute objects contain q and all tuples with the same length as the new length of q would have to be updated. However, if the query result had been Coalesced on 1-equality of the L attribute, the update would involve only the removal of q from one set and its addition to the appropriate set. Once again, allowing the optimizer the freedom to choose i-equivalence results in more efficient results.

The existence of the different notions of equivalence points out new decisions to be made in the query optimization process that are not necessarily required when objects do not have identity. In general we expect that, unless specified otherwise, a user requires a query to maintain its result type, but not necessarily its graphical representation. As a result, an optimizer can apply any structural or i-equivalent transformations and any weakly equivalent transformations that maintain result type. These transformations give the optimizer the freedom to use information about the database system in choosing transformations to provide efficient query processing. An operation invocation, such as the Image operation, could override the equivalences by requesting even weaker equivalence. On the other hand, a user could also require id-equivalence of query results. Indeed, we assume that the inclusion of Coalesce or DupEliminate instructions by a user indicates a desire for manipulation of object identities that should not be overridden by the optimizer. In this case id-equivalence is the default requirement and a weaker equivalence would have to be explicitly requested.

5. SUMMARY

The Encore query algebra and model provide a strong base for the exploration of query optimization in an object-oriented system. The algebra strongly supports abstract data types and encapsulation, and manipulates objects based on object identity. The algebra can create new objects, with new identities, to store data and associations defined by a query.

The presence of objects with identity leads to three definitions for equivalence of query transformations. Weak equivalence is based only on database objects returned by a query, i-equivalence recognizes the preservation of relationships between database objects, and structural equivalence recognizes differences in identities in the structures of objects storing query results. These distinctions need to be recognized by an object-oriented query optimizer in order to effectively use transformations when optimizing queries involving objects with identity.

We are currently exploring the use of transformations such as those presented here in optimizing queries on an object-oriented database. Although one might expect to require structural equivalence of transformations, an optimizer can exploit the weaker definitions when structural equivalence is not required. We must also develop the means to evaluate such query transformations. This involves the definition of a cost measure as well as a way to apply such a measure in the presence of encapsulated objects.

REFERENCES

- [1] Shaw, G. M. and Zdonik, S. B., "A Query Algebra for Object-Oriented Databases," to appear in *Proc. 6th International Conference on Data Engineering*, IEEE, 1990.
- [2] Bancilhon, F. *et al.*, "FAD, a Powerful and Simple Database Language," in *Proceedings of the 13th VLDB Conference*, pp. 97–105, 1987.
- [3] Maier, D. and Stein, J., "Development and Implementation of an Object-Oriented DBMS," in *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, eds.), pp. 355–392, Cambridge, MA: MIT Press, 1987.
- [4] Banerjee, J., Kim, W., and Kim, K.-C., "Queries in Object-Oriented Databases," in *Proceedings 4th Intl. Conf. on Data Engineering*, pp. 31–38, IEEE, February 1988.
- [5] Carey, M. J., DeWitt, D. J., and Vandenberg, S. L., "A Data Model and Query Language for EXODUS," in *SIGMOD Proceedings*, pp. 413–423, ACM, June 1988.
- [6] Kim, W., "A Model of Queries for Object-Oriented Databases," Tech. Rep. ACA-ST-365-88, MCC, 1988.
- [7] Osborn, S. L., "Identity, Equality and Query Optimization," in *Advances in Object-Oriented Database Systems*, pp. 346–351, 2nd International Workshop on Object-Oriented Database Systems, September 1988.
- [8] Zdonik, S. B. and Wegner, P., "Language and Methodology for Object-Oriented Database Environments," in *Proceedings of the Hawaii International Conference on System Sciences*, January 1986.
- [9] Shaw, G. M. and Zdonik, S. B., "An Object-Oriented Query Algebra," in *Proceedings of the 2nd International Workshop on Database Programming Languages*, June 1989. Reprinted in *IEEE Database Engineering*, September 1989.
- [10] Elmore, P., Shaw, G. M. and Zdonik, S. B., "The ENCORE Object-Oriented Data Model," tech. rep. in preparation, Brown University, November 1989.
- [11] Khoshafian, S. N. and Copeland, G. P., "Object Identity," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 406–416, ACM, September 1986.

- [12] Jaeschke, G. and Schek, H. J., "Remarks on the Algebra of Non First Normal Form Relations," in *Proceedings of the Symposium on Principles of Database Systems*, pp. 124–138, ACM, March 1982.
- [13] Graefe, G., *Rule-Based Query Optimization in Extensible Database Systems*. PhD thesis, Univ. of Wisconsin-Madison, November 1987.