A Query Algebra for Object-Oriented Databases

Gail M. Shaw and Stanley B. Zdonik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-89-19
March 1989

A Query Algebra for Object-Oriented Databases

Gail M. Shaw
Stanley B. Zdonik

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

Abstract

We define an algebra which synthesizes relational query concepts with object-oriented databases. The algebra fully supports abstract data types and object identity while providing associative access to objects, including a unique join capability. The operations take an abstract view of objects and access typed collections of objects through the public interface defined for the type. The algebra supports access to relationships implied by the structure of the objects, as well as the definition and creation of new relationships between objects. The structure of the algebra and the abstract access to objects offer opportunities for query optimization.

1 Introduction

Object-oriented databases are emerging in response to a requirement for more complete modelling of complex data in modern design and office environments. Such databases attempt to combine the modelling power of object-based programming models with traditional database concepts such as persistence, data sharing, consistency, resiliency and optimized query access. A number of object-oriented data models and languages have been proposed ([Shi81], [Ban87a], [Ban87b], [Row87], [Mai87], [Lec88], [Car88]), but no single model or language has emerged as a standard for object-oriented databases. Our model supports object identity, abstract data types with type inheritance, and encapsulation of behavior and state, as modelling capabilities present in most object-oriented systems.

The major contribution of this paper is the development of an algebra which provides a model for querying in object-oriented databases by supporting the concepts of object identity, abstract data types, type inheritance and typed collections distinct from the type hierarchy. Unlike other languages proposed for object-oriented databases (e.g. [Zan83], [Car88], [Osb88], [Ban87a], [Ban88], [Mai87]) we support all of these concepts and still provide full associative access to the database, including a unique join capability.

We discuss our work in relation to previous models and languages in the next section; then, in Sections 3 and 4, present our object-oriented data model and our algebra for querying collections of objects in that model. We discuss problems associated with supporting object identity in Section 5, and illustrate the use of the algebra through a series of examples in Section 6. In Section 7 we introduce a new notion of equivalence for queries and present some query transformations that maintain this equivalence. The algebra presented forms a base for our current research into optimization of object-oriented queries, and we introduce that research in Section 8.

2 Relation to Previous Work

We are interested in accessing data in an object-oriented database. Relational database systems offer efficient access to large amounts of data but place the first normal form restriction on the structure of the data. This restriction supports the development of query languages and optimization strategies for the model.

One research direction in the modelling of object-oriented databases is the extension of the relational model to build complex objects. Early extensions relaxed the first normal form requirement of the relational model, allowing set-valued attributes ([Jae82], [Dad86]). The relational model has also been extended with abstract data types ([Row87], [Mai87], [Ban87a], [Alb85]) allowing the modelling of more complex types, where components or attributes of objects can be other database objects. We extend the relational data model similarly, and also succeed in applying relational types of algebraic operations to the model. Our algebra provides selection operations, standard set operations, modified operations for nested sets (from [Jae82]), operations for manipulating object identities, and unique projection and join capabilities. The algebra supports an object-oriented model with complex types, type inheritance, abstract data types and object identity (as in [Kho86]). We also assume that the collections of stored objects are supported by, but separate from, the type structure. This model is most similar to the models of GemStone [Mai87] and EXODUS [Car88], although our algebra is more general than the former and our approach to the operations and object identity differs from that of the latter.

We consider the type of every object to be an abstract data type. The implementation of objects, and their attributes and behavior, are invisible to the query algebra, and all access to objects is through the interface defined for the type. Thus the model could incorporate procedural [Row87] and functional data [Shi81] in addition to stored data values. Our access to data values is similar to that of Postgres; we use dot-notation to refer to attributes of objects and invocation of a function or procedure implementing an attribute is implicit.

The types are structured as an inheritance hierarchy. The only restriction the algebra places on type inheritance is substitutability; an object of type A can be used in any context expecting a supertype of A. We distinguish between type and collections of objects having that type, similarly to GemStone and EXODUS, and query over the collections using the type structure as the database scheme. The collections are treated as though they are homogeneous; the type of a collection can be a supertype of the objects in the collection. The result of a query is a new, typed database collection, with the collection type determined statically using the type structure.

Our support for object identity leads to a model and algebra in which objects have independent existence, comparisons between objects can distinguish between identifier equality and data equality, queries return collections of objects, and duplication in set membership is distinguished using objects' identities. Most systems make the distinction between data equality and identifier equality by providing operators testing identity, deep equality and, possibly, shallow equality (e.g. [Ban87a], [Lec88], [Osb88], [Car88]). Two objects are identical when they are the same object, shallow equal when they have the same value, and deep equal when they have the same type and a recursive traversal of the objects eventually finds identical values (see [Kho86]). Our algebra supports the semantics of object identity by assuming that the equality operator refers to object identity and using an extended notion of deep equality (see Section 3).

Identity is also an issue when returning objects satisfying query predicates. We use object identity as a test for set membership, as do most other systems, but we also allow, to a limited extent, the introduction of deep-equality semantics in determining set membership. The need for deep-equality results from the ability to create new, strongly-typed objects in response to queries. Osborn notes [Osb88] that the creation of new objects means that responses to a query cannot be identical. We propose new definitions for query equivalence that consider data as well as identifier equality (see Section 7).

Many object-oriented database systems (e.g. [Mai87], [Alb85], [Ban88]) only support selection of objects already existing in the database. We recognize that many relationships requested by queries will exist in the database objects, but also feel that existing objects in the database may not necessarily reflect all relationships requested by a query. Thus, we additionally provide operators to create new relationships, possibly by joining objects from different collections. The join capability is unique in that we join collections of objects by creating tuples to store relationships between objects in the collections. These tuples are complex objects with well-defined types and unique identities. The creation of these tuples as statically-typed objects is supported by the existence of parameterized types in the data model. Our algebra can simulate the join of the Excess query language [Car88] and the unnest join of Korth [Kor88], but not the Combine operation of Osborn [Osb88] since it does not necessarily maintain the integrity of abstract data types.

A major aspect in the development of query languages is the potential for optimization. Query algebras can support optimization through syntactic transformations (e.g., [Osb88], [Ban87a]). However, the implementation of abstract data types involves the introduction of new operations, making optimization more complicated [Zdo89]. Syntactic transformations are not sufficient for such systems, and may need to be combined with optimization strategies for encapsulated behaviors (see [Gra88] for example). We expect the structure of our algebra and our consistent approach to abstract data types to offer opportunities for combining such optimization strategies.

3 Data model

The data model for our query algebra is based on the ENCORE object-oriented data model [Zdo86]. The model includes abstract data types, type inheritance, typed collections of typed objects, and complex objects with identity. We query over collections using the type of objects in the collection as a scheme for the collection.

A type is an abstract data type, and consists of an interface and implementation. Queries are concerned with the type interface, although their optimization may be concerned with the type implementation. An abstract type definition includes the *Name* of the type, a set of *Supertypes* for the type, a set of *Properties* defined for instances of the type and a set of *Operations* which can be applied to instances of the type.

Properties reflect the state of an object while operations may perform arbitrary actions. Properties are typed objects that may be implemented as stored values, procedures or functions. We assume the implementation of a property returns an object of the correct type and has no side-effects. The query algebra treats properties as stored values, addressing a property using dot-notation (e.g. $s.q$ where s is an object of type T and q is a property of T). If necessary, a request for a property will cause invocation of a function implementing the property, resulting in the return of an object representing the requested property of the object.

When type T is a subtype of type S, we say S is a supertype of T. Subtyping is limited to the addition of behavior (not constraints) to ensure that an instance of T can be used in any context expecting S (substitutability).

We assume a collection of atomic types (Int, String, Boolean, etc.), a global supertype Object, and parameterized types Set[T], Multiset[T], and Tuple[$< (A_1, T_1), ..., (A_n, T_n) >$].[1] These types have fixed sets of properties and operations, and they allow the creation of new, strongly-typed objects. The Tuple type defines types (T_i) for attributes (A_i) and has properties *Get_attribute_value* and operations *Set_attribute_value* for each attribute. Dot-notation may be used to abbreviate the Get_attribute_value properties ($o.A_i$ for object o having type Tuple with attribute A_i). The value of a tuple is represented as $< A_1 : v_1, ..., A_n : v_n >$ where the A's are attributes of the tuple and the v's are objects of the corresponding type.

Parameterized type Set[T] defines T as the type, or supertype, of objects in a collection having type Set, and defines operations (among others):

$$\begin{array}{lll} in: & Set[T], T \rightarrow Boolean & (or Member\_of) \\ subset\_of: & Set[T], Set[T] \rightarrow Boolean & \end{array}$$

[1] Other parameterized types (e.g. Ordered_List[T]) might be useful for forming collections of objects. We have not yet addressed the semantics of our algebra on such types of collections, and for now assume that the algebra operates on collections that are sets or multisets. If the algebraic operations need to be applied to other collection types, the collection would first have to be coerced into one of the types we consider.

Duplication in set membership depends on object identity; a set will not contain two objects with the same identifier (i.e., the same object will not appear in the set twice). Type Multiset[T] behaves similarly, with multiple membership of the same object allowed. These types are used by the query language to form result collections.

Type Object has properties *identifier*, *type*, and *value*, and operation *identical* (as in [Kho86]). We also define *i-equality*, an extended version of deep-equality, where i indicates how "deep" a comparison must go before finding equality, and corresponding operation $=_i$. Identical objects are *0-equal* ($=_0$) and, for $i > 0$, two objects are *i-equal* ($=_i$) if

1) they are both collections of the same cardinality and there is a one-to-one correspondence between the collections such that corresponding members are $=_{i-1}$,

or 2) they both have the same type (not a collection type) and the values of corresponding properties are $=_{i-1}$.

Thus, two tuple objects are i-equal when their corresponding attribute values (i.e. properties) are (i-1)-equal. Two objects are deep-equal ($=_d$) if there is some i for which they are i-equal, and we define shallow-equal to refer to 1-equality, or $=_1$. Note also that for all j greater than or equal to i, $=_i$ implies $=_j$.

Types induce a subtype of collection objects called *classes.* A type T has an associated class Ts which contains all instances of type T. These instances are objects having type T or type S, where S is a subtype of T. The type of Class Ts is Class[T], where Class is usually Set. If Q is a subtype of T, then class Qs will be a subclass (i.e. subset) of Ts. Constraint-based (restriction) subclasses are also allowed. A constraint-based subclass of Ts will have type Class[T]. Queries can create classes. The type of objects in the result class is defined by the query, and all objects in that class have the defined type or are substitutable for that type.

4 Query Algebra

The query algebra provides type specific operations against collections of complex objects with identity. We assume collection objects have type Set[T] or Multiset[T], although the semantics of the operations could be modified to support most other collection types. A query over a Set[T] (Multiset[T]) returns a Set[Q] (Multiset[Q]). Q is defined by the query and is either an existing type or a tuple type. The algebraic operations support the notion of abstract data type, and all access to objects in a collection having type Collection[T] is through the public interface defined for type T.

Most operations return collections of existing database objects. We recognize, however, that existing objects in the database may not explicitly reflect all relationships requested by a query. Thus we provide operators to create new objects to store the requested relationships. The objects are tuples and, as such, are strongly typed. The attribute values are objects from the database, thus maintaining the integrity of the type structure.

The operators preserve the typing of the object-oriented data model. By allowing for operations that produce tuples we are simply observing that the tuple data structure is very useful for holding the result of a dynamically computed relationship. Furthermore, there exists a rich algebra for manipulating these relationships and we can apply well-known optimizations to computations involving these tuples.

The algebra is also concerned with the identity of objects. Two objects are considered to be equal (=) when they are identical, i.e. equality refers to 0-equality. Set membership is based on object identity; a set contains objects with unique identities. In addition, the definition of i-equality can be used to modify the algebraic operations to indicate how much duplication should be allowed in the results of a query (see Section 4.5).

4.1 Select

The *Select* operation creates a collection of database objects which satisfy a selection predicate. The operation is defined as:

$$Select(S, p) = \{s \mid (s \ in \ S) \wedge p(s)\}$$

where S is a collection of objects and p is a predicate defined over the type of the objects in S. If S has type Class[T], then the result is an object of type Class[T] and will be a subclass (restriction) of S. When S is a set, Select returns a set. When S is a multiset, Select returns a multiset.

The operation creates a new collection object containing the identifiers of all members of collection S satisfying the predicate. The predicate is a first-order Boolean formula with variables representing database objects. In the predicate, one variable is bound to an object in the selection set, other variables and constants represent database objects or atomic values.

These constants and variables will usually be properties of bound variables (or constants). For example, if s is a variable having type Supplier (see

Table 1, Section 6), s.address.state is a valid variable. Constants are also built by applying query operations to other constants or bound variables. For example, if Q is a set constant (e.g., a class), Select(Q,p′) could be a set constant in a predicate.

Atoms of the formula are formed using Boolean operations defined for the type of a variable or constant. These include *subset_of* and *in* for set types, and *i-equality* for type Object. For example, if s is a variable of type Supplier and p is a constant representing a Part object, the Boolean operation "p in s.Inventory" is valid since the Inventory property of variable s has type Set[Part] and the *in* (member_of) predicate is defined for arguments of type T and Set[T].

4.2 Image and Project

The *Image* and *Project* operations are used primarily to return components of objects in the collection being queried. The *Image* operation is used to return a single component or value for each object in the queried collection, while the *Project* operation can return multiple components of a single object. A component being returned is determined by applying a function to the objects in the queried collection.

The *Image* operation has the form:

$$Image(S, f : T) = \{f(s) \mid s \ in \ S\}$$

where S is a collection of objects and f returns an object of type T. For S a set (multiset), Image returns a set (multiset). Function f must be defined for the type of objects in S and we assume f does not have any side-effects.

A common way of building functions is to compose the application of properties of objects in S. For example, *Image(Suppliers,*λs *s.address.city)* returns a set of objects representing the cities in which suppliers are located. A function may also be composed of operations defined on the type of objects in the collection being queried. For example, the result of operation $PartsSets := Image(Suppliers, \lambda s \ s.Inventory)$ is a set of sets of parts (Set[Set[P]]) and *Image(PartsSets,* λq *Select(q,p))*, where p is a predicate defined for objects of type P, is a valid operation.

The *Project* operation extends Image by returning multiple components of an object, thus supporting the maintenance of selected relationships between components of an object. The relationships are stored as tuples, with the tuple type defined by the operation as follows:

$$\begin{aligned} Project(S, \quad & <(A_1, f_1), ..., (A_n, f_n)>) = \\ & \{< A_1 : f_1(s), ..., A_n : f_n(s) > \mid s \ in \ S\} \end{aligned}$$

where S is of type Set[T], the A_i's are unique attribute names, and each f_i takes a single input of type T and returns an object of type T_i. (The functions are the same functions used in the Image operation.) The result type of the operation is then Tuple[$<(A_1, T_1), ..., (A_n, T_n)>$]. All properties and operations for type Tuple are thus defined for the objects of the result set.

Project returns one tuple for each object in the collection being queried. These tuples are created by the operation, and all have unique identifiers. Although the type of the collection returned is a set (since all tuples are uniquely identified), the type returned is considered to be Set[Tuple] if input collection S is a set and Multiset[Tuple] if S is a multiset.

The creation of tuple objects by the Project operation can hide i-equality in the result tuples. For example, if we have the following tuple objects:

$$\begin{aligned} t_1 &= < A : i_1, B : i_2, C : i_3 > \\ t_2 &= < A : i_4, B : i_2, C : i_3 > \end{aligned}$$

and object S = $\{t_1, t_2\}$, then the query *Project(S,*$\lambda s <(1, s.B), (2, s.C)>$*)* will return two uniquely identified, shallow-equal tuples. This difficulty is managed through the use of operation $Project_{=i}$ that operates as *Project* but eliminates i-equality from the result collection. This extension will be further discussed in Section 4.5.

The functions that can be applied as arguments in the Image and Project operations make these operations much more powerful than relational projection. We noted earlier that the function argument of Image can be operations defined on the type of objects in the collection queried. This ability might prove useful when updating objects in the database, an area we are not yet considering. The same types of operations can be used as arguments for a Project operation, allowing the creation of tuples with components derived from different collections of objects. For example, $Project(Parts, \lambda p < (P, p), (J, Select(Jobs, \lambda j \ p \ in \ j.PartsNeeded)>)$ creates a new relationship between part objects and the jobs that need those objects. This type of relationship can be obtained using our Ojoin operation (see next section), but is often more naturally expressed using Project.

4.3 Ojoin

Although many relationships between objects are represented by the complex objects themselves, we feel that an explicit join is needed to handle cases when a relationship is not defined in an object type. The *Ojoin* operator is an explicit join operator used to create relationships between objects from two classes in the database. Our definition of Ojoin is designed to preserve the associativity of the operation, which we expect to be important in query optimization.

Ojoin is essentially a relational join, modified to handle sets of objects which may not be tuples. We define:

$$Ojoin(S, R, A_1, A_2, p) = Relational\_Join(S', R', p')$$

Collections S and R must both be sets or both be multisets, A_1 and A_2 are optional parameters (see later) and p is a predicate (as in Select) defined over objects from S and R. If S and R are sets of tuples then S', R', and p' are S, R and p, respectively, A_1 and A_2 are ignored if present, and Ojoin is relational join.

If operand S has type Set[T] where T is some type other than tuple, then that operand is considered to be converted to a set of tuples using the operation $S' = Project(S, \lambda s < (A_1, s) >)$, i.e. each element of set S is represented by a single-attribute tuple. The type of the tuple is $< (A_1, T) >$, where T is the type of the elements in set S. Similarly, $R' = Project(R, \lambda r < (A_2, r) >)$ if necessary. Each reference to s (r) in predicate p is converted to a reference to $s.A_1$ ($r.A_2$) in p'. If not provided, parameters A_1 and A_2 can default to S and R (as attribute names), respectively.

For example, suppose we have a set S containing stack objects and a set Q containing queues. Then $Ojoin(S, Q, \lambda sq\ s.Top =_d q.Front)$ returns a set of stack-queue pairs in which the top of the stack is equal in value (deep-equal) to the front element of the queue.

The Ojoin operation creates new tuples in the database to store the generated relationships. The tuples are strongly typed using the parameterized type Tuple, with each attribute typed according to the type of the collection from which it is derived. The tuples created will have unique identities. When S and R are sets (multisets) the result is a set (multiset) of tuples. There will never be shallow-equal pairs of tuples if S and R are sets but there may be shallow-equal pairs if S and R are multisets.

The objects involved in the relationships are maintained and can be accessed as the value of the appropriate attribute in the tuple. In the previous example, the only operations available on the tuples are those operations defined for tuples, however all stack operations are available for *result.S* and queue operations for *result.Q*.

4.4 Set operations

The algebra includes set operations Union, Difference, and Intersection, operation Flatten (for sets of sets) and operations Nest and UnNest (for sets of tuples). The first three operations are used to create new collections of objects, the latter three generally re-structure collections of objects.

Union, *Difference* and *Intersection* are the usual set operations with set membership based on object identity ($=_0$). If the operands are multisets, the operations return multisets using the natural tests for membership. 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[2]. The types of the objects themselves are not changed; the type of the objects will be the type or a subtype of the type recorded for the collection. Any two sets can be combined, since all types have a common supertype of Object.

The *Flatten* operation is used to restructure sets of sets. We define

$$Flatten(S) = \{r \mid \exists t \ t \ in \ S \wedge r \ in \ t\}$$

where S has type Set[Set[T]], t has type Set[T] and r has type T. The result type is Set[T]. The operation can also support S with type Set[Multiset[T]] with the same type of result. If S is a multiset, the result is a multiset.

The Flatten operation is useful in conjunction with Image. Image can extract components with type set (multiset) from an object; Flatten allows consideration of the set members individually.

Nest and *UnNest* are similar to the same operators for non-First Normal Form (NF2) data models (see for example [Jae82]). They allow the representation of tuples as flat or nested relations. Since the tuple attribute values in our model can be any type of object, *Nest* compares attribute values using object identity and collects values (objects) for the nesting attribute into sets (multisets if the operand is a multiset).

$$\begin{aligned} Nest(S, A_i) = \quad & \{< A_1 : s.A_1, ..., A_i : t, ..., A_n : s.A_n >| \\ & \forall r \exists s (r \ in \ t \wedge s \ in \ S \wedge s.A_i = r)\} \end{aligned}$$

[2]Union and intersection types could be used, but then the queries would be affecting the type structure. Although we do have some effect in other situations (tuple types) we chose to use only existing types here.

Each of the tuples created by Nest has a unique identity, and no two tuples can be shallow-equal. However, it is possible for the A_i attribute values of different tuples to be shallow-equal. We consider this problem in the next section.

The *UnNest* operation essentially undoes the effects of Nest, although it is not an inverse operation. We define:

$$UnNest(S, A_i) = \{< A_1 : s.A_1, ..., A_i : t, ..., A_n : s.A_n >| \ s \ in \ S \wedge t \ in \ s.A_i\}$$

where the A_i attribute of the operand tuple type is set or multiset valued. The result collection will be the same type of collection as S (set or multiset). As for Project, the tuples that are created have unique identities, and thus the result collection may contain shallow-equal pairs of tuples (if A_i was type multiset). Thus we also define *UnNest*$_{=i}$ which works as UnNest but ensures that the result collection will not contain two i-equal tuples.

Nest and UnNest are not inverse operations. If the set being nested contains shallow-equal tuples, that duplication will not be maintained by a Nest followed by an UnNest on the same attribute. For example, if we have tuple objects

$$\begin{aligned} t_1 &= < A : o_1, B : o_2 > \\ t_2 &= < A : o_1, B : o_3 > \\ t_3 &= < A : o_2, B : o_4 > \\ t_4 &= < A : o_1, B : o_2 > \end{aligned}$$

and set $S = \{t_1, t_2, t_3, t_4\}$, Nest(S,B) creates tuples $t_5 =< A : o_1, B : \{o_2, o_3\} >$ and $t_6 =< A : o_2, B : \{o_4\} >$ and forms result $R = \{t_5, t_6\}$. UnNest(R,B) creates three tuples shallow-equal to t_1, t_2 and t_3. Similarly, if a set of tuples R contains two tuples that are shallow-equal on all but set-valued attribute B, Nest(UnNest(R,B)) will contain fewer tuples than set R.

4.5 Duplicate Elimination

The result of a query using the algebra presented thus far is a new collection of objects. The collection will be a newly identified object in the database, and thus there cannot be identical responses to a query. The objects in a result collection may be either existing database objects or new tuple objects created during the operation. The creation of new objects means that multiple objects with unique identifiers may be created when a single object is desired. We define operations *DupEliminate* and *Coalesce* to handle situations where i-equal objects are created by a query. We also combine DupEliminate with operations Project, UnNest and Image to redefine the latter as families of operations $Project_{=i}$, $UnNest_{=i}$ and $Image_{=i}$.

Operation *DupEliminate(S,i)* keeps only one copy of i-equal objects from collection S. Such duplication of objects can be created by operations Image, Project and UnNest. Project and UnNest create newly-identified tuples, by definition, and a function applied by the Image operation may create new objects.

For example, suppose we want to group a collection Jobs by the city in which they are located. One way to do this would be to assemble, for each job in Jobs, a collection of all jobs in the same city as follows:

$$Image(Jobs, \lambda j\ Select(Jobs, \lambda q\ q.city = j.city)) \tag{1}$$

However, this result is not exactly what was desired; there may be many collections containing exactly the same jobs, i.e. many shallow-equal collections. If there are x jobs in a particular city, there will be x collections all containing the same x job objects. However, if DupEliminate is applied to the result as follows

$$DupEliminate(Image(Jobs, \lambda j\ Select(Jobs, \lambda q\ q.city = j.city)), 1) \tag{2}$$

the result would contain the expected collections of jobs with no shallow-equal duplication.

Duplication can be introduced at any level by the algebraic operations. For example, consider an extension to problem 1, where we want to store the city value along with the set of job objects in that city:

$$Project(Jobs, \lambda j < (C, j.city), (J, Select(Jobs, \lambda q\ q.city = j.city)) >) \tag{3}$$

The result of query 3 will be a set of tuples, one for each object in Jobs. Many of those tuples will have duplicate C values, and those that are duplicates will have J attribute values that are shallow-equal. In this case, tuples which would be considered to be duplicates will not be shallow-equal, but will be 2-equal.

Query operations Project, UnNest and Image can be modified with $=i$ as a shorthand for application of operation DupEliminate after application of the operation. Problem 2 could be written, for example, using $Image_{=1}$

(or $Image_{=s}$). It is interesting to note that the application of operation DupEliminate can eliminate duplication created by the algebraic operations, as in the preceding examples, but, if used with a level deeper than that created by the operations, can also be used to eliminate duplication inherent in the data.

Operations such as Project (as in example 3) and Nest can create tuples with *components* (attribute values) that are i-equal (usually shallow-equal). We thus define operation $Coalesce(S, A_k, i)$ which, for collections S of tuple objects, eliminates duplication at level i for the A_k components of the tuples. If two tuples in S, call them t_1 and t_2, are such that $t_1.A_k \ =_i \ t_2.A_k$ then one of the objects is discarded (say $t_2.A_k.value$) and $t_2.A_k$ is assigned the value of $t_1.A_k$.

For example, when *Nest*ing a collection on a given attribute; a new set is created for each tuple in the result, and each new set is a distinct object. However, some of these new set objects may be shallow-equal. Consider the four tuples:

$$\begin{aligned} t_1 &= \ < A : o_1, B : o_2 > \\ t_2 &= \ < A : o_3, B : o_4 > \\ t_3 &= \ < A : o_1, B : o_4 > \\ t_4 &= \ < A : o_3, B : o_2 > \end{aligned}$$

and the operation

$$Nest(S, B) \tag{4}$$

This operation will result in the creation of two set objects(s_1 and s_2), two tuple objects (t_5 and t_6) and the result R as follows:

$$\begin{aligned} s_1 &= \{o_2, o_4\} \ and \\ s_2 &= \{o_2, o_4\} \\ \\ t_5 &= \ < A : o_1, B : s_1 > \ and \\ t_6 &= \ < A : o_3, B : s_2 > \\ \\ R &= \{t_5, t_6\} \end{aligned}$$

The duplication in objects s_1 and s_2 is similar to that created by problem 3, however operation DupEliminate does not apply here since the A attribute values are distinct in the result tuples. We could, however, execute

$$Coalesce(R, B, 1)$$

to modify R by eliminating s_2 and modifying t_6 to have value $< A : o_3, B : s_1 >$.

As with operation DupEliminate, the application of Coalesce could eliminate duplication inherent in the data as well as duplication created by the algebraic operations. Project and Nest can create results with duplicate components, as noted above; any application of Coalesce to results of Ojoin or UnNest, or at a level deeper than that created by Project or Nest, would eliminate duplication that was present in the data before application of the algebraic operation.

5 Maintaining object identity

The presentation of the results of a query is complicated by the presence of object identity. In order to support the identity of existing objects in the database, the operations create new objects in which to store the data objects satisfying a query. As we saw in the last section, though, the creation of new objects may result in duplication in the data presented to satisfy the query. This duplication provides information related to the result of the query by providing, for example, the information needed to quantify the results. However, in many cases the duplication is not needed or desired, and providing information that indicates which parts of the result are identical (i.e. object identity) is required.

In Section 4.5 operations DupEliminate and Coalesce were introduced to solve the problem of creating separate identities for objects considered to be equal. At the same time, families of operations Project$_{=i}$, UnNest$_{=i}$, and Image$_{=i}$ were defined to provide, in single operations, the application of DupEliminate following the basic, or 0-equal, version of the operations. Other operations, such as Select or Ojoin, may result in collections with i-equal objects (tuples or more general objects) only if equality exists in the input data. We chose not to provide new operators that automatically eliminate duplicate values in those cases, although the elimination operators can be applied to any query results.

The creation of newly identified collections to store results of a query poses a problem in maintaining object identity under the application of operations that are primarily concerned with the formatting of results (Flatten, Nest, UnNest, DupEliminate and Coalesce). For operations which actually extract data from the database it is clear that the result should be a new object. However, it is not clear that this is always required, or desired, for the formatting types of operations. Aliasing is intrinsic to the structure of the results, since references to objects are based on object identifiers, so it may be desirable to have formatting types of operations actually modify the objects being formatted rather than creating new objects. On the other hand, modifying data is not usually considered to be a function of querying.

We suggest offering alternative instructions $Flatten'$, $Nest'$, $UnNest'$, $DupEliminate'$ and $Coalesce'$ which operate as for their non-primed counterparts except that they actually modify the input collection rather than creating new collections or tuples. These instructions do not offer new results as far as data satisfying the query is concerned, but help circumvent aliasing types of problems associated with object identity.

6 Examples

We illustrate the operation of the algebra with some example queries on an object-oriented version of a Supplier-Parts-Job database [Dat86]. Types Supplier, Part, Job and Addr are defined in Table 1. Collections Suppliers, Parts and Jobs, with the obvious associated types, will be used for the queries. We assume these collections are sets.

The properties of the types are used as a scheme for each collection and the type of each property determines what basic operations are available for the property. For example, ordered_list operations Empty, Member_of, First, Last, etc. can be applied to the Preferred_Suppliers property of type Job. All set operations for the model (in, subset_of) as well as the set operations defined by the query algebra can be applied to properties having type Set.

Example 6.1 *Find all red parts. Which suppliers can supply all of the red parts?*

P_red := Select(Parts,λp p.color = "Red")
S_Pred := Select(Suppliers,λs P_red subset_of s.Inventory)

The first selection finds the red parts, the second selection finds all suppliers for which the inventory includes that set of parts. The *subset_of* operation is available since property Inventory and result P_red both have type Set[Part].

```
Type Supplier
    properties:
        ident: string
        address: Addr
        Inventory: Set[Part]
    operations:
        RecvOrder:
            Supplier, Set[Part] → Supplier

Type Job
    properties:
        num: string
        address: addr
        PartsNeeded: Set[Part]
        Preferred_Suppliers:
            ordered_list[Supplier]
    operations:
        NewPart: Job, Part → Job
        Cancel: Job

Type Part
    properties:
        Num: string
        address: addr
        color: string
        components:
            Set[<(P,Part),(Qty,Int)>]
        plan: drawing
        BillofMaterial: list[Part]
    operations:
        Order: Part

Type Addr
    properties:
        street: string
        city: string
        state: string
```

Table 1: Types for a Supplier-Parts-Job database.

Example 6.2 *What parts are needed by jobs in Boston?*

BostonJobs := Select(Jobs,λj j.address.city = “Boston”)
BostonParts := Flatten(Image(BostonJobs,λj j.PartsNeeded))

The first operation selects all jobs in Boston and the Image operation extracts the PartsNeeded sets from those jobs. The Flatten operation converts the set of sets of parts into type Set[Part]. If we want to retain information about which parts are needed with which job we could use a Project operation instead of Image:

BosJobParts := Project(BostonJobs,λj <(J,j),(Pt,j.PartsNeeded)>).

Note that operation NewPart (of type Job) cannot be applied to members of BosJobParts (since they have type Tuple), but is appropriate for BosJobParts.J. The results of the queries maintain the instances of type Job and provide multiple paths to Boston-based jobs (through collections Jobs and BostonJobs, and property J of BosJobParts).

Example 6.3 *Generate Bills of Material for all parts needed by Boston jobs.*

BofM := Image(BostonParts,λp BillofMaterial(p))

The BillofMaterial property is probably implemented as a function, but the implementation has no effect on the syntax of the query. If the property is indeed a function, it is applied to each member of BostonParts to create the result collection. The Image operation might also prove useful for updating objects, by allowing application of operations to objects.

Example 6.4 *Find all local suppliers for each job.*

LocalS := Ojoin(Jobs,Suppliers,J,S,λjs j.address.city = s.address.city)

The result is a set of tuples having type <(J,Job),(S,Supplier)>. This result is similar to a normalized relation. An application of operation $Nest(LocalS, S)$ would give a set of suppliers for each job, i.e. a collection of tuples having type $< (J, Job), (S, Set[Supplier]) >$. A similar result can be accomplished using

Project(Jobs,λj (<(J,j),
(S,Select(Suppliers,λs s.address.city =j.address.city))>))

For each job the set of suppliers in the same city is selected from the collection of all suppliers. However, if there were no local suppliers for a job this would result in tuples with null values (an empty set) for the S attribute. Although Project, with embedded Select, can simulate Ojoin (with formatting differences handled by UnNest and tuples with empty sets removed by Select), we offer both operators for user clarity as well as optimization potential.

Example 6.5 *Find the preferred local suppliers for each job.*

We assume that type ordered_list has a Boolean operation *Member_of* which can be applied to objects of that type, and select from result LocalS of the previous example.

Select(LocalS,λr r.S *Member_of* r.J.Preferred_Suppliers)

The result could also be obtained in a single Ojoin operation by conjoining the selection predicate of this example (with r.S changed to s and r.J to j) with the predicate of the Ojoin in Example 6.4.

6.1 Representing Relations

Querying a relational database can be modelled using our query algebra and object-oriented data model. Normalized relations are easily represented in our model as collections of type Set[Tuple] where all attributes of the tuple have atomic types. The relational algebraic operations map directly to our algebra applied to sets of tuples.

Conversely, the properties of objects in our model can be mapped to objects similar to relations using the Project operation. The tuples representing objects will retain, as attribute types, the property types. These attribute types could be treated as atomic types (e.g., Part.plan) or could, in some cases, be further expanded using UnNest and Project. For example, the following query creates a relation from the Suppliers class:

$$
\begin{aligned}
\text{NNR\_Suppliers} := \text{Project}(\text{Suppliers},\lambda s\ <&(\text{ident},\text{s.ident}),\\
&(\text{address},\text{s.address}),\\
&(\text{Inventory},\ \text{s.Inventory})>)
\end{aligned}
$$

with tuple type $<(ident, string), (address, Addr), (Inventory, Set[Part] >$. The relation could be further expanded, if desired, using Project to separate the address into its component parts (street, city and state) and UnNest to create tuples where the type of Inventory is Part instead of Set[Part].

7 Query Equivalence

We would like to identify transformations of queries that produce equivalent queries. Such transformations would provide a method for identifying redundant algebraic operations and would also provide strategies for query optimization. The result of a query is a collection of objects, thus equivalence of queries is closely related to equality of objects; equivalent queries should produce result objects that are, in some sense, equal.

We can identify two notions of equality for objects. The first type is i-equality, introduced in Section 3. This equality refers to an equivalence between data values; if two objects are i-equal they contain the same data. However, in order to detect the similarities in data, i-equality ignores the identities of the objects in which the data is stored. We can define a stronger notion of equivalence, called id-equality, which can detect structural differences in objects. For example, consider a collection of tuples, call it S, and another collection R = Coalesce(S,A,i). S and R are i-equal, but they are not id-equal. The A components of two tuples in S could be i-equal while the A components of the corresponding tuples in R are 0-equal.

Objects can be represented graphically, where nodes are labelled with object identifiers or atomic values, arcs connect collection type objects to all objects in the collection, and arcs also connect non-collection type objects to the values of all properties of that object (similarly to [Kho86] for example). If a collection is a multiset, there will be multiple arcs from the identifier node for the collection to the node representing the multiply occurring object. Similarly, if the values of more than one property of an object are identical there will be multiple arcs between the nodes representing the object and its components' value.

We define two objects to be *id-equal at depth i*, written $\simeq_i$, if they are i-equal and their graphical representations are isomorphic. I-equality is necessary, but not sufficient, to ensure id-equality.

We define two queries to be *id-equivalent at depth i* ($\cong_i$) if, when applied to an arbitrary database, they return collections of the same type of object and those collections are id-equal. Similarly, two queries are *data-equivalent at depth i* ($\equiv_i$) if they return i-equal collections. Thus, id-equivalence implies data-equivalence. As is the case with i-equality, either equivalence at depth i implies the same equivalence for all depths greater than i. We also note that $\cong_i$ and $\equiv_i$ are equivalence relations, and that

- A $\equiv_i$ B and B $\equiv_j$ C $\Longrightarrow$ A $\equiv_k$ C where k = max(i,j)
- A $\cong_i$ B and B $\cong_j$ C $\Longrightarrow$ A $\cong_k$ C where k = max(i,j)

We will use $\cong$ and $\equiv$ to mean equivalence is present for some i.

As an example of query equivalence, consider the following queries, both of which match each object in set As with a set of objects from set Bs according to predicate p. In Query 1, the Project operation does the matching and the outermost Select operation eliminates tuples from that result which have empty B attribute value (i.e. tuples indicating no Bs objects match an object from As). In Query 2, As and Bs are joined according to predicate p then nested on the B attribute.

Query 1: $Select(Project(As, \lambda a < (A, a), (B, Select(Bs, \lambda b\ p(a,b))) >), \lambda t\ NOT\ Empty(t.B))$

Query 2: $Nest(Ojoin(As, Bs, A, B, p(a,b)))$

Queries 1 and 2 are data-equivalent at depth 3 since the collections are unique (depth 0), the tuples in those collections have unique identities (depth

$$
\begin{array}{lll}
1.\ \ Ojoin(As, Bs, A, B, p(a,b)) & \cong_2 & Select(UnNest(Project(As, <(A,a),(B,Bs)>), B) \\
 & & \quad \lambda t\ p(t.A, t.B)) \\
2.\ \ Intersection(A, B) & \cong_1 & Difference(A, Difference(A, B)) \\
3.\ \ Flatten(S) & \cong_1 & Image(UnNest(Project(S, \lambda s <(A, s)>), A), \lambda a\ a.A) \\
4.\ \ Nest(S, A_i) & \cong_2 & Project(SminusA_i, \\
 & & \quad \lambda t <(A_1, t.A_1), \ldots, (A_{i-1}, t_{i-1}), \\
 & & \qquad (A_i, Image(Select(S, \lambda s\ (s.A_1 = t.A_1) \wedge \ldots \\
 & & \qquad\qquad \wedge(s.A_{i-1} = t.A_{i-1}) \\
 & & \qquad\qquad \wedge(s.A_{i+1} = t.A_{i+1}) \wedge \\
 & & \qquad\qquad \ldots \wedge (s.A_n = t.A_n)), \\
 & & \qquad\quad \lambda q\ q.A_i)), \\
 & & \qquad (A_{i+1}, t.A_{i+1}), \ldots, (A_n, t.A_n)>) \\
\qquad \text{where} & & \\
\qquad SminusA_i & := & Project_{=s}(S, \lambda s <(A_1, s.A_1), \ldots, (A_{i-1}, s.A_{i-1}), \\
 & & \qquad (A_{i+1}, s.A_{i+1}), \ldots, (A_n, s.A_n)>) \\
5.\ \ Coalesce(S, A_k, i) & \cong_3 & Project(Ojoin(S, NoDupA_k, , K, \lambda ab\ a.A_k =_i b), \\
 & & \quad \lambda t <(A_1, t.A_1), \ldots, (A_{k-1}, t_{k-1}), \\
 & & \qquad (A_k, t.K), \\
 & & \qquad (A_{k+1}, t.A_{k+1}), \ldots, (A_n, t.A_n)>) \\
\qquad \text{where} & & \\
\qquad NoDupA_k & := & Image_{=i}(S, \lambda s\ s.A_k)
\end{array}
$$

Table 2: Id-equivalent Operations

1), and corresponding tuples are not shallow-equal (depth 2) but their B components *are* shallow-equal (depth 3). They are also id-equivalent at depth 3 since the Nest following the Ojoin will build the same structure as the Project with nested Select.

7.1 Query Transformations

The algebra can be divided into two categories of operations:

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 identities: Flatten, Nest, UnNest, DupEliminate, Coalesce, and their primed counterparts

Operation families $\text{Project}_{=i}$ and $\text{Image}_{=i}$, for i not 0, fall into both categories. For transformations involving only operations in the first category, id-equivalence is the same as data-equivalence. This is not necessarily true for transformations involving operations in the second category, since such transformations can involve structural modifications.

The equivalences in Table 2 illustrate that the algebraic operations Ojoin, Intersection, Flatten, Nest and Coalesce can be considered to be redundant since they are id-equivalent to queries composed using other algebraic operations. The Nest operation, for example, is id-equivalent to applications of Project, Image and Select. First, a collection ($\mathrm{SminusA}_i$) is created containing tuples with all attributes except the nesting attribute. No two tuples in the collection are shallow-equal. For each of these tuples, all matching tuples from the original collection are selected. Finally, the nesting attribute value is extracted from these tuples (using Image), creating a result set for the tuple attribute.

The Image, Ojoin and Project operations can be combined in a similar manner to create a query that is id-equivalent to Coalesce. The Image operation is used to create a collection of unique A_k attribute values (NoDupA_k) which is matched with the original collection of objects using Ojoin. The result tuples of Ojoin will contain two copies of the values of the attribute being coalesced; the original attribute value (A_k) and a value that will be uniquely identified across all tuples (K). The final Project operation will choose the second version of the value, resulting in consistency between tuples on the A_k attribute value.

The operations in category 2 are id-equivalent to their primed counterparts. Flatten and DupEliminate are id-equivalent at depth 1 to Flatten$'$ and DupEliminate$'$, respectively. The depth 1 equivalence reflects the fact that the identifiers of the result collections are different. UnNest and Coalesce are id-equivalent at depth 2 to their primed counterparts and Nest is id-equivalent at depth 3 to Nest$'$. The depth 2 and 3 equivalences result from the creation of newly identified tuples and collections by each operation.

The id-equivalence of these transformations suggest that a reduced operator set could include operators Select, $\mathrm{Image}_{=0}$, $\mathrm{Project}_{=0}$, Union, Difference, $\mathrm{UnNest}_{=0}$ and DupEliminate. The i-equality modifiers are not necessary for Image, Project and UnNest as long as the 0-equality modifier is present. Higher levels of equality can be achieved by applying operation DupEliminate to a result.

It is also possible to identify Select and Ojoin queries whose predicates can be simplified and replaced using algebraic query operators. For example,

$$\begin{array}{ll} Select(S, \lambda s\ \exists j & (j\ in\ K) \wedge p(s,j)) \ \cong_1 \\ & Image(Select(MatchK, \lambda t\ \neg Empty(t.B)), \lambda t\ t.A) \\ \text{where } MatchK & := Project(S, \lambda s < (A, s), (B, Select(K, \lambda j\ p(s,j))) \end{array}$$

and K is a bound variable representing a collection. The selection nested in the Project operation finds any j's in K that satisfy the predicate. The outer Select operation performs the existence test and the Image extracts the selected objects out of the tuples where they were stored by Project. Similarly,

$$\begin{aligned} Select(S, \lambda s\, \forall j\ (j\ in\ K) \wedge p(s,j)) &\cong_1 Select(S, \lambda s\ Empty(AllinK)) \\ \text{where } AllinK &:= Difference(K, Select(K, \lambda j\ p(s,j))) \end{aligned}$$

Again, the quantifier is replaced by an existence test; if the Difference between collection K and the members of K satisfying the selection predicate is empty, then all members of K satisfied the predicate.

In general, queries with first order predicate arguments cannot be transformed into queries without such arguments, since the *j in K* conjunct can not necessarily be moved within the predicate (K may not be bound properly). The relationship between first-order predicates in Select and Ojoin that can be transformed into other algebraic computations, and those queries that we want to allow in the algebra, is still being studied.

8 Implications for Optimization

We expect the structure of the operations and the redundancies in the operator set presented here to offer many opportunities for optimization. Query optimization may also depend on the implementations of the types queried, but we do not consider that aspect yet. In addition, the different types of equivalence introduced in Section 7 suggest that we may want to allow optimization guaranteeing id-equivalence or just data-equivalence.

The similarities between our algebra and relational algebra imply that relational optimization results may prove useful in object-oriented query optimization. For example, one relational optimization strategy is to push Selection past Join. That same strategy is available in our algebra (see identity 1 Table 3). 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 (Table 3 identity 2). These two ideas are combined in identity 3 of Table 3. 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.

The redundancies presented in Section 7 may also 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. The primed and non-primed operations may also allow optimization in intermediate query results.

$$Select(Ojoin(As, Bs, A, B, p), \lambda t\ p_s(t.A)) \cong$$
$$Ojoin(Select(As, p_s(a)), Bs, A, B, p) \quad (1)$$
$$Select(Ojoin(As, Bs, A, B, p(a, b)), \lambda t\ p_s(t.A, t.B)) \cong$$
$$Ojoin(As, Bs, A, B, p(a, b) \wedge p_s(a, b)) \quad (2)$$
$$Ojoin(As, Bs, A, B, \lambda ab\ p(a) \wedge p'(a, b)) \cong$$
$$Ojoin(Select(As, p(a)), Bs, A, B, \lambda ab\ p'(a, b)) \quad (3)$$

Table 3: Some Select-Join Identities

Another optimization consideration for object-oriented databases is the use of object identifiers, as opposed to the objects themselves, whenever possible. The Project operation should prove useful in such optimizations by allowing the extraction of identifiers and atomic values needed by subsequent operations. For example, consider the Ojoin of example 6.4 in Section 6 which creates set LocalS containing suppliers and jobs matched by city. The join could involve multiple accesses to each supplier, job, and address object to obtain the city information. However, the sequence of operations:

JC := Project(Jobs,λj <(J,j),(JCity,j.address.city)>)
SC := Project(Suppliers,λs <(S,s),(SCity,s.address.city)>)
Temp := Ojoin(JC,SC,,,λjs j.JCity = s.SCity)
LocalS := Project(Temp,λt <(J,t.J),(S,t.S)>)

accomplishes the same result with only one access to each of the job and supplier objects. The information required for the join (i.e., object id and associated city value) is stored in tuples in collections JC and SC.

9 Summary

The support of type inheritance, abstract data types, object identity, and typed collections by the Encore data model is integral to the design of the query algebra proposed in this paper. The distinction between types and collections in the model allows the algebra to query over collections, using a scheme for each collection described by the type structure. The algebraic operations return collections of objects and the type of those collections can be statically determined using the type structure. Objects in a collection will have the type defined for the collection, or will have a subtype of the type defined for the collection.

The query algebra supports abstract data types by accessing objects only through the interface defined by their type. Results of queries are collections of existing objects or collections of tuples, built by the query, having existing objects as components. The manufacture of tuples by the Project and Ojoin operations allows the expression of relationships between objects not identified by the type design.

The manipulation of objects by the algebraic operations is based primarily on object identity, and most operations create new objects (tuples, collections) to store query results. The algebra supports object identity by maintaining the identities of existing objects and by providing operations to modify, if desired, the identities of objects created to store results. The support for object identity leads to distinctions between data-equality and identifier-equality of objects, and to related levels of equivalence in query results.

Our current research involves the optimization of queries using the algebra presented in this paper. The structure of the operations and the operator set offer many opportunities for optimizations that are independent of the implementation of the types queried. We expect to combine those with type-dependent optimizations to provide efficient query access to data in our object-oriented database.

We are also investigating techniques for estimating the cost of query expressions. The encapsulation provided by the object model makes this a non-trivial problem.

References

[Alb85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *Communications of the ACM*, 10(2):230–260, June 1985.

[Ban87a] Francois Bancilhon et al. FAD, a Powerful and Simple Database Language. In *Proceedings of the 13th VLDB Conference*, pages 97–105, 1987.

[Ban87b] Jay Banerjee et al. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Ban88] 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.

[Car88] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg. A Data Model and Query Language for EXODUS. In *SIGMOD Proceedings*, pages 413–423. ACM, June 1988.

[Dad86] P. Dadam et al. A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. In *SIGMOD Proceedings*, pages 356–367. ACM, May 1986.

[Dat86] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA., 4th edition, 1986.

[Gra88] Goetz Graefe and David Maier. Query Optimization in Object-Oriented Database Systems: A Prospectus. In *Advances in Object-Oriented Database Systems*, pages 358–363. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Jae82] 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.

[Kho86] 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.

[Kor88] Henry F. Korth. Optimization of Object-Retrieval Queries. In *Advances in Object-Oriented Database Systems*, pages 352–357. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Lec88] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *SIGMOD Proceedings*, pages 424–433. ACM, June 1988.

[Mai87] 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.

[Osb88] S. L. Osborn. Identity, Equality and Query Optimization. In *Advances in Object-Oriented Database Systems*, pages 346–351. 2nd International Workshop on Object-Oriented Database Systems, September 1988.

[Row87] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES Data Model. In *Proceedings of the 13th VLDB Conference*, pages 83–96, 1987.

[Shi81] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Zan83] Carlo Zaniolo. The Database Language GEM. In *SIGMOD Proceedings*, pages 207–218. ACM, May 1983.

[Zdo86] S. B. Zdonik and P. Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the Hawaii International Conference on System Science*, January 1986.

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