A MODEL OF QUERIES FOR OBJECT-ORIENTED DATABASES

Won Kim

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

Abstract

One major source of confusion, and consequent criticisms, about object-oriented databases has been the lack of a comprehensive model of queries. Although there is a reasonable degree of agreement about an object-oriented data model, few operational systems support a query model for object-oriented databases. In this paper, we present a rather comprehensive query model which is consistent with object-oriented concepts embodied in the object-oriented data model. The model takes into account the semantics of the class hierarchy and nested objects, and as such is inherently richer than the relational or nested relational model of queries. A significant subset of the model has been cast into a query language which is supported in the ORION object-oriented database system.

1. INTRODUCTION

An object-oriented database system is a database system which directly supports an object-oriented data model. Further, an object-oriented data model includes a number of concepts found in many object-oriented programming languages and knowledge representation languages. An object-oriented database system must provide persistent storage for the objects and their descriptors (schema). The system must provide the user with an interface for schema definition and modification, and for creating and accessing objects by sending messages to the objects.

This basic system may be extended in several dimensions, just as several types of features had to be added to the relational database systems during the past decade. These features include a declarative query language; integrity features, such as transaction management and triggering; perform ance-related features, such as secondary indexing and clustering; and concurrency control and authorization for a multi-user environment.

One of the important shortcomings in most object-oriented database systems operational today is the query capabilities. In most systems, the query model captured in the query language fails to take into account some of the fundamental object-oriented concepts. Some systems, such as VBase [ANDR87], merely provide an SQL relational query interface. Some attempt to provide a new query language which is backward-compatible with a relational query language: SQL in the case of IRIS [FISH87], and QUEL [STON76] in the case of POSTGRES [STON86, ROWE87]. Other systems, such as ORION [BANE87a, KIM88a, KIM89] and GemStone [MAIE86], support a new query language which is based more on the nested-relational model [MAKI77, ABIT84, ABIT86, JAES82, IEEE88].

[BANE88] presents a query model for object-oriented databases and compares it with the relational query model; the model is subsequently extended in [KIMK89] to better account for cyclic queries. Although the model is limited in a number of important ways, to our knowledge it is the first attempt to define a query model for object-oriented databases which is *consistent* with basic object-oriented concepts. The model takes into account the two-dimensional nature of the schema of an object-oriented database: a class has a number of attributes whose types (domains) are also classes, and a class may have a number of superclasses and a number of subclasses.

The model of [BANE88, KIMK89] is not comprehensive: it does not include any consideration for operations comparable to relational joins, set operations, and views. Further, the model, even in its limited form, has not been adequately formalized. The objectives of this paper are to extend the query model to a significantly more comprehensive model, and to attempt to formalize the extended model.

2. CORE OBJECT-ORIENTED DATA MODEL

In this section we will review a core object-oriented data model, which is based on what we consider a minimal set of fundamental object-oriented concepts, and emphasize the importance of these concepts from a database perspective. A more detailed discussion of the minimal set of object-oriented concepts is given in [KIM88b] (in fact, part of this section has been taken from [KIM88b]). The reader will need to clearly understand this section in order to follow the query model we will develop.

object and object identifier

In object-oriented systems and languages any entity is uniformly modeled as an object. Further, an object is associated with a unique identifier; an identifier is not reused even when the object with which it was associated is deleted from the system. The uniform treatment of any entity as an object simplifies the user's view of the world. The object identifier is used to pin-point an object to retrieve.

attributes and methods

Every object encapsulates a state and a behavior. The state of an object is the values for the attributes of the object, and the behavior of an object is the set of methods

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

which operate on the state of the object. The value of an attribute of an object is also an object in its own right. Further, an attribute of an object may take on a single value or a set of values.

class

All the objects which share the same set of attributes and methods are grouped into a higher level object called a class. An object must belong to only one class as an instance of that class. The relationship between an object and its class is the familiar instance-of relationship. A class is similar to an abstract data type. A class may be a primitive class, such as integer, string, and boolean; a primitive class is a class with no attributes.

The concept of a class as a set of instances provides the basis on which a query may be formulated. In relational databases, a query is issued against a relation or a collection of relations; similarly, in object-oriented databases, a query must be issued against a class or a collection of classes. Without the notion of a class, or a set of instances, to aggregate together related objects, it will be very difficult to conceptualize (and evaluate) a query.

The value of an attribute of an object, since it is necessarily an object, also belongs to some class. This class is called the domain of the attribute of the object. The domain of an attribute may be any class, including a primitive class. This represents a significant difference from the normalized relational model in which the domain of an attribute is restricted to a primitive class. The fact that the domain of an attribute may be an arbitrary class gives rise to the nested structure of the definition of a class. That is, a class consists of a set of attributes (and methods); the domains of some or all of the attributes may be classes with their own sets of attributes; and so on. Then the definition of a class results in a directed graph of classes rooted at that class; as in [KIM88b] we will call this graph a *class-composition hierarchy*. If the graph for the definition of a class is restricted to a strict hierarchy, the class becomes a nested relation.

class hierarchy and inheritance

Object-oriented systems allow the user to derive a new class from an existing class; the new class, called a subclass of the existing class, inherits all the attributes and methods of the existing class, called the superclass of the new class, and the user may specify additional attributes and methods for the subclass. A class may have any number of subclasses. Some systems allow a class to have only one superclass, while others allow a class to have any number of superclasses. In the former, a class inherits attributes and methods from only one class; this is called *single inheritance*. In the latter, a class inherits attributes and methods from more than one superclass; this is called *multiple inheritance*. In a system which supports single inheritance, the classes form a hierarchy, called a class hierarchy. If a system supports multiple inheritance, the classes form a rooted directed acyclic graph, sometimes called a class lattice.

The concept of a class hierarchy is completely orthogonal to that of a class-composition hierarchy. A class hierarchy captures the generalization relationship between one class and a set of classes specialized from it. A class-composition hierarchy has nothing to do with inheritance of attributes and methods. Figure 1 shows an

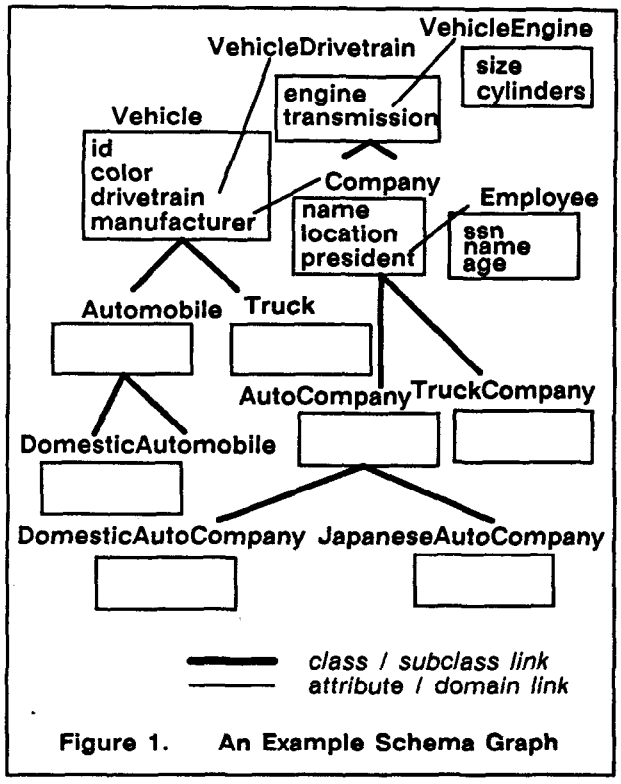

Figure 1. An Example Schema Graph

example schema. The class Vehicle is the root of a class-composition hierarchy which includes the classes VehicleDrivetrain, VehicleEngine, Company, and Employee. The class Vehicle is also the root of a class hierarchy involving the classes Automobile, DomesticAutomobile, and Truck. The class Company is in turn the root of a class hierarchy with subclasses AutoCompany, JapaneseAutoCompany, DomesticAutoCompany, and TruckCompany. It is also the root of a class-composition hierarchy involving the class Employee.

3. QUERY MODEL

The query model implemented in ORION, despite its limitations, is the first to be based on a consideration of the power and constraints of object-oriented concepts. The model restricts the target of a query to a single class or a class hierarchy rooted at that class. This is an important restriction, since this excludes operations comparable to relational joins and set operations. However, the model explicitly takes into consideration some of the important consequences of object-oriented concepts. First, it allows the user to use the directed graph model of the definition of the target class for specifying a query; predicates may be applied to any attributes of any classes on the graph. This is similar to the nested-relational extensions of the relational selection operation [JAES82, DESH88]. Second, a query may be directed against a single class or a class hierarchy rooted at the class. This is important, since a class hierarchy captures the IS-A relationship between a class and all its subclasses; and as such instances of a class may be regarded as belonging to the class and all classes on the superclass chain starting from the class. In

fact, the domain of an attribute of a class is the specified class and all direct and indirect subclasses of the class.

In this section we will use the model of [BANE88] as the basis for what we hope to be a comprehensive query model for object-oriented databases. We will define our query model for a set of operations defined for relational database systems, namely, selection, projection, join, and set operations. We hasten to note that, although we will use the relational terminology, the semantics of these operations are rather different from those used in relational systems.

In Section 3.1 we first define our model for a single-operand query, that is, a query whose target is only one class or a class hierarchy rooted at one class; the relevant operations are those comparable to relational restriction and projection. In Section 3.2, we will extend the model to account for a multiple-operand query, that is, a query which captures operations comparable to relational joins and set operations. The current implementation of ORION includes single-operand acyclic queries. In the remainder of this paper, we will use, without any detailed explanation, a rather intuitive syntax for a query language for illustrative purpose. We restrict the scope of this paper to a presentation of an object-oriented query model, and defer to a future paper a full description of the query language that captures the model.

3.1 SINGLE-OPERAND QUERIES

schema graph

As we have seen, the definition of a nonprimitive class forms a two-dimensional directed graph of classes which we will call a *schema graph* for the class. Figure 1 is an example of a schema graph. The following is a formal definition of the schema graph SG for a class C.

1. SG is a rooted directed graph consisting of a set of classes N and a set of arcs E between pairs of classes. C is the root of SG, and is a nonprimitive class.
2. E has two types of arcs. One is between a pair of classes C1 and C2, such that C2 is the domain of one of the attributes of C1. Another is between a pair of classes C1 and S1, such that S1 is an immediate subclass of C1. The direction of an arc is from a class to the domain of its attribute, and from a class to its subclass.
3. An arc from an attribute of a class to the domain of the attribute may be either single-valued or set-valued. A single-valued arc means that the attribute can have only one value from its domain; while a set-valued arc means that the attribute has as its value a set of instances of its domain.
4. The set of arcs from any class Ci on SG to the direct and indirect subclasses of Ci forms a directed acyclic graph; that is, the class hierarchy rooted at Ci on SG is a directed acyclic graph.
5. The set of arcs from the root class C to the direct and indirect domains of the attributes of C forms a directed graph rooted at C; that is, the class-composition hierarchy of SG is a rooted directed graph which may be cyclic.

query graph

In relational databases, the schema graph for a relation is the relation itself. The selection operation on a relation, that is, a single-relation query on a relation with a Boolean combination of predicates on the attributes of the relation, identifies the tuples of the relation which satisfies all the predicates. In terms of the schema graph, the predicates apply on the lone node of the graph. To define the selection operation on a class in an object-oriented database, we extend this observation. The selection operation on a class C retrieves instances of the class C which satisfies a Boolean combination of predicates on a subgraph of the schema graph for C; we will call such a graph a *query graph*.

Let us consider an example query and its corresponding query graph. Using the schema for the class Vehicle shown in Figure 1, we may formulate a query to "find all blue vehicles manufactured by a company located in Detroit and whose president is under 50 years of age" as follows.

Q1: (**select** Vehicle (Color = "blue"
and Manufacturer Location = "Detroit"
and Manufacturer President Age < 50))

The query graph for this query on the class Vehicle is shown in Figure 2; it is a subgraph of the schema graph for the class Vehicle.

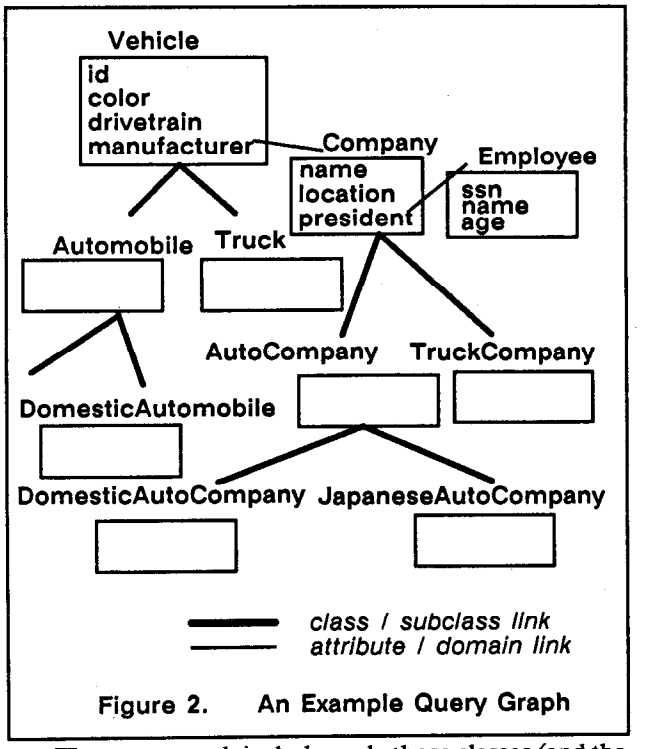

Figure 2. An Example Query Graph

The query graph includes only those classes (and the class hierarchies rooted at them) on which the predicates of the query are specified (as we will show shortly, this is not completely accurate). A predicate may be one of two types: a simple predicate and a complex predicate. Note that we define these terms differently from [BANE88]. A *simple predicate* is of the form < attribute-name operator

value >. The value may be an instance of a primitive class (string, integer, etc.) or an object identifier of an instance of some class. The latter is important, since it may be used for testing the object equality [KHOS86], that is, equality of referenced objects. This is excluded by an oversight from the definition of a simple predicate given in [BANE88]. The predicate (Color = "blue") in the example query above is a simple predicate.

Another type of predicate is a *complex predicate*; this concept is also explored in [ZANI83]. The definition of a complex predicate given in [BANE88] as a predicate on a complex attribute of a class is inaccurate. A complex predicate is actually a predicate on a contiguous sequence of attributes along a branch of the class-composition hierarchy of a class. The predicate (Manufacturer Location = "Detroit") in the above example is a complex predicate. All classes (and the class hierarchies rooted at them) to which any of the attributes in the sequence of attributes specified in a complex predicate belong are included in the query graph.

So far we have assumed a predicate to simply be an expression of the form <attribute-name operator value>, without specifying what the operator is. In traditional database systems the operator is a scalar comparison operator (=, <, >, etc.) or a set comparison operator (contained-in, contains, set-equality, etc.). In object-oriented systems, the user may define methods on a class, a method may be used for any part of a predicate, that is, as the attribute-name, the operator, or the value. The use of a method in a predicate, as an attribute-name or as an operator, cause difficulties with query compilation and optimization.

Next, since any subclass of the domain of an attribute is also a valid domain of the attribute, it is certainly useful to be able to specify in a query a subclass of the domain of an attribute as the domain of the attribute. In other words, an arc in a schema graph from an attribute of a class to the domain of the attribute may be changed in the query graph to one from the attribute to any subclass of the domain. For example, it may be useful to be able to change the example query Q1 such that the domain of the attribute Manufacturer is the class DomesticAutoCompany, rather than the class Company as specified in the schema. The query model presented in [BANE88] implies that the domain of an attribute in the schema is necessarily preserved in queries, thereby placing an unnecessary restriction on the expressiveness of a query.

The following is a formal definition of a query graph QG for a single-operand query on a class C. The result of a single-operand query is a set of instances from the scope of the target class which satisfy the query predicates.

1. QG is a connected subgraph of the schema graph SG for C. C is the root of QG; that is, QG for C and SG for C have the same root. C is a nonprimitive class.
2. QG includes only those nodes of the corresponding SG on which at least one predicate of the query is specified.
3. An arc from an attribute of a class to the domain in SG may be changed in QG to one from the attribute to a subclass of the domain. Then only the class hierarchy rooted at the new domain is included in QG.
4. The set of arcs from the root class C to the direct and indirect domains of the attributes of C included in QG form a directed graph rooted at C; that is, the class-composition hierarchy of QG is a rooted directed graph, in which some branches contain cycles and others do not. (The nature of the cycles will be described further below.)
5. The leaf node of an acyclic branch has only simple predicates on it. The interior nodes of any branch (cyclic or acyclic) may have both simple and complex predicates defined on them.

cyclic branch in a class-composition hierarchy

Now we will characterize precisely the cyclic branch of the class-composition hierarchy (either in a query graph or in a schema graph). A branch of a class-composition hierarchy is cyclic, if it contains a class Ci and a class Cj on the branch, such that Cj is the (indirect) domain of an attribute of Ci, and Ci is the domain of an attribute of Cj; or Cj is the (indirect) domain of an attribute of Ci, and a superclass or a subclass of Ci is the domain of an attribute of Cj. This definition brings together four different types of cycle shown in Figure 3. Two of them, namely,

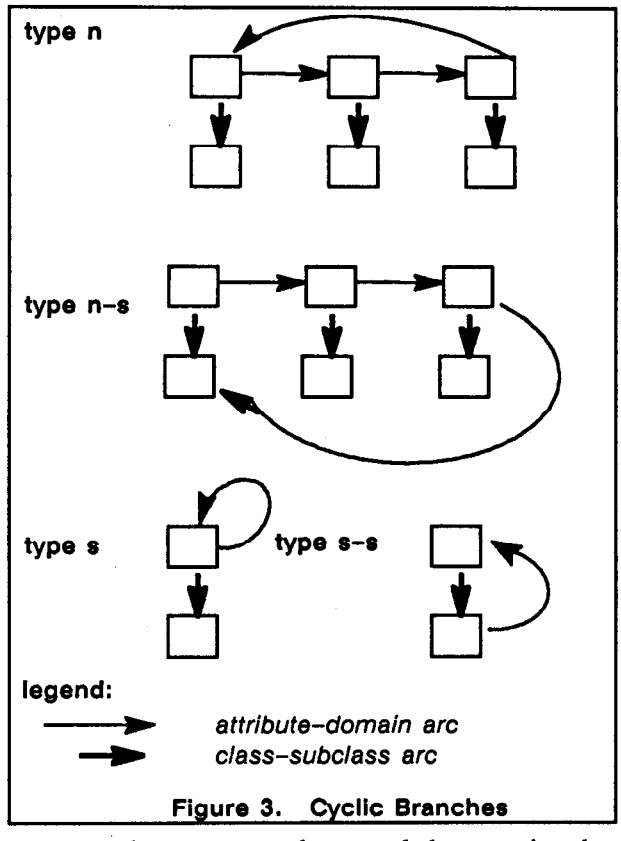

Figure 3. Cyclic Branches

type-ns and type-ss, may be regarded as quasi-cycles, since they are not cycles in conventional sense. They may be viewed as the type-n and type-s cycles, respectively, with additional conditions on the query result (we will illustrate this shortly). The following defines the four types of cycle for a branch of a class-composition hierarchy. We will assume that the branch has n nodes, such that each node may in turn be the root of a class hierarchy.

1. A type-n cycle is a cycle formed by n > 1 nodes on the branch.
2. A type-ns cycle is a quasi-cycle corresponding to a type-n cycle. It is formed by n > 0 nodes on the branch, and a superclass or a subclass of one of the n nodes.
3. A type-s cycle is a cycle formed for a single node.
4. A type-ss cycle is a quasi cycle corresponding to a type-s cycle. It is formed by a class and its superclass or a subclass.

Assuming that the class Employee has an additional attribute Drives whose domain is the class Vehicle, a meaningful query is to "find all blue vehicles driven by the president of the company that manufactured them." This is an example of a query with a type-n cycle in its query graph. The following is an intuitive syntax for expressing the query. We leave it as a simple exercise for the reader to construct the query graphs for the following example queries.

```
Q-c1: (select (Vehicle :V)
         ( Color = "blue"
           and Manufacturer President Drives = V))
```

In this query, the variable :V is used for binding each instance of the class Vehicle, as the set of instances of Vehicle is scanned one at a time. The class Vehicle is the target class.

A query to "find all blue vehicles manufactured by a company whose president drives a Japanese automobile" is an example of a query with a type-ns cycle in the query graph. This query may be expressed as follows.

```
Q-c2: (select Vehicle
         (:J is-a JapaneseAutomobile)
         ( Color = "blue"
           and Manufacturer President Drives = J))
```

The expression (:J is-a JapaneseAutomobile) adds a constraint to the value the complex attribute (Manufacturer President Drives) may take on to instances of the class JapaneseAutomobile. Note that the class Vehicle is still the target class.

For the next two examples, let us add the Manager attribute to the class Employee; the domain of Manager is the class Employee. The following is an example query with a type-s cycle. The query is to find "all managers of an employee named Johnson."

```
Q-c3: (select Employee
         (recurse Manager)
         ( Name = "Johnson"))
```

The expression (recurse Manager) specifies that, once an instance of the class Employee is found which satisfies the predicate (Name = "Johnson"), the recursive values of the Manager attribute of the instance are retrieved.

For our final example, let us assume that the class Employee has a subclass FemaleEmployee. The following is an example query with a type-ss cycle, which finds "all female managers of an employee named Johnson."

```
Q-c4: (select Employee
         (recurse Manager :M
           (:M is-a FemaleEmployee))
         ( Name = "Johnson"))
```

The expression (:M is-a FemaleEmployee) restricts the result of the query to instances of the class FemaleEmployee.

query result

In relational databases, the result of a single-relation query is a subset of the tuples of the relation that satisfies the selection predicates. The concept of a class hierarchy in object-oriented databases captures the generalization abstraction, which means that a class, when used as a generalized concept, subsumes all its subclasses. This observation leads to two equally valid interpretations for the access scope of a query; the access scope of a query on a class C is either the set of instances of C, or the set of instances of the entire class hierarchy rooted at C. Therefore, the result of a single-operand query on a class C may be the set of instances of the class C or the set of instances of the entire class hierarchy rooted at C which satisfy the query predicates. For example, the access scope of a query against the class Vehicle may be restricted to the instances of Vehicle, or may be interpreted to also include the instances of all types of Vehicle, that is, all subclasses of Vehicle.

If a query involves a projection operation, only the desired attributes in the instances that satisfy the predicates will be returned, regardless of whether the access scope of the query is a single class or a class hierarchy. If the scope is a class hierarchy, we need to take into consideration the consequences of inheritance of attributes.

First, attributes may be renamed in classes which have inherited them, as observed in [BANE87b]. For the purposes of queries, a renamed attribute should be regarded as the original attribute. For example, suppose that the attribute Color, defined in the class Vehicle, is renamed as AutoColor in the class Automobile. Although the attribute has two different names, the distinction should be ignored in queries.

Second, in the case of multiple inheritance, a class may inherit an attribute from one of its superclasses [BANE87b]. If the attributes in all superclasses are *identical*, that is, they have the same name and same domain, there is no problem. However, complications arise if the attributes are only *equivalent*, that is, they were all defined in the same class somewhere higher in the class hierarchy, but they have been renamed or their domains have been changed. Suppose that the superclasses Si and Sj of a class C have equivalent attributes Ai and Aj, and that C inherits only the attribute Ai of Si. Now consider a query on Sj whose scope of evaluation is the class hierarchy rooted at Sj, and the attribute Aj is to be output. Then the values of the attribute Ai inherited into the class C (and its subclasses) should also be output, since Ai and Aj are equivalent after all.

If the access scope of a query on a class C is the class hierarchy rooted at C, the result of the query is a heterogeneous set of instances, since the instances belong to a number of different classes with different number of attributes. The application (or the user) which issues the query must obviously be prepared to deal with this hetero-

geneous mix of instances. One approach is for the database system to return only a list of object identifiers (along with the class identifiers) of the instances in the query result, and have the application make explicit requests for all or some of the actual instances. This approach has been implemented in ORION [KIM88a]; and it is in a sense similar to the cursor mechanism in SQL/DS [IBM81] or portals in POSTGRES [STON86]. If the query involves a projection operation, the database system may return the set of object identifiers and the values of the projected attributes of the corresponding objects.

The projection operation entails the familiar problem of eliminating duplicates from the query result. For object-oriented databases, unlike relational databases, two different criteria exist for determining the uniqueness of an instance from a collection of instances. One criterion is the equality of the object identifiers, which may be called *object equality*; while another is the equality of the contents of the instances, which may be called *value equality*, that is, the values of the *user-defined* attributes in the instances. We note that an instance may contain a number of system-defined attributes, such as the version number, update timestamp, and so on, to support various functionality of the system [CHOU88]. The values of system-defined attributes really should not be included in testing value equality of objects, since the users do not explicitly create their values. In relational databases, only value equality is used. For object-oriented databases, both may be useful; however, value equality requires some additional consideration. A class in general does not have the same set of attributes as its superclass, since a class inherits all attributes from its superclass and may have additional attributes defined on it. This means that value equality between classes with different sets of attributes need not be tested.

It is interesting to note that our type s and type s-s queries are recursive queries and that they are safe, that is, the result of such a query is a finite set of objects. This is because a primitive class is never the target of a query.

Throughout our discussion so far, we have assumed that the query result is simply returned to the application (or the user). Now we need to consider the problem of saving the result of a query. As we discussed earlier, an object belongs to a class, and a class has a position somewhere in the class hierarchy. These simple principles impose some difficult constraints on the query model for object-oriented databases. In relational databases, the result of a query is itself a relation, and it may simply be saved as a new relation. In object-oriented databases, the result of a query is a set of objects belonging to a class or a class hierarchy rooted at some class. Let us consider a query on a class C whose scope of access is the class hierarchy rooted at C. The result of the query will in general be a heterogeneous set of instances that form a separate class hierarchy each of whose class is derived from a corresponding class in the original class hierarchy; let us denote the root of the new class subhierarchy by C-new. The instances in the newly created classes must all have different object-identifiers from those in the corresponding instances of the original classes.

A difficult question is where we should place the class C-new in the class hierarchy for the entire database. Intuitively, there are two options to consider. One is to place C-new as a new subclass of the superclasses of the class C; and the other is to treat C-new as an immediate subclass of the class OBJECT, which is the root of the class hierarchy for the entire database. These options are shown in Figure 4, where the class AutoCompany-new is the root of a new class hierarchy which is created to save the result of a query on the class-hierarchy rooted at the class AutoCompany.

Let us consider option-1, where C-new is made a subclass of the superclasses of the class C. If the query involves a projection operation, this solution would result in a subclass with fewer attributes than its superclass, thereby violating the *full-inheritance invariant* in schema evolution as observed in [BANE87b]. In particular, if any attribute which C inherited from its superclasses is dropped from the query result, C-new will have fewer attributes than the superclasses of C; for example, in Figure 4, the class AutoCompany-New may have fewer attributes than the class Company. This suggests that C-new with fewer attributes than C should perhaps be made a superclass (rather than a subclass) of a superclass of C. However, this fix does not work either, if C-new contains any attribute which was defined in C (rather than inherited into C). C-new will then end up with an attribute which its subclass does not have.

There is another problem with option-1, regardless of whether the query involves a projection operation. If C-new is placed on the class hierarchy, it becomes subject to schema evolution on its superclasses. In other words, changes to the definition of any of its new superclasses propagate down to C-new, again in accordance with the full-inheritance invariant. For example, if an attribute is dropped from the class Company, the same attribute must be dropped from the class AutoCompany-New. This defeats the purpose of saving the result of a query.

Let us now consider option-2, where C-new is made an immediate subclass of the system-defined class OBJECT. One drawback of this solution is that all attributes and methods inherited into C-new must now be explicitly replicated in C-new. This requires extra storage; however, it does not appear to be a serious drawback. Option-2 is our choice.

3.2 MULTIPLE-OPERAND QUERIES

3.2.1 Join

A relational database consists of a number of relations. Join is used to correlate n different relations (n > 1) on the basis of the values in attributes common to the relations. It is important to realize that join is a crucial operation in relational databases largely because of the limitations of the relational model of data. In particular, the relational model of data restricts the attribute of a relation to a single primitive value; that is, an attribute may have only one value, and the value must be of a primitive data type (integer, float, string, boolean). These limitations have been removed in object-oriented and nested-relational models of data.

implicit joins in a class-composition hierarchy

Let us consider the three classes Vehicle, Company, and Employee in Figure 1. These classes may be defined as relations as follows.

Vehicle (ID, Color, DriveTrain,
Manufacturer)
Company (CompanyName, Location,

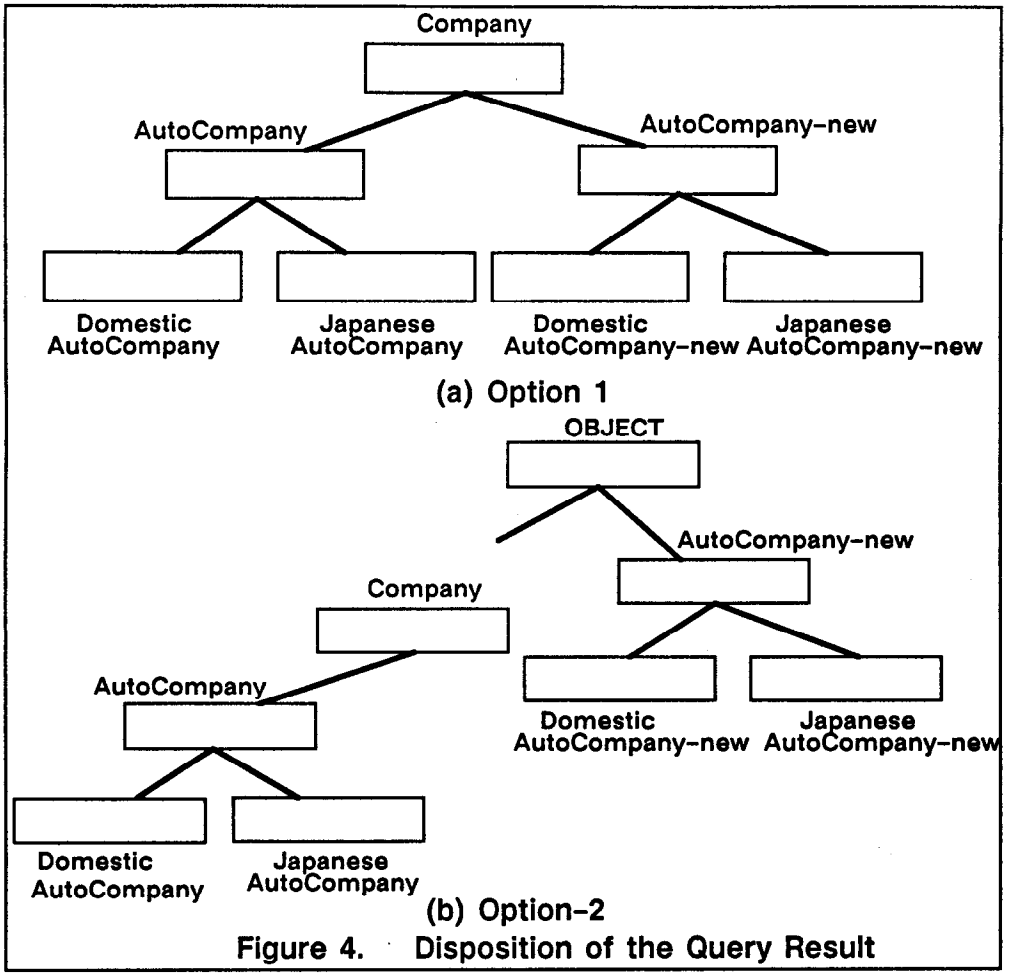

Figure 4. Disposition of the Query Result

President)
Employee (SSN, EmployeeName, Age)

The attributes Manufacturer and CompanyName have the same domain; and
the President and EmployeeName attributes share the same domain.

To find all blue vehicles manufactured by a company located in Detroit and whose president is under 50 years of age (example query Q1), the query must be formulated as a join of the above three relations, and the user must explicitly specify the join attributes. As we have seen already, however, the same query may be formulated in object-oriented databases as a retrieval from a class-composition hierarchy in which joins between the classes are defined in the schema between an attribute of a class and the domain of the attribute.

A single-operand query in object-oriented databases is then one type of join; it is an implicit join of the classes on a class-composition hierarchy rooted at the target class of the query. However, it has a few important limitations relative to the relational join. One is that the output attributes are restricted to those in the target class; that is, it is not possible to output any attribute in any class which is not the root of the query graph for a query. This may be easily remedied. The query syntax needs to be augmented with some means of specifying the output attributes; the output attributes may be specified in a single list, or they may be specified in a separate list associated with each of the classes in the query.

Another, much more serious, limitation of the implicit join in a single-operand query is that the join is restricted to the attribute-domain relationship specified in the schema; that is, the implicit join is essentially the materialization of the values of an attribute. Consider a class Ci with an attribute A whose domain is the class Cj. As observed in [BANE88], the materialization of the instances of Ci is equivalent to a join of Ci and Cj, in which the attribute A of Ci and the object-identifier attribute of Cj are the join attributes. Below, we elaborate on the limitations of the implicit join, and propose solutions to overcome these limitations.

explicit joins of classes

The limitation of a join implied in a single-operand query is essentially that it statically determines the classes to be joined, and the join ordering of the classes. Let us examine this in more detail. The implicit join statically determines the join ordering between a pair of classes Ci and Cj, where Cj is the domain of an attribute of Ci, such that Ci is always the 'outer class' and Cj the 'inner class'

(analogous to the outer and inner relations in a join ordering for a pair of relations in relational databases). This means that it is not possible to formulate a query whose semantics require implicit reversal of an attribute-domain link specified in the schema. For example, the class Company is the domain of the attribute Manufacturer of the class Vehicle in Figure 1. It is possible to find all vehicles manufactured by certain companies. However, it is not possible to find all companies that manufacture certain vehicles; the class Company has no attribute, say, Manufactures, whose domain is the class Vehicle.

In relational databases, the existence of a common attribute in a pair of relations implies, correctly, the attribute-domain relationship in both directions between the relations. It is certainly useful to extend our model of single-operand queries by postulating, for any attribute of a class C whose domain is the class D, an implicit attribute of the class D whose domain is the class C; and by allowing implicit joins to be formulated along the attribute-domain links defined by the implicit attribute. For example, the attribute Manufacturer of the class Vehicle, and the class Company define an implicit attribute of the class Company whose domain is the class Vehicle. Then a single-operand query on the class Company may be formulated in which instances of the class Vehicle are materialized as values of the implicit attribute of the class Vehicle.

However, the implicit join also statically determines the classes which may be joined to those pairs of classes Ci and Cj such that Cj is the domain of an attribute of Ci. This means that it excludes joins between an arbitrary pair of classes with attributes which share a common domain. For example, suppose that, as shown in Figure 5, the class Employee has an additional attribute Address, and that the domains of Address and the attribute Location of the class Company are the same, say a class City. Then a meaningful query may be to find all employees who live in the city in which their companies are located. However, as long as there is no attribute-domain relationship between the attribute Address and the class Company, or between the Location attribute and the class Employee, it is not possible to formulate this query.

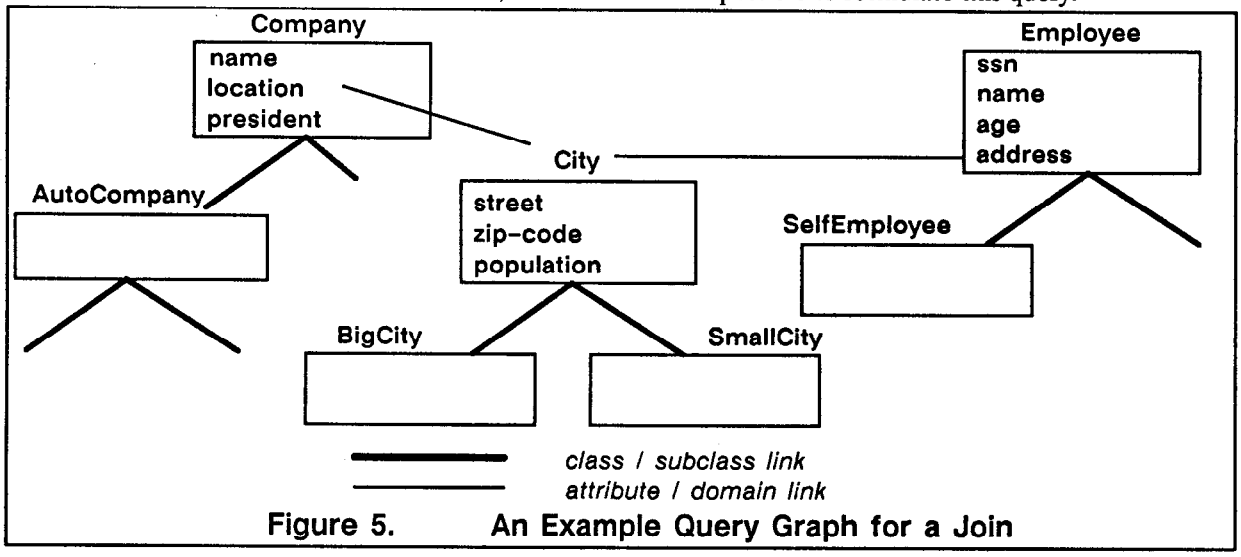

Figure 5. An Example Query Graph for a Join

We now extend our query model to admit joins of classes on user-specified arbitrary join attributes. Again, the extended model must be consistent with the object-oriented data model. To fully define the semantics of joins, we need to introduce the notion of attribute compatibility. Two attributes Ai and Aj are *compatible*, if the domain of Ai is identical to the domain of Aj, or is a superclass or a subclass of the domain of Aj. Unlike relational joins, in which the domains of the join attributes must be identical, we require the join attributes to only be compatible. For example, in Figure 5 the attributes Address and Location are compatible, since the share the same domain. Suppose now that the domain of the Location attribute is the class BigCity, which is a subclass of City. The attributes Address and Location are still compatible, since the domain of Location is a subclass of the domain of Address.

Next, we must account for the class hierarchy in the scope of query evaluation. Each class to be joined is the root of a class hierarchy. The scope of query evaluation, for each class to be joined, may be either instances of the class alone or instances of the class hierarchy rooted at the class. The query syntax must allow the user to specify the scope for each class in a join, just as was the case for a single-operand query.

The result of a join is a set of instances formed by concatenating instances from different classes. This presents a problem for systems like ORION which first return the list of object identifiers and then return specific instances on demand; this problem does not arise in systems which directly return the actual set of concatenated instances. The problem is whether to return a set of n object identifiers (one for each class containing an output attribute from the join) for each concatenated instance or to return a new object identifier for the concatenated instance. If the system simply returns a set of n object identifiers for each concatenated instance, the user must in general send the entire list back to the system to request the retrieval of the actual concatenated instance. However, if the system must return a new object identifier for each concatenated instance, it must incur the overhead of generating the identifiers, and temporarily saving all the concatenated

instances along with their identifiers. If the query result is not to be saved, this is a rather high overhead; therefore we adopt the first approach.

Just as is the case with a single-operand query, if the result of a join needs to be saved as a snapshot, the instances in the result must be given unique identifiers and assigned to a class. The same considerations we had for saving the result of a single-operand query apply here, and the result of a join is to be saved in a new class which is an immediate subclass of the class OBJECT, the root of the class hierarchy for the entire database.

Figure 5 is the query graph for a query involving a join of two classes Company and Employee; the join attributes are Location and Address. The following is a formal definition of the query graph for a join of a pair of classes Ci and Cj; it is straightforward to extend the definition for a join of more than two classes, and we leave it as an exercise to the reader. Let us denote by Si the domain of the join attribute in Ci, and by Sj the domain of the join attribute in Cj; let us assume that Si is a superclass of Sj or Si = Sj. The result of the join is a set of instances formed by concatenating the instances from the scope query evaluation for Ci and Cj which satisfy the join predicate.

1. Ci and Cj each is the root of a query graph corresponding to a single-operand query on Ci and Cj, respectively. That is, each query graph is the root of a class hierarchy and a class-composition hierarchy rooted at the class to be joined.
2. The query graphs for Ci and Cj partially overlap, because of the compatibility in the join attributes. Si and Sj each is the root of a class-composition hierarchy, each of whose node is in turn the root of a class hierarchy. If Si = Sj, the entire class-composition hierarchy is shared by Ci and Cj via their respective join attributes. If Si is a superclass of Sj, the class-composition hierarchy rooted at Si is the domain of the join attribute of Ci; while the domain of the join attribute in Cj is the class-composition hierarchy rooted at Sj.

3.2.2 Set Operations

The set operations for object-oriented databases also require some extensions to the corresponding operations for relational databases. As in relational databases, these operations are useful largely for further manipulating the results of queries, that is, when the operands are queries.

The operand is a set of instances; more specifically, it may be the set of instances of a class defined in the database, or it may be a set of instances obtained as the result of a query. On the surface, the semantics of the set operations for object-oriented databases are identical to those for relational databases. Actually, however, they differ in three interesting ways.

First, for object-oriented databases, the operand, and the result of the operation, may be a heterogeneous set of instances, that is the instances of the operand may not all belong to the same class; in relational databases, the operand is a homogeneous set of tuples.

Second, as we have seen already, for object-oriented databases, two different criteria exist for determining the uniqueness of an instance from a collection of instances, namely, object equality and value equality.

Third, as usual, the scope of evaluation for an operand which is a database class may be instances of the class or the instances of a class hierarchy rooted at the class. The user may specify the scope for each class in the operation.

4. COMPARISON WITH OTHER QUERY MODELS

There are three common, and major, differences between our query model and other query models, including those for hierarchical, relational, nested relational, and object-oriented data models. First, our query model reflects the semantics of the class hierarchy. The concept of a class hierarchy is not a part of a nonobject-oriented data model, and therefore, of a query model derived from it. Unhappily, even the few query models proposed for object-oriented database systems [COPE84, BEEC87, ROWE87, ALAS88] neglect to account for the impacts of the class hierarchy on queries.

Second, the record-type hierarchy in hierarchical databases and the nested relations in nested relational databases are similar to the class-composition hierarchy in an object-oriented data model. However, the record-type hierarchy and nested relations form a directed acyclic graph, unlike our class-composition hierarchy which may include cyclic branches. The query model based on the normalized relational model does not give rise to a nested structure of relations.

Third, other query models use only value equality, that is, equality testing between entities is done on the basis of the contents of the entities, rather than object equality based on the object identifiers. This difference is significant in the definition of a predicate and the definition of the semantics of the set operations.

One additional major difference between our query model and other models proposed for object-oriented databases is the treatment of a snapshot, that is, the query result which is saved and becomes a part of the persistent database. Our model solves an important problem posed in [KIM88b].

SUMMARY

In this paper, we provided what we believe now to be a comprehensive model of queries for object-oriented databases. The work represents a significant formalization and extensiosn of the query model first proposed in [BANE88]. The model proposed in [BANE88] and elaborated somewhat in [KIMK89] is based on the view that a query model may be defined as a subschema of the database schema; the database schema is reduced to a query model by applying the selection and projection operations. To our knowledge, it is the first query model which made serious efforts to capture the semantics of object-oriented concepts. However, the model defined only limited type of a single-operand query, that is, a query whose target is a single class or a class hierarchy rooted at that class. Further, the model contained some important oversights, notably in its treatment of the projection operation, and the directionality of the arcs in the class-composition hierarchy.

In this paper, we first provided a considerably more rigorous treatment of the single-operand query, and corrected the mistakes in the model given in [BANE88]. Next, we significantly extended the model of [BANE88]

to provide a formal basis for a query which involves more than one operand, namely, object-oriented equivalents of the relational join and set operations. Then, we briefly summarized the essential differences between our query model, and the query models for traditional data models and the models proposed by other researchers for object-oriented databases.

ACKNOWLEDGEMENTS

This paper is a compendium of the ideas about queries in object-oriented databases during the past three years. Past and present, both permanent and temporary, members of the ORION project have provided some of the ideas which have helped me to shape my conclusions about queries for object-oriented databases. I owe special thanks to Jay Banerjee (now with Unisys) and Fausto Rabitti (with C.N.R., Italy) for their ideas and discussions. In addition, Elisa Bertino (with C.N.R., Italy) and Fred Mellender (Eastman Kodak), and the anonymous referees, gave me a number of insightful and helpful comments on an initial draft of this paper.

REFERENCES

[ABIT84] Abiteboul, S., and N. Bidoit. "Non First Normal Form Relations to Represent Hierarchically Organized Data," *Proc. ACM Symp. on Principles of Database Systems*, Waterloo, Canada,1984.

[ABIT86] Abiteboul, S., and N. Bidoit. "Non First Normal Form Relations: An Algebra Allowing Data Restructuring," *Journal of Computer and System Sciences*, no. 33, pp 361-393, 1986.

[ALAS88] Alashqur, A., et al. "OQL - An Object-Oriented Query Language," technical report, Database Systems R/D Center, University of Florida, Gainesville, Florida, 1988.

[ANDR87] Andrews, T., and C. Harris, "Combining Language and Database Advances in an Object-Oriented Developmant Environment," *Proc. OOPSLA 87 Conference*, Orlando, Florida, 1987.

[BANE87a] Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems*, January 1987.

[BANE87b] Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1987.

[BANE88] Banerjee, J., W. Kim, and K.C. Kim. "Queries in Object-Oriented Databases," in *Proc. 4th Intl. Conf. on Data Engineering*, Los Angeles, Calif. Feb. 1988.

[BEEC87] Beech, D. "OSQL: A Language for Migrating from SQL to Object Databases," working paper, Hewlett-Packard Lab., Palo Alto, Calif., 1987.

[COPE84] Copeland, G., and D. Maier. "Making Smalltalk a Database System," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, June 1984, pp. 316-325.

[DESH88] Deshpande, A., and D. Van Gucht. "An Implementation for Nested Relational Databases," in *Proc. Intl. Conf. on Very Large Data Bases*, 1988.

[FISH87] Fishman, D., et al. "IRIS: an Object-Oriented Database Management System," *ACM Trans. on Office Information Systems*, vol. 5. no. 1, Jan. 1987, pp. 48-69.

[IBM81] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, Jan. 1981.

[IEEE88] IEEE Computer Society, *Database Engineering*, special issue on Non-First Normal Form Relational Databases (ed. Z.M. Ozsoyoglu), Sept. 1988.

[JAES82] Jaeschke, G., and H. Schek. "Remarks on the Algebra of Non First Normal Form Relations," *Proc. ACM Symp. on Principles of Database Systems*, Los Angeles, CA,1982.

[KHOS86] Khoshafian, S., and G. Copeland. "Object Identity," in *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, Oct. 1986.

[KIM88a] Kim, W., et al. "Integrating an Object-Oriented Programming System with a Database System," in *Proc. 2nd Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, San Diego, Calif., Sept. 1988.

[KIM88b] Kim, W. "Object-Oriented Databases: Definition and Research Directions" submitted for publication, Sept. 1988.

[KIM89] Kim, W., et al. "Features of the ORION Object-Oriented Database System," *Object-Oriented Concepts, Applications, and Databases*, (ed. W. Kim, and F. Lochovsky), Addison-Wesley, 1989.

[KIMK89] Kim, K.C., W. Kim, and A. Dale. "Cyclic Query Processing in Object-Oriented Databases," in *Proc. 5th Intl. Conf. on Data Engineering*, Los Angeles, Calif., Feb. 1989.

[MAIE86] Maier, D., et al. "Development of an Object-Oriented DBMS," in *Proc. 1st Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, Oct. 1986.

[MAKI77] Makinouchi, A. "A Consideration of Normal Form of Not-necessarily Normalized Relations in the Relational Data Model," in *Proc. Intl Conf. on Very Large Data Bases*, 1977, pp. 447-453.

[ROWE87] Rowe, L., and M. Stonebraker. "The POSTGRES Data Model," in *Proc. Intl. Conf. on Very Large Data Bases*, Brighton, England, Sept. 1987, pp. 83-95.

[STEF86] Stefik, M., and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40-62.

[STON76] Stonebraker, M., E. Wong, P. Kreps, G. Held. "The Design and Implementation of INGRES," *ACM Trans. on Database Systems*, vol. 1, no. 3, Sept. 1976, pp. 189-222.

[STON86] Stonebraker, M., and L. Rowe. "The Design of POSTGRES," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, May 1986.

[ZANI83] Zaniolo, C. "The Database Language GEM," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, May 1983, pp. 207-218.