Information Systems Vol. 18, No. 7, pp. 461–487, 1993
Printed in Great Britain.

0306-4379/93 $6.00 + 0.00

USING SQL WITH OBJECT-ORIENTED DATABASES†

JAN VAN DEN BUSSCHE[1] and ANDREAS HEUER[2]
[1]University of Antwerp (UIA), Department of Mathematics and Computer Science, Universiteitsplein 1, B-2610 Antwerp, Belgium
[2]Technical University of Clausthal, Institute of Computer Science, Erzstrasse 1, D-3392 Clausthal-Zellerfeld, Germany

(Received in final revised form 4 September 1992)

Abstract—We investigate how the standard database query language SQL can be extended to deal with the newly emerging trends of complex objects and object orientation. Our main concern is to extend SQL as naturally as possible, rather than to redesign SQL into "yet another" object-oriented query language. We achieve this goal through a faithful mapping from a complete object-oriented database model, compatible with recent proposals in the field, to the nested relational database model, which is widely accepted as a natural extension of the relational database model on which standard SQL is based. We provide formal definitions of syntax and semantics. We also review related research and situate our work into it.

Key words: Object-oriented database models, query languages, SQL

1. INTRODUCTION

The current trend toward more complex data-intensive applications has triggered an important amount of research in the introduction of object-oriented concepts such as *complex objects*, *object identity*, *inheritance* and *methods* into database systems [2, 3]. Recent proposals in this field (among others [4–6]) seem to converge towards a model supporting all the above notions, while still having the standard relational model as a basic subset. This last feature is important. Object-oriented database models that naturally extend the relational model retain more easily the highly desirable properties that relational database systems provide [7], such as a well-understood and formally defined technology, and a powerful *ad-hoc* query facility. In particular this last feature was not apparent in object-oriented databases until recently, although the central role of a complete, safe, high-level and efficiently evaluable query language in database systems is generally recognized.

In this paper we focus on this need, by defining and examining an object-oriented query language that is a natural extension of SQL, the standard query language for relational databases [7–9].

The data model on which our language is based unifies a number of influential recent proposals for object-oriented models [4, 5, 10]. As such, it supports typed complex objects, object identity, typed classes consisting of object identifiers with associated state and an inheritance hierarchy. Furthermore, it provides classical relationships: this renders the query language more flexible and enables the support of methods in an elegant way [4, 5].

The classes, methods and relationships of our data model can be faithfully mapped into *nested relations* [11] with special semantics for object identity. The one-to-one correspondence thus obtained enables us to build a query language that naturally extends SQL, since nested relations are widely accepted as a natural complex object extension of the relational database model, on which standard SQL is based.

In designing our language, care was taken to obtain a language that is *closed* and *adequate* with respect to its underlying object-oriented data model. With closure we mean that the result of a query can be represented in the data model itself, i.e. as a derived relationship or as a derived class, properly linked into the inheritance hierarchy. Closure is important in the context of query composition and view definition. With adequacy we mean that our language provides techniques to handle all data model constructs present in the data model. In particular, we provide logical

†A preliminary version of this paper was presented at the *Second Workshop on Foundations of Models and Languages for Data and Objects* [1].

predicates and quantifiers adequate for "browsing" in complex structures, restructuring techniques and an inheritance mechanism.

We would like to make it clear that, rather than proposing "yet another" query language, our main concern was to stick as closely as possible to the standard query language SQL, both in syntax as semantics, and from this perspective extend it naturally in order to incorporate object-oriented concepts. Motivations for such an approach are legio (cf. also Ref. [12]): besides the for better or worse establishment of SQL as "intergalactic dataspeak" [13], we can more easily exploit standard relational or nested relational database technology with regard to optimization, formal semantics and correctness proving, and implementation. Moreover, we preserve the applicability of earlier work on extending SQL in other directions, such as adding recursion [14] or dealing with historic information [15]. On the other hand, we are of course certainly not implying that non-SQL-like languages are uninteresting; rather, our work is an investigation of one possible direction to go.

This paper is organized as follows: in Section 2, we introduce our object-oriented database model, and give the mapping of the model to the nested relational database model. We also present a non-traditional database application, the Geometry database, which will be used as a running example. In Section 3, we give an informal overview of our query language, stressing the points where SQL is extended to deal with object-oriented concepts. In Section 4, we formally define syntax and semantics, utilizing the well-known nested relational algebra. Our semantics is specified in an operational manner, in order to facilitate a more direct implementation. This elaboration is, to our knowledge, the most thorough one reported in the literature to date. We give conclusions in Section 5, where we also situate our work within related research, in particular on other SQL-like languages and on views.

We assume from the reader familiarity with the relational and the nested relational database model [11], as well as with common object-oriented concepts and terminology [3].

2. OBJECT-ORIENTED DATABASE MODEL

In this section, we introduce the model for object-oriented databases which we will use throughout the paper. Furthermore, we present a mapping from this model to the nested relational database model, which will be central to our further development. We illustrate the introduced concepts on a running example, the *Geometry Database*.

Attribute names, objects and values—Before being able to start with the definition of our data model, we need to make a few basic assumptions. We assume the existence of two sufficiently large, disjoint sets: the $\mathcal{O}$ of *objects*, and the set U of *attribute names*. Additionally, we have a set $\mathcal{D}$ of *basic data types*, which are assumed to be system-defined, such as Integer, String or Bitmap. Elements of these data types are called *basic values*.

Schemes: class and relationship names, class hierarchies, and types:

Definition 2.1. A *database scheme* has the form $(\mathcal{C} \leqslant \mathcal{R}, \tau)$, where:

1. $\mathcal{C}$ is a set whose elements are called *class names*;
2. $\leqslant$ induces a lattice on $\mathcal{C}$ (the *class hierarchy*);
3. $\mathcal{R}$ is a set whose elements are called *relationship names*;
4. $\tau : \mathcal{C} \cup \mathcal{R} \to \mathcal{T}$ assigns a *type* to each class and relationship name. Here, the set $\mathcal{T}$ of types is defined by:
   - $\mathcal{D} \subset \mathcal{T}$ (*basic types*);
   - $\mathcal{C} \subset \mathcal{T}$ (*abstract types*);
   - if $\{A_1, \ldots, A_n\} \subset U$ and $T_1, \ldots, T_n$ are non-tuple types in $\mathcal{T}$ such that no A_i occurs in T_j, then $[A_1 : T_1, \ldots, A_n : T_n] \in \mathcal{T}$ (*tuple types*);
   - if T is a tuple type then $\{T\} \in \mathcal{T}$ (*set types*).

The result of τ must be a tuple type. Furthermore we require that no attribute occurs in the types of two different class names that are connected in the class hierarchy.

The last requirement will turn out to be useful later on for preventing conflicts in the (multiple) inheritance of attributes (Definition 2.3), much in the same way as is done in Ref. [16].

Basic and abstract types will be sometimes referred to as *atomic types*, while tuple and set types will be sometimes referred to as *complex types*. It will be convenient later to extend the class hierarchy, defined above on abstract types alone, to all atomic types in the trivial manner, by putting $D \leqslant D$ for each $D \in \mathscr{D}$. The abbreviation $A_1, \ldots, A_n : T$ instead of $A_1 : T, \ldots, A_n : T$ will be used frequently in the sequel.

Instances: objects and their states. We can now define the extent of a type, and afterwards the *database instance*, consisting of the actual contents of the database: sets of objects and their states and relationships. Relationships will provide a natural mechanism to introduce methods, and a flexible mechanism for representing query results.

Definition 2.2. Consider the set of class names of some database scheme. We can associate actual *classes*, i.e. sets of objects, to the class names, in a manner which respects the class hierarchy of the database scheme, using a *class assignment*. A class assignment is a mapping *class* which assigns to each class name C a subset $class(C)$ of $\mathscr{O}$, in such a way that $C \leqslant C' \Leftrightarrow class(C) \subseteq class(C')$. Given such a class assignment, the *extent* $\|T\|$ of a type T is then defined as follows:

- For $D \in \mathscr{D}$, $\|D\|$ is the (system-supplied) set of basic values of type D;
- For $C \in \mathscr{C}$, $\|C\| := class(C)$;
- If T is of the form $[A_1 : T_1, \ldots, A_n : T_n]$,

$$\|T\| := \{[A_1 : v_1, \ldots, A_n : v_n] \mid v_i \in \|T_i\|\};$$

- If T is of the form $\{T'\}$, $\|T\| := \mathscr{P}(\|T'\|)$.†

For a tuple type T, elements of $\|T\|$ are called *tuples*.

As the elements of basic type extents were called basic values, the elements of tuple and set type extents are called *complex values* [5].

We remark that, to allow for an easy usage of objects and values, we could add a *name space* to assign unique names to certain objects and values [5, 17]. For example, in Section 3, we will use *c_45* as the name of a certain fixed object of class *Rotation_point*, and *empty* will be the name for the constant { }, the empty set. We will not formally incorporate a name space in our model, but it is straightforward to do so.

We are finally ready to define:

Definition 2.3. A *database instance* over a database scheme $(\mathscr{C}, \leqslant, \mathscr{R}, \tau)$ has the form $(class, st, rel)$, where:

1. *class* is a class assignment;
2. *st* is a mapping, assigning to each pair (o, C), where $C \in \mathscr{C}$ and $o \in class(C)$, a value in $\|\tau(C)\|$, called the *local state of o at C*, and denoted by $st_C(o)$.
   Knowing the local states of o, then $\overline{st}_C(o)$, the *inherited state of o at C* is defined as the concatenation of all local states in $\{st_{C'}(o) \mid C \leqslant C'\}$.
3. *rel* is a mapping, assigning to each relationship name $R \in \mathscr{R}$ a relationship $rel(R) \subseteq \|\tau(R)\|$.

We point out that an object indeed may have several local states, since due to the class hierarchy, it can belong to different classes. Concatenation of these local states gives then the usual semantics for multiple inheritance. It follows from Definition 2.1 that inherited state is a well-defined notion, i.e. for any object its inherited state is in the interpretation of some tuple type.

Also, for each $R \in \mathscr{R}$, $\tau(R)$ is a tuple type, so $rel(R)$, being a set of tuples, is indeed a relationship.

Example 2.4. Let us illustrate the notions introduced above on a good-sized example, which we will use throughout this paper.

In a *Geometry database*, we consider points, vectors and polygons as possible abstract objects, i.e. elements of $\mathscr{O}$. The objects have attributes, the names of which are enumerated in the set U, the universe of attribute names in the system:

$$U = \{X, Y, p1, p2, list, elt, \#, v1, v2, angle, receiver, result, arg1, \ldots\}.$$

X and Y are the attribute names for the coordinates of point objects.

†For a set S, $\mathscr{P}(S)$ denotes the powerset of S.

These coordinates take real numbers as values, so we have a basic data type *Real* in $\mathscr{D}$, among others:

$$\mathscr{D} = \{Real, Integer, String, \ldots\}.$$

The attribute values can be atomic or complex values like integers in the case of the coordinates, or objects as in the case of the starting and ending points of vectors.

Let us now look in more detail to the scheme of the Geometry Database. It consists of four components: the set of class names $\mathscr{C}$, the class hierarchy $\leqslant$, the set of relationship names $\mathscr{R}$ and the type assignment τ.

The objects are classified in classes, the names of which are enumerated in the set $\mathscr{C}$:

$$\mathscr{C} = \{Point,\ Vector,\ Polygon,\ Intersection,\ Rotation\_point\}.$$

Since objects of class *Intersection* are special *Point* objects, and the same holds for objects of class *Rotation_point*, the class hierarchy is defined by:

$$Intersection \leqslant Point \quad and \quad Rotation\_point \leqslant Point.$$

The type assignment τ is defined as follows. Regular points have two attributes, namely their two coordinates:

$$\tau(Point) = [X, Y : Real].$$

Vectors consist of a starting and an ending point:

$$\tau(Vector) = [p1, p2 : Point].$$

Intersections are special points, determined by the intersection of two vectors:

$$\tau(Intersection) = [v1, v2 : Vector].$$

Intersections also inherit X and Y attributes, due to $Intersection \leqslant Point$. Another special class of points are the rotation points; they serve as centers of rotation operations, and therefore have an additional angle attribute:

$$\tau(Rotation\_point) = [angle : Real].$$

Finally, we have the more complex polygon objects, which consist of a list of points. The list is modeled by a set of pairs, the second component of which is an actual point, and the first component of which is the point's sequence number in the list. Thus:

$$\tau(Polygon) = [list : \{[\# : Interger, elt : Point]\}].$$

Different *methods* work on the objects. In fact, in "pure" object-oriented (e.g. Smalltalk-based) systems, attributes are a special case of methods, i.e. unary methods where the implementation is simply an access to the attribute value stored in the database. Similarly, *derived* or *computed* attributes found in semantic databases [18], are (unary) methods whose implementation involves a computation. In our model, both kinds of these unary methods are modeled as attributes. We also support more general n-ary methods ($n > 1$), i.e. methods having $n - 1$ additional parameters, which can be seen as functions with n inputs and 1 output. Since from a mathematical viewpoint, such functions are nothing else than special $n + 1$-ary relationships, we represent methods as relationships. This is one major reason for the inclusion of relationships in our data model.

Thus, suppose we have a method *distance* defined on points, which returns the distance between the receiver point and an argument point, denoted as:

$$Point \cdot distance(Point) : Real.$$

To represent this method, we have a relationship with name *distance*, included in the set $\mathscr{R}$, which contains also all other method/relationship names:

$$\mathscr{R} = \{distance, rotate, mirror, \ldots\}.$$

We have:

$$\tau(distance) = [receiver, arg : Point, result : Real].$$

The method *rotate*, defined on polygons, rotates the receiver around the argument, a rotation point. We have:

$$\tau(rotate) = [receiver : Polygon, arg : Rotation\_point, list : \{[\# : Integer, elt : Point]\}].$$

Remark that *rotate*, being a *state-changing* method, returns a new polygon *value*, which is the new state after rotating (i.e. a new *list*-attribute). We also have the state-changing method *mirror*, defined on points, which mirror the receiver against the argument, a vector. So, *mirror* returns a new point state, i.e. a tuple value of type, say, $[mX, mY : Real]$. We have:

$$\tau(mirror) = [receiver : Point, arg : Vector, mX, mY : Real].$$

So far, we only considered methods defined on classes, i.e. whose receiver is an abstract object. This is the standard situation in most object-oriented systems. However, there is no good reason to exclude methods working directly on values (for a discussion see e.g. [19]). As a simple example, computing the maximum of a set of reals can be seen as a method *max*, defined on sets of reals, and returning their maximum:

$$\{[elt : Real]\}.max : Real.$$

We have:

$$\tau(max) = [receiver : \{[elt : Real]\}, result : Real].$$

An overview of the scheme of the Geometry Database is given in Fig. 1. Let us next concentrate on how an instance of the Geometry Database might look like. Formally, it consists of three components: a class assignment *class*, the state mapping *st* and the relationship assignment *rel*.

The class assignment *class* is rather straightforward. It populates the different classes with abstract objects:

$$class(Point) = \{\alpha, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \ldots\};$$

$$class(Vector) = \{\gamma_1, \gamma_2, \ldots\};$$

$$class(Intersection) = \{\alpha_4, \ldots\};$$

$$class(Polygon) = \{\delta_1, \delta_2, \ldots\};$$

$$class(Rotation\_point) = \{\alpha_2, \ldots\}.$$

The state mapping *st* assigns a state to each object–class pair. So is α the object representing the center point (0, 0):

$$st_{\text{Point}}(\alpha) = [X : 0, Y : 0].$$

Four other points are represented as follows:

$$st_{\text{Point}}(\alpha_1) = [X : 3.8, Y : 1.9];$$

$$st_{\text{Point}}(\alpha_2) = [X : 3.5, Y : 1];$$

$$st_{\text{Point}}(\alpha_3) = [X : 2, Y : 3];$$

$$st_{\text{Point}}(\alpha_4) = [X : 3, Y : 1.5].$$

The *Vector* objects γ_1 and γ_2 represent the vectors $\alpha \rightarrow \alpha_1$ and $\alpha_2 \rightarrow \alpha_3$, respectively:

$$st_{\text{Vector}}(\gamma_1) = [p1 : \alpha, p2 : \alpha_1];$$

$$st_{\text{Vector}}(\gamma_2) = [p1 : \alpha_2, p2 : \alpha_3].$$

The *Point* object α_4 is also an *Intersection* object, representing the intersection of the vectors represented by γ_1 and γ_2:

$$st_{\text{Intersection}}(\alpha_4) = [v1 : \gamma_1, v2 : \gamma_2].$$

Class *Point*
$\tau(Point) = [X, Y : Real]$

Class *Vector*
$\tau(Vector) = [p1, p2 : Point]$

Class *Polygon*
$\tau(Polygon) = [list : \{[\# : Integer, elt : Point]\}]$

Class *Intersection* **isa** *Point*
$\tau(Intersection) = [v1, v2 : Vector]$

Class *Rotation_point* **isa** *Point*
$\tau(Rotation\_point) = [angle : Real]$

Method *Point.distance*(*Point*): *Real*
Relationship *distance*
$\tau(distance) = [receiver, arg : Point, result : Real]$

Method *Polygon.rotate*(*Rotation_point*) : $[list : \{[\# : Integer, elt : Point]\}]$
Relationship *rotate*
$\tau(rotate) = [receiver : Polygon,$
$arg : Rotation\_point,$
$list : \{[\# : Integer, elt : Point]\}]$

Method *Point.mirror* (*Vector*): $[mX, mY : Real]$
Relationship *mirror*
$\tau(mirror) = [receiver : Point, arg : Vector,$
$mX, mY : Real]$

Fig. 1. Geometry database scheme.

The two coordinates of α_4 are inherited: $st_{\text{Point}}(\alpha_4)$ was already given above. Thus, we have for the inherited state of α_4 at *Intersection*:†

$$\overline{st}_{\text{Intersection}}(\alpha_4) := st_{\text{Point}}(\alpha_4) \cdot st_{\text{Intersection}}(\alpha_4)$$
$$= [X : 3, Y : 1.5, v1 : \gamma_1, v2 : \gamma_2].$$

Of course, the inherited state of α_4 at *Point* equals the local state at *Point*. More details on object states can be found in Fig. 2, where the Geometry Database is shown under the form of (nested) relations, in a way which will be explained momentarily.

The relationship assignment function *rel* assigns actual relations to the relationship names in $\mathscr{R}$. Initially, these are mainly methods, as was explained above. In later stages, there may be also other relationships, holding query results (*views*). So, *rel*(*distance*) represents the semantics of the *distance* method:

$$rel(distance) = \{[receiver : \alpha, arg : \alpha_1, result : 4.2], \ldots\}.$$

From this we can see that the distance method applied to α with α_1 as argument returns the number 4.2. Similar relationships are assigned to the other methods in the system. Obviously, this relational representation of methods is important only from a conceptual, formal semantics viewpoint; an actual implementation will compute the distances between points, not store them.

†The dot '$\cdot$' stands for concatenation of tuples.

Point	*X*	*Y*
α	0	0
α_1	3.8	1.9
α_2	3.5	1
α_3	2	3
α_4	3	1.5
α_5	3	6
⋮	⋮	⋮

Intersection	*X*	*Y*	*v1*	*v2*
α_4	3	1.5	γ_1	γ_2
⋮	⋮	⋮	⋮	⋮

Vector	*p1*	*p2*
γ_1	α	α_1
γ_2	α_2	α_3
⋮	⋮	⋮

Polygon	*list*	
	#	*elt*
δ_1	1	α_1
	2	α
	3	α_4
	4	α_1
δ_2	1	α
	2	α_5
	3	α_1
⋮	⋮	⋮

Rotation_point	*X*	*Y*	*angle*
α_2	3.5	1	45
⋮	⋮	⋮	⋮

Fig. 2. Geometry database instance.

Mapping the object-oriented database to nested relations. The data model developed here can be seen as a slight extension of the *nested relational database model.* The extension lies in the fact that besides the usual basic values, also objects can appear in nested relations. Here, we make this formal by introducing a mapping from our object-oriented data model to nested relations. It can be shown [20] that this mapping is *information-preserving* in the sense of Ref. [21].

Concerning relationships, for each relationship name R, $rel(R)$ is really a nested relation of tuple type $\tau(R)$. So in this case, the mapping is simply the identity.

Classes and objects are mapped into nested relations as follows. For a class name C, we have that for each $o \in class(C)$, $\overline{st}_C(o)$ is a nested relational tuple of the type obtained by concatenating all types in $\{\tau(C')/C \leqslant C'\}$. Call this type $\bar{\tau}(C)$. Now we assume an *identifying attribute* $C \in U$, that does not appear in the type of any class or relationship of the database scheme. Denote by $\hat{\tau}(C)$ the type obtained by adding the attribute $C : C$ to $\bar{\tau}(C)$. Then we associate to class C the nested relation $\widehat{rel}(C)$, of tuple type $\hat{\tau}(C)$. For each object $o \in class(C)$, $\widehat{rel}(C)$ contains exactly one tuple t, defined by $t(C) := o$ and t restricted to $\bar{\tau}(C)$ equals $\overline{st}_C(o)$.

It will become clear later that the thus obtained representation of object-oriented databases as nested relations is essential to having a formal semantics and evaluation mechanism for the object-oriented SQL language proposed in this paper. We also point out that inheritance is completely explicit in the nested relational representation. Indeed, for each object of a class, the *inherited*, not the local, state is stored in the nested relation representing the class.

Example 2.5. Figure 2 shows part of an example instance of the Geometry Database, presenting nested relations for the classes *Point*, *Intersection*, *Vector*, *Polygon* and *Rotation_point*.

3. QUERY LANGUAGE: INFORMAL DESCRIPTION

In this section we informally introduce our object-oriented database query language. We show that our language naturally extends the standard relational database query language SQL, to deal with complex objects and object orientation. The examples presented here are all based on the Geometry Database described in Section 2. The language's exact syntax and complete semantics are formally specified in Section 4.

3.1. Basic select-project-join queries

Since our language is a natural extension of standard SQL, typical relational select–project–join queries are expressed using standard SQL select-from-where expressions.

For example, to retrieve for each vector the X-coordinate of its first point together with the Y-coordinate of its second point, we use:

select *first.X*, *second.Y*
from *Vector*, *Point first*, *Point second*
where *p1* = *first.Point* and *p2* = *second.Point*

If we regard the attributes *p1* and *p2* of *Vector* as unary methods defined on vectors to points (as discussed in Section 2), and treat X and Y similarly, we can rewrite the above query in a more object-oriented flavor as:

select *Vector.p1.X*, *Vector.p2.Y*,
from *Vector*.

In general, the result of a query is a relationship. So is the first query's result a relationship of type [*first.X* : *Real*,*second.Y* : *Real*].

Besides select-from-where expressions, our language supports all other standard SQL constructs. In particular, we provide all three of the binary set operations union, difference and intersection, which can be "orthogonally" applied to any two so-called *compatible* expressions in our language (cf. Section 4). We also point out already now that subqueries in where-clauses will turn out to be particular constructs of our uniform treatment of *complex objects*; this will be illustrated later on in this section.

3.2. Applying methods

The previous example already mentioned the application of unary methods in queries. Consider the more general method *Polygon.rotate*(*Rotation_point*) : [*list* : {[# : *Integer*, *elt* : *Point*]}]. Suppose there is an object of class *Rotation_point* which we have given the name *c_45*, with local states:

$$st_{\text{Point}}(c\_45) = [X : 0, Y : 0] \quad \text{and} \quad st_{\text{Rotation\_point}}(c\_45) = [angle : 45].$$

So, *c_45* represents the rotation of 45° around the center point (0, 0). The following query defines the view obtained by rotating each polygon according to *c_45*:

select *Polygon*, *Polygon.rotate*(*c_45*).*list* into *Polygon*, *rot_list*
from *Polygon*

Without the into-clause, the query result would be of type:

[*Polygon* : *Polygon*, *Polygon.rotate*(*c_45*).*list* : {[# : *Interger*, *elt* : *Point*]}];

the into-clause renames this to

[*Polygon* : *Polygon*, *rot_list* : {[# : *Integer*, *elt* : *Point*]}].

Notice that the above query can effectively be used as the definition of a *class view*. Indeed, its result consists of a set of objects together with their state. See also Section 5.

3.3. Inheritance

Recall that, in the from-clause of a query, a class stands for its corresponding nested relation,

which contains the actual objects together with their inherited state, as described in Section 2. In this way, inheritance is naturally handled in our language.

Consider for example the class *Intersection*, with local state type $[v1, v2 : Vector]$ which inherits the attributes $[X, Y : Real]$ from class *Point*. Recall also the method $Point.mirror(Vector) : [mX, mY : Real]$, defined on points, which returns the new point state obtained by mirroring a point over an argument vector. Suppose one wants to check the intersections for integrity; this can be done by retrieving the "corrupt" intersections, i.e. those that do *not* lie on their respective vectors. Using the fact that a point lies on a vector if mirroring it over the vector leaves it unchanged, we can use the query:

```
select Intersection
from Intersection
where
  Intersection.mirror(v1).mX = X and
  Intersection.mirror(v1).mY = Y and
  Intersection.mirror(v2).mX = X and
  Intersection.mirror(v2).mY = Y
```

The above provides a two-fold illustration on the use of inheritance. First, since intersections are special points, we can apply the method *mirror*, originally defined on points, to intersections. Second, the inherited attributes X and Y can be readily used in the where-clause.

3.4. Complex attributes

The next query returns those pairs of polygons whose sets of point elements are disjoint:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where ((select first.elt from first.list)
  intersection (select second.elt from second.list)) = empty
```

The above expression has a clear SQL-style syntax. The where-subquery is of the form "$(Q) = empty$", and could also have been written in the standard SQL form "not exists (Q)", *empty* is a name for the constant $\emptyset$, the empty set. The only fundamental SQL-extension is in the inner from-clauses, where complex attributes (like *list* of polygons) can be sepecified as fully-fledged relations. This construct extends the *synchronized subquery* mechanism of standard SQL in a natural way in the context of complex objects. Observe also that we allow the use of intersection within subqueries; actually, we allow *any* valid query expression as a subquery, thus rendering our language more orthogonal.

An alternative way to specify the same query in our language is:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where not exists
  (select first.elt from first.list
  where first.elt = any second.elt)
```

This illustrates how the standard SQL quantifier "any" is given extended meaning to deal with complex structures: "any *second.elt*" refers to any *elt* point appearing in the complex attribute *second.list*.

Selections can also take place *directly* inside complex attributes, without using subqueries. For example, the following query returns pairs of lists, where the first list is some polygon's list and the second list is obtained from another polygon's list by selecting only those elements that also appear in the first list:

```
select first.list, second.list
from Polygon first, Polygon second
where select.elt = any first.elt
```

The ability to specify such queries in the above way is to our knowledge a unique feature of our proposal.

3.5. *Subqueries in from-clauses*

There has been a lot of critique concerning the lack of orthogonality in the design of standard SQL (see e.g. [22]). For example, the new version of SQL, SQL2, will support subqueries also in from-clause. We follow this point in the design of our language. For example, consider the query:

```
select first.Polygon, second.Polygon
from
  (select *
  from Polygon first, Polygon second
  where second.elt = any first.elt)
where first.list = second.list
```

The subquery is essentially the previous example query; hence, it can be easily verified that the above query is in fact equivalent to:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where second.list contains first.list
```

This also shows how the contains-predicate of Ref. [8], there originally defined to work on subqueries, is naturally extended to work also on set-valued attributes.

3.6. *Restructuring through select: projection and unnesting*

In algebraic query languages for complex object databases, the *unnest* operation is crucial for "reaching" attribute values that are nested within the complex structure. For example, the scheme of class *Polygon* is $[Polygon, list : \{[\#, elt]\}]$; unnesting the complex attribute *list* brings # and *elt* on the top level, resulting in the (flat) scheme $[Polygon, \#, elt]$. We call the latter scheme a *subscheme* of the former one. It seems natural to extend the notion of subscheme to projections: for example, disregarding sequence numbers in polygons lists results in the subscheme $[Polygon, list : \{[elt]\}]$. A formal definition of the notion of subscheme is in Section 4.1. Exploiting the close connection between unnestings and projections through subschemes, our language treats these two operations uniformly, in the select-clause. An arbitrary sequence of unnestings and projections can be specified in one select-clause by simply giving the associated subscheme. This naturally extends standard, flat SQL: there, select-clauses express projections, while the subschemes of a flat relational scheme exactly corresponds to all projections of that scheme.

As a simple example, the following query lists all sequence numbers appearing in the list of some polygon in the database. First, *list* is unnested, then, by projection, only the attribute # is retained:

```
select # from Polygon
```

Observe that for notational simplicity, the square brackets are omitted in the syntax of subschemes.

We next introduce a useful shorthand, allowing project–unnest restructuring to be performed directly on complex attributes too. As an example, the query retrieving those pairs of polygons that consist of the same set of points:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where (select first.elt from first.list)
  =(select second.elt from second.list)
```

can be rewritten as:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where first.list : {elt} = second.list : {elt}
```

It is also apparent from both cases that we can compare subqueries as well as set-valued attributes for equality in the same natural SQL-style manner.

As a second example, recall the query mentioned previously in Subsection 3.4, retrieving all pairs of "disjoint" polygons:

```
select first.Polygon, second.Polygon
from Polygon first, Polygon second
where not exists
  (select first.elt from first.list
  where first.elt = any second.elt)
```

The comparison

$$first.elt = \text{any } second.elt$$

can be rewritten as

$$first.elt \text{ in } second.list : \{elt\}$$

which is in line with standard SQL comparisons of the form:

$$A_1, \ldots, A_n \quad \text{in} \quad (Q)$$

where Q is a subquery. More generally, a strong point of our language is that subquery results and complex attributes are both treated uniformly, as complex structures.

3.7. Restructuring through group by: nesting

Besides restructuring operations based on projection and unnesting, as in the previous subsection, complex object languages provide a second restructuring mechanism, *nesting*, which groups certain components of tuples that agree on the remaining components together in complex attributes, in much the same way as the group by construct of SQL. Therefore we naturally employ group by to express nesting in our language.

As an example, suppose we want to define a view on the information concerning polygons, structured in such a way that polygon identifiers are grouped by the points that are elements or their lists. For this we can use:

```
select elt, contained_in
from
  (select Polygon, elt
  from Polygon)
group by elt into contained_in
```

Here, *contained_in* is the name of the new complex attribute created by the nest operation. Disregarding the sequence number #, we have thus expressed the restructuring shown in Fig. 3.

As in standard SQL, in queries of the form select ... from ... group by, where both select and group by occur on the same lexical level, the group by-clause is performed before the select-clause. In other words, nesting is done before projection and unnesting. Since the restructuring of Fig. 3 requires unnesting the *list* attribute *before* nesting by *elt*, the above query needs a subquery in its from-clause. So, without from-subqueries, certain restructuring cannot be specified.

This provides us, apart from Date's argumentation (cf. above), with an important reason for including from-subqueries in our language. Note the contrast with the fact (e.g. [23]) that adding

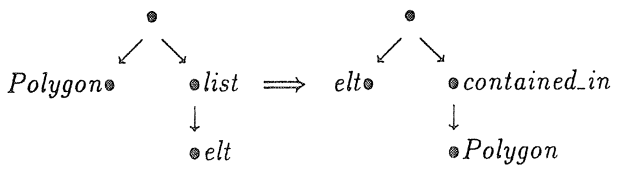


Fig. 3. Restructuring.

from-subqueries to *standard* SQL would *not* render it more powerful. The reason that it *does* in our case is due to the presence of complex objects.

3.8. Aggregate functions

We next mention how SQL aggregates and external computations are treated.

Suppose we want to define a view where polygons are shown together with their diameter. The diameter of a polygon $P = (p_1, \ldots, p_n)$ equals $\max_{i,j} d(p_i, p_j)$, where $d(p_i, p_j)$ denotes the distance between points p_i, p_j. Recall that in our Geometry database we have a method *Point.distance*(*Point*) : *Real*, computing the distance between its two argument points. The view can be specified as follows:

```
select Polygon, distances.max
from
  (select Polygon, first.elt.distance(second.elt)
  from
  (select first.Polygon, first.elt, second.elt
  from Polygon first, Polygon second
  where first.Polygon = second.Polygon))
group by Polygon into distances
```

Note how *distances.max* is used and written in exactly the same way as method calls. In general, we do not treat standard SQL aggregate functions (count, max, ...) differently from other methods. So *max* is a method {*Real*}.*max* : *Real*, defined on values (sets of numbers in this case), as explained in the previous section.†

3.9. Nested application of queries using select-subqueries

In the previous query, each polygon list is unnested in order to allow for a Cartesian product with itself. After computing the distance of every pair of points thus obtained, the structure is nested back into form. Several complex object query languages were proposed, e.g. [24–26], where arbitrary operations, like selection or Cartesian product, can be performed directly on the nested information, thus avoiding explicit restructuring. In fact, our language already supports an important special case of such a mechanism, namely the application of selections at lower levels (cf. Subsection 3.4). In Ref. [24], a proposal was made to incorporate this feature in SQL by allowing subqueries in select-clauses. Using nested select-clauses, the diameter view can alternatively be specified as follows:‡

```
select Polygon,
  (select first.elt.distance(second.elt)
  from list first, list second).max
from Polygon
```

This also illustrates how the method *max* can be applied directly to subqueries.

3.10. Conclusion

Summarizing, we would like to re-emphasize that all features for dealing with complex objects and object-orientation are obtained by natural extensions of existing SQL constructs. The only fundamental extensions made to standard SQL are the subqueries in from-clauses and select-clauses, and the equal treatment of complex attributes as relations.

4. SYNTAX AND SEMANTICS

In this section we formally specify our query language's syntax and semantics. At the end of the section we give a few examples, illustrating the most important utilized techniques. We define the

†Of course, for reasons of portability, it may be desirable to also still support the current SQL notation max(*distances*), which would then automatically be converted to the newer object-oriented method notation.

‡Although this way of formulating the diameter view is elegant, its appearance is quite distant from standard SQL, in contrast with the previous formulation.

formal semantics by inductively defining a translation of each expression of the language into a query described in the nested relational algebra [26].

We make this strategy work by employing the mapping from our object-oriented data model to nested relations, given in Section 2, viewing the object-oriented database as a nested relational one. This gives us the major advantage of being able to reduce the considerable task of formally specifying syntax and semantics of a query language for object-oriented databases to a somewhat more accessible task, by assuming that the query language works solely on nested relations. A secondary advantage of this approach is that the result of a query on an object-oriented database can be simply defined to be a nested relation. This provides us with an easy, at least initial solution to the problem of how the result of an object-oriented query should be formally defined [27], and is, besides the modeling of methods, the reason why we choose to include relationships on first-class basis in our data model. In Section 5, we briefly touch upon the issue of interpreting the nested relational query result alternatively as a *derived class* or *class view*.

Our language is both syntactically and semantically a natural extension of SQL. We assume the reader to be familiar with standard SQL terminology.

4.1. Preliminaries

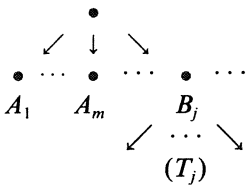
Here, we introduce various preliminary notions which will needed in our further formal development.

First we recall the well-known fact that (tuple) types induce a *tree structure* on the attributes occurring in them. In what follows, "types" stand for "tuple types", unless stated otherwise. In particular, in what follows, let T be the type:

$$[A_1 : D_1, \ldots, A_m : D_m, B_1 : \{T_1\}, \ldots, B_n : \{T_n\}], \qquad (*)$$

where the D_i are atomic types. Following standard nested relational terminology, we put:

Definition 4.1. $ato(T) := \{A_1, \ldots, A_m\}$ (*atomic attributes*), $com(T) := \{B_1, \ldots, B_n\}$ (*complex attributes*). Elements of $ato(T) \cup com(T)$ are called *top-level* attributes of T. We now associate with T the tree of the form:



So, the root node is unlabeled, and has one child node for each top-level attribute. The nodes labeled with atomic attributes are leafs, while at each node labeled with a complex attribute B_j the tree of T_j originates inductively, as a subtree. In general, for *any* node occurring in T, we define:

Definition 4.2. If A is a leaf node, the *type* of A is the atomic node type D for which $A : D$ occurs in T. If B is an internal node, the type of B is the type corresponding to the subtree originating at B.

The correspondence between types and trees enables us to use standard tree terminology in the context of types and schemes, such as child, parent, sibling, ancestor, descendant and isomorphism. This is exploited in the following Definitions 4.3, 4.4 and 4.5, where we define isomorphic and compatible types and subtypes.

Informally, two types are isomorphic if two values of that type have the same structure. Formally:

Definition 4.3. Two types are *isomorphic* if their associated trees are isomorphic.

The notion of compatibility directly reflects the class hierarchy and extends it to tuple types. Informally, type T' is compatible with type T if wherever a value of type T is expected,† a value of type T' may appear according to the class hierarchy.

†In our case, this will be as an argument of a method call, or when taking the union, difference of intersection of two typed sets.

Definition 4.4. Type T' is called *compatible* with type T if there is a bijective mapping θ from the top-level attributes of T' to the top-level attributes of T such that: (i) for each $A \in ato(T')$, the type of A is less or equal than the type of $\theta(A)$ in the class hierarchy; and (ii) for each $B \in com(T')$, the types of B and $\theta(B)$ are compatible.

It is important to note that: (i) compatibility is *not* a symmetric notion, since the class hierarchy is not symmetric; (ii) if two types are compatible then they are isomorphic.

Definition 4.5. Type T' is called a *subtype* of type T if T' can be obtained from T by removing some (nonroot) nodes from T, where "removing" a nonleaf node means *unnesting* it, i.e. linking all its children as additional children of its parent.

Notice that each subtree of T comprises a subtype. Also, the well-known subtyping relation of Ref. [16], when restricted to the types we consider here, implies subtypes in the above-defined sense. Actually, we have the following characterization:

Proposition 4.6. Let T be the type (*). The subtypes of T can be inductively obtained as follows:

- *Basis*: T is a subtype of T;
- *Projection*: if T' is a type such that:
  1. $ato(T') \subseteq ato(T)$;
  2. $com(T') \subseteq com(T)$; and
  3. for each $B_r \in com(T')$, the type T'_r of B_r in T' is a subtype of T_r,

  then T' is a subtype of T;
- *Unnesting*: if T' is a subtype of T'', where T'' is a type for which there is a $B_r \in com(T)$ such that $ato(T'') = ato(T) \cup ato(T_r)$, $com(T'') = com(T) - \{B_r\}$, and for each $B_j \in com(T'')$, the type of B_j in T'' equals T_j, then T′ is a subtype of T.

Precisely the unnesting rule renders our subtype notion a broader one than that of [16]. However, it will become clear that we use our subtypes for entirely different purposes, so from a semantical viewpoint the two notions are incomparable.

Example 4.7. If $T = [Polygon : Polygon,\ list : \{[\# : Integer,\ elt : Point]\}]$ then $[list : \{[elt : Point]\}]$ is a subtype by the projection rule, and $[Polygon : Polygon,\ \# : Integer]$ is a subtype by applying unnesting followed by projection.

By Proposition 4.6, a subtype specifies a (nested) sequence of projections and unnestings, but up to now only on a syntactical level. Intuitively, it is clear that given a relation r of type T, i.e. $r \subseteq \|T\|$, a subtype T' of T specifies an action on r, described by the same sequence of projections and unnestings.

We make this intuition precise by associating with (T', T) a *pruning* operation (*pro*jection-*un*nesting or π-μ-operation, introduced in Ref. [26]), denoted $\pi\text{-}\mu_{T',T}$. Recall that T has the general format (*). The construction goes along the lines of Proposition 4.6, as follows:

Definition 4.8.

- *Basis*: if $T' = T$, then $\pi\text{-}\mu_{T',T}$ is defined as the identity operation.
- *Projection*: if
  1. $\{A_{i_1}, \ldots, A_{i_k}\} = ato(T') \subseteq ato(T)$;
  2. $\{B_{j_1}, \ldots, B_{j_l}\} = com(T') \subseteq com(T)$;
  3. as in Proposition 4.6, item 3,

  then $\pi\text{-}\mu_{T',T}$ is defined as:
  $$\pi[A_{i_1}, \ldots, A_{i_k}, \pi\text{-}\mu_{T'_{j_1}, T_{j_1}}(B_{j_1}), \ldots, \pi\text{-}\mu_{T'_{j_l}, T_{j_l}}(B_{j_l})]$$
- *Unnesting*: if T'', B_r are as in Proposition 4.6, then $\pi\text{-}\mu_{T',T}$ is defined as:†
  $$\pi\text{-}\mu_{T',T''} \cdot \mu_{B_r}.$$

Example 4.9. Let $T = [Polygon : Polygon,\ list : \{[\# : Integer,\ elt : Point]\}]$.

- For $T' = [list : \{[elt : Point]\}]$, we have:
  $$\pi\text{-}\mu_{T',T} = \pi[\pi[elt](list)].$$

†"·" Denotes concatenation of expressions.

- For $T' = [Polygon : Polygon, \# : Integer]$, we have:

$$\pi\text{-}\mu_{T',T} = \pi\text{-}\mu_{T',T''} \cdot \mu_{\text{list}}.$$

with $T'' = [Polygon : Polygon, \# : Integer, elt : Point]$, whence:

$$\pi\text{-}\mu_{T',T} = \pi[Polygon, \#]\mu_{\text{list}}.$$

In the syntax of our SQL language, in the notation of types, the atomic types will be left out for syntactical simplicity. The square brackets will be left out as well because they are notationally redundant. This is conform with standard SQL notation, where one writes "select A,B" instead of, say, "select [A : string, B : integer]". Formally:

Definition 4.10. If T is a type, its associated scheme $\tilde{T}$ is obtained from T by replacing all occurrences of the form "$A : D$" by "A", where $A \in U$ and D is an atomic type, and further removing all "[" and "]" symbols.

Schemes of tuple types are called *tuple schemes.*

Example 4.11. With T as above, $\tilde{T} = Polygon, list : \{\#, elt\}$.

Clearly, all notions introduced above, in particular the pruning operation, apply equally well to schemes as to types.

4.2. Overview

A typical query expression in our SQL language has the general format:

```
select-clause
from-clause
where-clause
group by-clause
set-operation
...
```

The various clauses correspond to various steps in the query expressed by this expression, in the following order:

1. The from-clause expresses the Cartesian product (up to renaming by the standard SQL range variable technique) of its arguments, which are either relationships or subqueries;
2. The where-clause expresses a restriction (selection) of the result of the preceding step, the condition of which may involve method calls and subqueries;
3. The group by-clause restructures (nests) the results of the preceding step;
4. The select-clause prunes (projection and unnesting) the result of the preceding step, as indicated by its arguments, which are either attributes, method calls or subqueries, applied on complex attributes;
5. Finally, set-operations (difference, intersection, union) are applied.

In the remainder of this section we will inductively specify all possible query expressions E, following the above order, and define their semantics by a query $Q(E)$ in the nested algebra. In the syntax specifications, we will employ usual notation: in particular, square brackets denote an optional construct, ellipses denote finite repetition and vertical bar means *or*.

4.3. From-clauses

Syntax: Consider a query expression E of the form:

```
select *
from from-arg, ...
```

A *from-arg* is a *rel-spec* optionally followed by a range variable. A *rel-spec* is either an identifier (as in standard SQL), or is a subquery of the form (E'), where E' is a query expression.

Semantics: For each *rel-spec*, Q(*rel-spec*) is defined as follows:

- if *rel-spec* is an identifier, Q(*rel-spec*) is the relation denoted by the identifier;

- if *rel-spec* is a subexpression (E'), then $Q(\textit{rel-spec}) := Q(E')$.

Now for each *from-arg*, $Q(\textit{from-arg})$ is defined as follows:

- if *from-arg* is a *rel-spec* followed by a range variable v, then $Q(\textit{from-arg}) := \rho_\phi Q(\textit{rel-spec})$, where ρ_ϕ is the renaming operation with renaming function ϕ which maps every attribute A occurring in the result-type of $Q(\textit{rel-spec})$ to $v.A$;
- otherwise, *from-arg* is just a *rel-spec* and $Q(\textit{from-arg}) := Q(\textit{rel-spec})$.

Thus, we treat range variables simply and elegantly as renamings, by assuming that if $A \in \mathcal{U}$ and v is a range variable, then also $v.A \in \mathcal{U}$, i.e. is also a legal attribute name.

$Q(E)$ is now defined as the Cartesian product:

$$Q(\textit{from-arg}) \times \cdots$$

for all *from-args* in the from-clause of E.

4.4. *Where-clauses*

4.4.1. Basics. **Syntax:** Consider a query expression E of the form:

select *
from-clause
where $term_1$ relop $term_2$

We denote the part of E without where-clause by E^{-w}. As for now,† the syntax of *term* is:

term ← attribute-name | *restructuring* | subquery | constant
restructuring ← attribute-name : {tuple-scheme}

The comparison operator (relop) is any of the usual ones, including the standard SQL predicates =, contains, in, ... In fact, anticipating Subsection 4.4.2, where method calls will be considered, it can be seen that in principle it is sufficient to have = as only relop, since arbitrary predicates can be tested by a call to a Boolean-valued method/function defined in the underlying object-oriented system. For example, one could have a method:

$$\textit{Number}.<(\textit{Number}) : \textit{Boolean},$$

(in the style of Smalltalk), and write "$A.<(B) = \textit{true}$", which of course can be syntactically sugared into "$A < B$".

As constants we allow any value as defined in Section 2.

Semantics: By induction, we may assume that $Q(E^{-w})$ is already defined. In general, $Q(E)$ is defined as the *selection*:

$$\sigma[Q(\textit{term}_1)\text{relop}\, Q(\textit{term}_2)] \cdot Q(E^{-w}),$$

where $Q(\textit{term}_i)$ is defined according to the following rules:

- If *term* is an attribute name, then it must be a top-level attribute of the result type T^{-w} of $Q(E^{-w})$. $Q(\textit{term})$ is defined as *term* itself; we also define the type of *term* as its type in T^{-w}.
- Similarly, if *term* is a constant, $Q(\textit{term})$ is *term* itself; the type of the term is obvious.
- If *term* is a restructuring of the form $Z : \{S'\}$, where Z is an attribute name and S' is a tuple scheme, then Z must be in $com(T^{-w})$. $Q(\textit{term})$ is defined as $\pi - \mu_{S',S}(Z)$, where S is the type of Z. The type of the term is the result type of the pruning operation.
- If *term* is a subquery of the form (E'), then $Q(\textit{term})$ is defined as $Q(E')$ and the type of the term is the result type of $Q(E')$.

Remark how we can conveniently exploit the facility of the nested relational algebra to use subqueries in selection conditions [26]. We also have the standard requirement that in the where-clause, the types of the $term_i$ must be properly typed with respect to the relop. For example, in "A_1, A_2 in B", if the type of A_i is T_i, then that of B must be the set type $\{T\}$, where T is such that the tuple type obtained by concatenating the T_i is compatible‡ with T.

†In Subsection 4.4.2 we will extend terms to include method calls.
‡Recall Definition 4.4.

4.4.2. Method calls. **Syntax:** Since we want to allow method calls in comparison, we extend our syntax:

normal-term ←(term as defined up to now)
term ←*normal-term* | *attribute-call* | *method-call*
attribute-call ←attribute-name . attribute-name
attribute-call · attribute-name
method-call ←*term* . method-name(*term*, . . .)

So, we discriminate *attribute calls*, i.e. method calls where the method name is an attribute name—which (as mentioned earlier) can be seen as unary methods—from more general methods defined in the system. This distinction is made since below, in defining the formal semantics, both types of method call will need a different treatment.

We again consider query expressions E, E^{-w} as above, but now one (or both) of the $term_i$ is a method call.

Semantics: We define the semantics of method calls by reducing them to normal terms. That is, we construct from E an equivalent query expression E' by replacing the method calls by normal terms (in fact simply by attribute-names), and adapting the from- and where-clauses in an appropriate way. Intuitively, what will be done is to "join in" the relationship (or class in case of an attribute call) corresponding to the method.

For simplicity we assume that the receiver and arguments of a method or attribute call are normal terms, i.e. no method calls in turn, but the construction can be straightforwardly generalized to deal with this.

First consider an *attribute-call* of the form $A . B$, where A, B are attribute names. We know that the type of A is a class name (the receiver class), say C. E' is obtained as follows. The select-clause of E' is that of E. The from-clause of E' is that of E, augmented with the additional *from-arg* (cf. Section 4.3):

$$C\ v.$$

That is, C is the *rel-spec* and v is an arbitrary but fixed rangle variable uniquely corresponding to the attribute call. The where-clause of E' is obtained from that of E by replacing A by v in the considered occurrence of the attribute call, and by adding the conjunct:

$$\text{and} \quad A = v.C.$$

The logical connective and will be treated in Subsection 4.4.4. Remark that $v.B$ and $v.C$ are legal attribute names, since v is a range variable, as was explained in Subsection 4.3.

Next consider a *method-call*, having the form:

$$t_0 . m(t_1, \ldots, t_n),$$

where m is the name of the method, and the t_i are normal terms. The signature of the method is of the form:

$$T_0 . m(T_1, \ldots, T_n) : T.$$

It is naturally required that the type of t_i is compatible with T_i, for $0 \leqslant i \leqslant n$. We distinguish two cases:

1. T is an atomic type. Then the type of the relationship representing m is:

$$[receiver : T_0,\ arg_1 : T_1, \ldots, arg_n : T_n,\ result : T].$$

The select-clause of E' is that of E. The from-clause of E' is that of E, augmented with the additional *from-arg*:

$$m\ v.$$

That is, m (standing for the relationship representing the method) is the *rel-spec* and v is an arbitrary but fixed rangle variable uniquely corresponding to the method call. The where-

clause of E' is obtained from that of E by replacing the considered occurrence of the method call by *v.result*, and by adding the conjuncts:

$$t_0 = v.receiver \quad \text{and} \quad t_1 = v.arg_1 \quad \text{and} \cdots \text{and} \quad t_n = v.arg_n.$$

2. T is a tuple type (i.e. m is a state-changing method). If T has the general format:

$$[result_1 : T_{n+1}, \ldots, result_k : T_{n+k}],$$

then the type of the relationship representing m has the form:

$$[receiver : T_0, arg_1 : T_1, \ldots, arg_n : T_n, result_1 : T_{n+1}, \ldots, result_k : T_{n+k}].$$

In this case, the method call occurs in a subexpression *method-call.result*$_i$ of a larger term.† So in obtaining the where-clause of E', instead of replacing the method call by *v.result* as above in obtaining the where-clause of E', we now replace it by just v itself. Everything else in the construction of E' is similar to the above.

4.4.3. More comparisons. Standard SQL provides some more comparisons in where-clauses, which we discuss here, together with some other ones which are specific to our complex object-oriented SQL.

The expression "exists (E^s)", where E^s is a subquery, is reduced to the equivalent form "(E^s)$\langle\,\rangle$*empty*".

The quantifiers any and all can be used in our language in the same way as they are in standard SQL. Moreover, we extend their meaning and usage to express selections directly inside complex attributes, which we will formally introduce next.

Syntax: Consider a query expression E of the form:

select *
from-clause
where $term_1$ relop quantifier $term_2$

where the quantifier is one of any and all. By Subsection 4.4.2, we can assume that the $term_i$ do not contain method calls, i.e. are normal terms. Moreover here, we require them to be attribute names; the case of one or both of them being *restructurings* goes analogous.

Semantics: As in Subsection 4.4.1, $Q(E)$ is defined as:

$$\langle\sigma\rangle[term_1 \text{ relop quantifier } term_2] \cdot Q(E^{-w}).$$

However now, unlike as in Subsection 4.4.1, the $term_i$ do not have to be top-level attributes of T^{-w}. Indeed: the powerful selection operation $\langle\sigma\rangle$ that we use here is not the usual selection operation of the nested relational algebra. It can work directly within complex values, possibly appearing deeper inside the nested structure.‡

The effect of $\langle\sigma\rangle$ is defined as follows. Let R be a nested relation of type T. Let A, B be attribute names occurring in T. Let Z be the type of the greatest common ancestor of A and B in T, and let P_A (P_B) be the type of the parent of A (B) in T. Let t be an appearance of a tuple value of type P_A in R. Denote the tuple over Z appearing in R, in which t appears, by $\bar{t}$. Then we say that t *satisfies* "A relop quantifier B" if one of the two following conditions is true:

- if quantifier = any, then there exists a tuple t' over P_B appearing in $\bar{t}$ such that $t'(B)$ relop $t(A)$.
- if quantifier = all, then for all tuples t' over P_B appearing in $\bar{t}$, $t'(B)$ relop $t(A)$.

Finally, $\langle\sigma\rangle[A \text{ relop quantifier } B](R)$ is the nested relation obtained from R by filtering out the appearances t over P_A that do not satisfy "A relop quantifier B". We say that the selection *works on level* P_A.

Note that if $P_A = P_B$, then the quantifier is redundant, since then $t = \bar{t} = t'$. Hence, in this case, the quantifier may be omitted in expression E. In particular, if $P_A = P_B$ is the root of T, then we

†Because tuple-valued terms are not supported directly.
‡An example of this was given at the end of Section 3.4.

obtain the usual standard selection operation of Subsection 4.4.1 as a special case, which works on the level of the root.

We refer to Ref. [48] for a detailed study of extended selection operations for nested relations.

4.4.4. Logical connectives. To conclude the part on where-clauses, we treat the logical connectives and, or, not.

Syntax: We consider the following *conditions* that can occur in a where-clause:

condition ← *comparison*
condition ← *condition*$_1$ and *condition*$_2$
condition ← *condition*$_1$ or *condition*$_2$
condition ← not *condition*
condition ← (*condition*)
comparison ← *term*$_1$ [quantifier] relop *term*$_2$

By the preceding Subsections 4.4.2 and 4.4.3, it is indeed sufficient to consider comparisons of the above form. Now consider a general expression E of the form:

select *
from-clause
where *condition*

E^{-w} is defined as before.

Semantics: We define $Q(E)$ for the different syntactical possiblities of the where-condition. The case that the condition is a comparison is already known:

1. The where-condition has the form not *condition*. By De Morgan's rules, we may assume that *condition* is actually a *comparison*. Let E' be the expression:

   $$E^{-w} \text{ where } comparison$$

   Then we know that $Q(E')$ has the form $\sigma[C] \cdot Q(E^{-w})$ or $\langle\sigma\rangle[C] \cdot Q(E^{-w})$, where C is a selection condition. We now simply define $Q(E)$ to be:

   $$\sigma[\text{not } C] \cdot Q(E^{-w}),$$

   or

   $$\langle\sigma\rangle[\text{not } C] \cdot Q(E^{-w}),$$

   depending on the form of $Q(E')$.†
2. Next, suppose the where-condition has the form *condition*$_1$ binop *condition*$_2$, where binop is one of and, or. Let, for $i = 1, 2$, E_i be the expression:

   $$E^{-w} \text{ where } condition_i.$$

   By induction, $Q(E_i)$ is known, and we consider two cases:

   - Both $Q(E_i)$, for $i = 1, 2$, have the form $\sigma[C_i] \cdot Q(E^{-w})$, where $\sigma[C_i]$ is a standard selection of the nested relational algebra. Then $Q(E)$ can be simply defined as:

     $$\sigma[C_1 \text{ binop } C_2] \cdot Q(E^{-w}),$$

     since $\sigma[C_1 \text{ binop } C_2]$ is also a standard selection.
   - One or both of the $Q(E_i)$ has the form $\langle\sigma\rangle[C_i] \cdot Q(E^{-w})$, where $\langle\sigma\rangle[C_i]$ is a "deep-level" selection operation, as defined in Subsection 4.4.3. We will assume that both the $Q(E_i)$ has this form; the general case is analogous. Again we consider two cases:

     —Both the selections $\langle\sigma\rangle[C_i]$, for $i = 1, 2$, work on the same level. Then it is straightforward to give a meaning to $\langle\sigma\rangle[C_1 \text{ binop } C_2]$ and $Q(E)$ can still be defined as:

     $$\langle\sigma\rangle[C_1 \text{ binop } C_2] \cdot Q(E^{-w}).$$

†The semantics of the negated version of $\langle\sigma\rangle$-selection selection is defined similarly to that of the unnegated version of Subsection 4.4.3.

—They do not work on the same level. This is a somewhat esoteric case: indeed, it is readily seen that it is not at all clear what $\langle\sigma\rangle[C_1 \text{ binop } C_2]$ might mean. We propose to define $Q(E)$ as:

$$\begin{cases} \langle\sigma\rangle[C_2]\cdot\langle\sigma\rangle[C_1]\cdot Q(E^{-w}) & \text{if binop is and;} \\ Q(E_1)\cup Q(E_2) & \text{if binop is or.} \end{cases}$$

The handling of or is inspired by the situation in the relational algebra, where a disjunction in a selection condition is equivalent to a union. For and, we could have taken the corresponding option of intersection, but choose instead the alternative shown above. Note that in relational databases:

$$\sigma[C_1 \text{ and } C_2, \quad \sigma[C_1]\cap\sigma[C_2] \quad \text{and } \sigma[C_2]\cdot\sigma[C_1].$$

are all equivalent, but in a complex object setting they are not. We believe that the latter alternative is the most natural in this setting. A consequence however is now that, in this case, and may no longer be commutative in general (see Ref. [48] for more details).

3. Finally, the case that the where-condition has the form (*condition*) is trivial.

4.5. Group by-clauses

We now turn to group by-clauses, and show how their semantics can be defined using the *nesting* operation of the nested relational algebra. We point out that also in standard SQL, the group by-facility essentially is a restricted form of nesting. However there, the set of grouped data, constructed by the nesting, can only be used for performing an aggregate computation, such as count or max. In our context, this is not required, and the newly constructed grouping set is simply part of the query result.

Syntax: Formally, consider an expression E of the form:

select *
from-clause
where-clause
group by attribute-name$_1$, ..., attribute-name$_n$ into attribute-name

n may be zero, in which case the group by-clause reduces to group by into attribute-name. We will denote the part of E without group by-clause by E^{-g}.

Semantics: By introduction, we may assume that $Q(E^{-g})$ is already defined. Let T^{-g} be the resulting type of $Q(E^{-g})$. All the attribute-name$_i$ in the group by-clause must occur as siblings in T^{-g}, i.e. they must have a common parent P. Let T_P denotes the type of P. Then the attribute$_i$ are required to be top-level attributes of T_P, and let N denote the set of all other top-level attributes of T_P. We also require that the attribute-name of the into-part of the group by-clause does not appear in T^{-g}. $Q(E)$ is now defined as:

$$\nu[N\rightarrow\text{attribute-name}]\cdot Q(E^{-g}),$$

where ν stands for nesting.† This nesting operation is well-defined due to the above requirements on the attribute-name$_i$'s and the attribute-name.

Remark: repeated nestings: Suppose one wants to nest repeatedly, as in:

$$\nu[A, B\rightarrow C]\cdot\nu[D, E\rightarrow F].$$

One possibility that our language provides for this is to use a subquery in the form-clause, as in:

select-clause
from
  (select-clause
  from-clause
  where-clause

†Recall from nested relational algebra that the effect of this nesting is that in each nested relation over T_P occurring in the instance, tuples with equal attribute$_i$-values are grouped into one new tuple with a new complex attribute, containing the values of these tuples on the remaining attributes, the attributes in N.

group by A, B into C)
group by D, E into F

An alternative could be to enhance the language slightly, allowing multiple group by-clauses, as in:

select-clause
from-clause
where-clause
group by A, B into C
group by D, E into F

4.6. Select-clauses

4.6.1. Basics. In standard SQL, the select-clause essentially expresses a projection operation. As we saw at the end of Subsection 4.1, a natural generalization of projection to complex structures is *pruning*.

Syntax: Formally, consider an expression E of the form:

select tuple-scheme
from-clause
where-clause
group by-clause

The where- and group by-clauses are optional.

Let E^{-s} denote the expression obtained from E by replacing the select-clause with select *.

Semantics: By induction, $Q(E^{-s})$ is already defined; let T^{-s} be its resulting type. Then we require that the tuple-scheme S in the select-clause of E is a subscheme of T^{-s}. $Q(E)$ is now simply defined as (cf. Definition 4.8):

$$\pi\text{-}\mu_{S,T^{-s}} \cdot Q(E^{-s}).$$

4.6.2. Explicit renaming using into. We already saw that the usage of range variables boils down to applying a renaming. Our SQL also provides a mechanism for explicit renaming, using an optional into-construct in the select-clause.

Syntax: Formally, consider an expression E as above, but now with a select-clause of the form:

select tuple-scheme$_1$ into tuple-scheme$_2$.

Let E^{-i} denote E without the into-part.

Semantics: We require that tuple-scheme$_1$ and tuple-scheme$_2$ are isomorphic. Note that there can be several different isomorphisms from tuple-scheme$_1$ to tuple-scheme$_2$, let θ be the "canonical" one, determined by the particular order in which the attribute names occurring in the tuple schemes are written down in E. Observing that θ can be used as a renaming function, we then define $Q(E) := \rho_\theta Q(E^{-i})$.

4.6.3. More select-clauses. In the above, we considered basic select-clauses, consisting of a tuple scheme, expressing a pruning operation, as natural extension of the basic select-clause of standard SQL which expresses projection.

However, standard SQL also allows the application of aggregate functions in select-clauses. We generalize this to our object-oriented setting by allowing arbitrary method calls. Moreover, as mentioned earlier, for achieving maximal effectiveness in treating complex structures, we also allow subqueries in select-clauses.

Syntax: Formally, we extend the syntax of tuple schemes (cf. Definition 4.10), by allowing arbitrary *terms*, as defined in Subsection 4.4, instead of only attribute-names to occur in a tuple scheme.

Semantics (sketch): The formal semantics of the extended select-clauses described above can again be conveniently defined using the π-μ operation of the nested relational algebra. This operation itself already provides subqueries [26]. Hence, the only construct we have to handle explicitly are the method calls. The strategy for dealing with method calls is essentially the same

as that used in processing method calls in where-clauses, as was exposed in detail in Subsection 4.4.2.

So, every method call is reduced to an attribute name by "joining in" the relationship corresponding to the method. This is done by adapting the from- and where- clause of E, if the method call occurs on the top-level in the tuple scheme, or otherwise in a subquery at the level of the type most closely surrounding the method call; we just mentioned that subqueries pose no problem. We leave the details as an exercise to the reader; illustrative examples are given in Subsection 4.8 (in particular, see the last one).

4.7. Set operations

We finally have arrived at the last stage, set operations. Following the original SQL proposal of Ref. [8], our language orthogonally supports the three basic set operations: union, difference and intersection. The same is done in SQL2 [9].

Syntax: Consider an expression E of the form:

$$E_1$$
$$\text{setop}$$
$$E_2$$

where setop is one of union, intersect, minus and where the E_i are general query expressions of the form select-from-where-group by (the where- and/or group by-clauses are optional).

Semantics: By induction, the $Q(E_i)$ are already known. Let T_i be the resulting type of $Q(E_i)$. In standard SQL, T_1 and T_2 are required to be equal. In our object-oriented contex, we only require T_2 to be compatible with T_1 (recall subsection 4.1). Since this implies that T_1 and T_2 are isomorphic, we can consider the canonical isomorphism θ from T_1 to T_2, as in Subsection 4.6.2. We now define $Q(E)$ as:

$$\begin{cases} Q(E_1) \cup \rho_\theta Q(E_2) & \text{if setop} = \textsf{union;} \\ Q(E_1) \cap \rho_\theta Q(E_2) & \text{if setop} = \textsf{intersect;} \\ Q(E_1) - \rho_\theta Q(E_2) & \text{if setop} = \textsf{minus.} \end{cases}$$

The resulting type of $Q(E)$ then is T_1.

Note that, due to our rather strict typing requirements, union and intersect are no symmetric operations. However, loosening the requirements would yield paradoxical situations. e.g. assume E is

```
select Point
from Point
  intersection
select Intersection
from Intersection
```

Then $T_1 = [Point]$ and $T_2 = [Intersection]$. Since $Intersection \leqslant Point$, T_2 is compatible with T_1. According to the above typing rule, the result type T of E is $[Point]$. However, assume that we instead would define T as $[Intersection]$, which is (at first sight) more intuitive. Then, if $E_1 - E_2$ would still be defined to be of type $[Point]$, which is natural, the set-theoretic truth:

$$E_1 \cap E_2 = E_1 - (E_1 - E_2),$$

would no longer be valid w.r.t. the typing: the left- and right-hand side would then have different types.

4.8. Examples

We end this section by giving a few examples, illustrating the most important utilized techniques. We review some examples given in Section 3, and give their translation into the nested relational algebra.

Let E be:

```
select Polygon, Polygon.rotate(c_45).list into Polygon, rot_list
from Polygon
```

Then E is first reduced to the equivalent E', eliminating the method call:

select *Polygon*, *v.list* into *Polygon*, *rot_list*
from *Polygon*, *rotate v*
where *Polygon* = *v.receiver* and *c_45* = *v.arg*

$Q(E')$ now is:

$$\rho[v.list \rightarrow rot\_list] \cdot \pi[Polygon, v.list] \cdot \sigma\begin{bmatrix} Polygon = v.receiver \\ \&v.arg = c\_45 \end{bmatrix} \cdot \left(Polygon \times \rho\begin{bmatrix} receiver \rightarrow v.receiver \\ arg \rightarrow v.arg \\ list \rightarrow v.list \end{bmatrix}(rotate) \right)$$

To give an example of the treatment of state-changing methods, consider the query:

select *Intersection*
from *Intersection*
where
Intersection.mirror(*v1*).*mX* = *X* and
Intersection.mirror(*v1*).*mY* = *Y* and
Intersection.mirror(*v2*).*mX* = *X* and
Intersection.mirror(*v2*).*mY* = *Y*

Eliminating the method- and attribute-calls, the query reduces to:

select *Intersection*
from *Intersection*, *mirror.r1*, *mirror.r2*
where *r1.receiver* = *Intersection* and *r1.arg* = *v1* and
r2.receiver = *Intersection* and *r1.arg* = *v2* and
r1.mX = *X* and *r1.mY* = *Y* and *r2.mX* = *X* and *r2.mY* = *Y*

Note that *v1*, *v2* are attribute names and *r1*, *r2* are range variables. The translation of the above expression to the algebra is now easy.

As another example, consider:

select *first.Polygon*, *second.Polygon*
from *Polygon first*, *Polygon second*
where not exists
(select *first.elt* from *first.list*
where *first.elt* = any *second.elt*)

In the nested relational algebra, this query becomes:

$$\pi[first.Polygon, second.Polygon] \cdot \sigma[\pi[first.elt]\langle\sigma\rangle[first.elt = \text{any } second.elt](first.list) = \emptyset] \cdot (\rho[\_\rightarrow first.\_](Polygon) \times \rho[\_\rightarrow second.\_](Polygon))$$

Another interesting example is the following:

select *elt*, *contained_in*
from
(select *Polygon*, *elt*
from *Polygon*)
group by *elt* into *contained_in*

The corresponding algebra query is:

$$v[Polygon \rightarrow contained\_in] \cdot \pi[Polygon, elt] \cdot \mu[list](Polygon)$$

As a final example, consider:

select *Polygon*,

(select *first.elt.distance*(*second.elt*)
from *list first*, *list second*).*max*

Its translation into the nested relational algebra is:

$$\pi[Polygon,\ max.result] \cdot \sigma\Big[max.receiver = \pi[distance.result] \cdot \sigma[distance.receiver = first.elt\ \&\ distance.arg = second.elt] \cdot (\rho[\_ \to first.\_](list) \times distance)\Big]$$

$$(Polygon \times max)$$

4.9. Conclusion

We have shown in this section that every query expressible in our object-oriented query language is expressible in, and in fact is defined in terms of, the nested relational algebra. It can be shown [23] that also the converse of the above property holds: every query expressible in the nested relational algebra has an equivalent in object-oriented SQL. Thus, we can conclude that our language is *relationally sound and complete*.

We can now conclude by defining a given syntactical expression E in our SQL language to be semantically correct if its reduction $Q(E)$, as was defined in this section, is a semantically correct algebraic expression. This provides us with an operational semantics for our language, as well as a strong and static typing mechanism.

5. DISCUSSION

In this paper we have attempted to show that it is possible to use a natural, conservative extension of SQL to query object-oriented databases. On the model level we achieved this by moving from standard relations to nested relations, mapping an object-oriented database to a (nested) relational database. Correspondingly, on the language level we mapped object-oriented features (such as inheritance, object identity and method calls) into nested relational constructs. Our task was then finally reduced to the design and formal definition of a natural extension of SQL for nested relations, which turned out to be quite manageable using a translation to the nested relational algebra.

So, the three main intended contributions of this paper are: (i) staying to standard SQL as closely as possible; (ii) complete and precise definitions of syntax and semantics; and (iii) an effective translation of the language into an optimizable query algebra. As already mentioned in the Introduction, we certainly do not have a religious belief in feature (i), nor do we think that non-SQL-based languages are uninteresting; rather, our work examines one possible way to go.

An early paper arguing in favor of an "evolutionary" approach to object-oriented database systems (including staying as closely as possible to standard SQL) is Ref. [12]. There, a language called OSQL is presented in an informal way (cf. also Ref. [28]). The data model used supports no built-in set types (instead one has a class "Set"). Therefore, the mapping from classes and methods to relations is flat, not nested.

SQL-like query languages for nested relations have been known for quite some time [23, 24, 29, 30], and our work has benefited from the experience gained there. References [23, 29] are most close to our philosophy of natural, conservative extension of standard SQL. In particular, the ideas of using the select-clause for expressing unnesting, and the group by-clause for nesting were already present here. The extensions to SQL proposed in Refs [24, 30] depart much more from the basic paradigm of standard SQL.

More recently, various query languages for object-oriented databases have been proposed that are inspired by SQL (e.g. Refs [31–34]). O_2 Query [31] is a very flexible language, but is quite distant from standard SQL. Its semantics has been formally defined using a calculus framework [35]; its translation into an algebra, as presented in Ref. [36], is quite rudimentary at best. The languages ESQL2 [33] and XSQL [34] are formally defined in a calculus framework; no operational semantics in terms of an algebra are provided. Furthermore, the manipulation of set values in these languages is not nested relational in perspective, but is based on object creation (see later). The syntax of

Polygon	#
δ_1	1
δ_1	2
δ_1	3
δ_1	4
δ_2	1
δ_2	2
δ_2	3
⋮	⋮

Pair	*Polygon*	#
ϵ_1	δ_1	1
ϵ_2	δ_1	2
ϵ_3	δ_1	3
ϵ_4	δ_1	4
ϵ_5	δ_2	1
ϵ_6	δ_2	2
ϵ_7	δ_2	3
⋮	⋮	⋮

Fig. 4. An object-creating class view.

ESQL2 is an extension of SQL2. In contrast, the syntax of XSQL is *ad-hoc* and quite distant from standard SQL. The language CQL++[32] is based on the data model of C++.

We also mention Object SQL [37], one of the first commercial SQL-like query language for object-oriented databases. Object SQL is implemented on a commercial basis, and therefore is far less sophisticated than our research proposal and the ones cited above, whose implementation is currently at best in a prototyping stage.

A branch of related research which is also relevant is that on *views*. Views in object-oriented database systems have received considerable attention recently (e.g. Refs [34, 38, 39]. In relational or nested relational databases, a view is simply a (nested) relation, derived from the database using the query language. In object-oriented databases however, the presence of object identity and inheritance complicates matters.

The issue of object identity comes in when we want to define *class views*, i.e. interpret the result of a query, which in our framework is formally a nested relation, as a (derived) class. The problem is that, although every class can be seen as a nested relation, as explained in Section 2, the converse is not true. A given nested relation represents a class only if there is an *identifying attribute*, carrying the name of the class and containing object identifiers, which is moreover a key for that relation. If this is the case, then each tuple of the relation represents a different object according to the value of the identifying attribute; the attributes other than the identifying attributes represent the state of the object. So, when expressing a query intended to define a class view, care must be taken that the just mentioned conditions are satisfied. The reader is invited to determine which queries of Section 3 can indeed serve as class view definitions.

Using the strategy for defining class views just described, the identifying attribute will hold objects that already appeared in the database. Apart from such *object-preserving* views, one also needs *object-creating* views, where the objects in the derived class can be new. As observed in Ref. [40], object creation can be conveniently expressed using function (or, in our framework, method) calls. For example, consider the query:

select *Polygon*, # from *Polygon*

This query returns pairs (p, n), where p is a polygon and n is a sequence number in the list of points of the polygon. The result is shown in Fig. 4, left. Since there are several sequence numbers for a single polygon, the relation resulting from this query cannot be interpreted as a derived class. However, we can use object creation and define a new class *Pair*, consisting of new objects whose states are the (p, n) pairs, as follows:

create class view *Pair* as
select *Pair.new*(*Polygon*, #), *Polygon*, # from *Polygon*

The result is shown in Fig. 4, right. In the style of object-oriented programming languages such as Smalltalk, method *new* is called directly to the class *Pair* to create new objects of that class. Following Ref. [40], the arguments *Polygon* and # ensure that a new object will be created for each (p, n) pair. Essentially the same object-creation mechanism is present in XSQL [34], but with a different, somewhat *ad-hoc* syntax. We believe that our syntax is more natural, closer both to

standard SQL and the object-oriented paradigm. We should mention that object creation in XSQL is also used for the manipulation of sets.

Inheritance also plays a role in the issue of views; it is often desirable to connect the derived class with the other classes in the class hierarchy. Initial investigations on this subject can be found in Ref. [39, 41], to which we refer for the details.

To conclude this section, we briefly discuss *updates*. Our SQL language proposal has focused on queries; however, standard SQL also provides a number of update constructs. It might be interesting to try to extend these constructs in a natural way to object-oriented databases. An alternative, simple and straightforward way to specify updates would be to specify a query, the result of which is the new value for the particular relation or class one wants to update. However, the naive implementation of such an update mechanism will be inefficient. The design of natural languages for updates on complex object or object-oriented databases is a challenging issue for further research.

REFERENCES

[1] A. Heuer and J. Van den Bussche. Using SQL with object-oriented databases. In *Second Workshop on Foundations of Models and Languages for Data and Objects*, Informatik-Bericht 90/3, (J. Göers and A. Heuer, Eds) pp. 103–122. TU Clausthal (1990).

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik. The object-oriented database system manifesto. In Ref. [43], pp. 40–57 (1989).

[3] W. Kim and F. H. Lochovsky (Eds), *Object-Oriented Concepts, Databases, and Applications. Frontier Series.* ACM Press, Addison-Wesley (1989)

[4] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In Ref. [44], pp. 159–153 (1989)

[5] C. Beeri, A formal approach to object-oriented databases. *Data Know. Engng* **5,** 353–382 (1990).

[6] M. H. Scholl and H.-J. Schek, A relational object model. In *ICDT'90 Proc.* Number 470 in *Lecture Notes in Computer Science* (S. Abiteboul and P. C. Kanellakis, Eds) pp. 89–105. Springer-Verlag (1990).

[7] J. Ullman. *Principles of Database and Knowledge-based Systems*, Vol. I. Computer Science Press (1988).

[8] D. Chamberlain *et al.* SEQUEL 2: A unified approach to data definition, manipulation and control. *IBM J. Res. Dev.* **20,** 560–575 (1976).

[9] C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, Second Edn (1989).

[10] C. Lécluse, P. Richard and F. Velez. O_2, an object-oriented data model. In *1988 Proc. SIGMOD Int. Conf. on Management of Data* (H. Boral and P. A. Larson, Eds), pp. 424–433. ACM Press (1988).

[11] S. Abiteboul, P. Fischer and H.-J. Schek (Eds). *Nested Relations and Complex Objects in Databases.* Number 361 in Lecture Notes in Computer Science. Springer-Verlag, New York (1989).

[12] D. Beech, A foundation for evolution from relational to object databases. In *Advances in Database Technology—EDBT'88,* Number 303 in *Lecture Notes in Computer Science* (J. W. Schmidt, S. Ceri and M. Missikoff, Eds), pp. 251–270. Springer-Verlag, New York (1988).

[13] The Committee for Advanced DBMS function. Third-generation database system manifesto. *SIGMOD Rec.* **19,** 31–44 (1990).

[14] J. Eder. Extending SQL with general transitive closure and extreme value selections. *IEEE Trans. Know. Data Engng* **2,** 381–391 (1990).

[15] N. Sarda. Extensions to SQL for historical databases. *IEEE Trans. Know. Data Engng* **2,** 220–230 (1990).

[16] L. Cardelli. A semantics of multiple inheritance. *Inform. Comput.* **76,** 138–164 (1988).

[17] Y. Lou and Z. M. Ozsoyoglu. LLO: an object-oriented deductive language with methods and method inheritance. In Ref. [42], pp. 198–207 (1991).

[18] R. Hull and R. King. Semantic database modelling: survey, applications, and research issues. *ACM Comput. Surv.* **19,** 201–260 (1987).

[19] C. Lécluse and P. Richard. Manipulation of structured values in object-oriented databases. In Ref. [45], pp. 113–121 (1989).

[20] A. Heuer. Equivalent schemes in semantic, nested relational, and relational database models. In *2nd Symp. Mathematical Fundamentals of Database System, Proc.*, Number 364 in *Lecture Notes in Computer Science*, pp. 237–253. Springer-Verlag (1989).

[21] R. Hull. Relative information capacity of simple relational schemata. *SIAM J. Comput.* **15,** 856–886 (1986).

[22] C. Date. A critique of the SQL database language. *ACM SIGMOD Rec.* **14,** 8–54 (1984).

[23] J. Van den Bussche. A formal basis for extending SQL to object-oriented databases. *Bull. EATCS* **40,** 207–216 (1990).

[24] M. Roth, H. Korth and D. Batory. SQL/NF: a query language for $-$1NF relational databases. *Inform. Syst.* **12,** 99–114 (1987).

[25] L. S. Colby. A recursive algebra for nested relations. *Inform. Syst.* **15,** 567–582 (1990).

[26] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inform. Syst.* **11,** 137–147 (1986).

[27] W. Kim. A model of queries for object-oriented databases. In P. Apers and G. Wiederhold, eds *Proc. Fifteenth Int. Conf. on Very Large Data Bases*, pp. 423–432. Morgan Kaufmann (1989).

[28] P. Lyngbaek. OSQL: A language for object databases. Technical Report DTD-91-4, HP Labs (1991).

[29] P.-A. Larson. The data model and query language of LauRel. *Database Engng* **11,** 138–145 (1988).

[30] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Inform. Syst.* **11,** 323–336 (1986).

[31] F. Bancilhon, S. Cluet and C. Delobel. Query languages for object-oriented database systems. In Ref. [45], pp. 122–138 (1989).

[32] S. Dar, N. H. Gehani and H. V. Jagadish. CQL++: a SQL for the Ode object-oriented DBMS. In *Advances in Database Technology—EDBT'92, Number* 580 *in Lecture Notes in Computer Science*. (A. Pirotte, C. Delobel and G. Gottlob, Eds) pp. 201–216. Springer-Verlag, New York (1992).
[33] G. Gardarin and P. Valduriez. ESQL2: an object-oriented SQL with F-Logic semantics. In *Proc. Seventh Int. Conf. on Data Engineering*. IEEE Computer Society Press (1992).
[34] M. Kifer, W. Kim and Y. Sagiv. Querying object-oriented databases. In *Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data*, Vol. 20:2 of *SIGMOD Record* (Stonebraker, Ed.), pp. 393–402. ACM Press (1992).
[35] F. Bancilhon, S. Cluet and C. Delobel. The O_2 query language syntax and semantics. Technical Report 45-90, GIP Altaïr (1990).
[36] S. Cluet, C. Deloble, C. Lécluse and P. Richard. RELOOP, an algebra based query language for an object-oriented database system. *Data Know. Engng* **5,** 333–352 (1990).
[37] Ontos Inc. *Ontos 2.1 SQL Guide* (1992).
[38] S. Abiteboul and A. Bonner. Objects and views. In Ref. [42], pp. 238–247 (1991).
[39] M. H. Scholl, C. Laasch and M. Tresch. Updatable views in object-oriented databases. In Ref. [47], pp. 189–207 (1991).
[40] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In PODS [46], pp. 379–393 (1989).
[41] A. Heuer and P. Sander. Classifying object-oriented query results in a class/type lattice. In *Proc. Third Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems*, Number 495 in *Lecture Notes in Computer Science*, (B. Thalheim, J. Demetrovics and H.-D. Gerhardt, Eds), pp. 14–28. Springer-Verlag, New York (1991).
[42] J. Clifford and R. King (Eds) *Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data*, Vol. 20:2 of *SIGMOD Record*. ACM Press (1991).
[43] W. Kim, J.-M. Nicolas and S. Nishio (Eds), *Proc. Ist Int. Conf. on Deductive and Object-Oriented Databases*. Elsevier, Amsterdam (1989).
[44] J. Clifford, B. Lindsay and D. Maier (Eds), *Proc. 1989 ACM SIGMOD Int. Conf. on the Management of Data*, Vol. 18:2 of *SIGMOD Record*. ACM Press (1989).
[45] R. Hull, R. Morrison and D. Stemple (Eds), *Proc. Second Int. Workshop on Database Programming Languages*. Morgan Kaufmann, Los Angeles (1989).
[46] *Proc. Eighth ACM Symp. on Principles of Database Systems*. ACM Press (1989).
[47] C. Delobel, M. Kifer and Y. Masunaga (Eds) *Deductive and Object-Oriented Databases*, Number 566 in *Lecture Notes in Computer Science*. Springer-Verlag (1991).
[48] J. Van den Bussche. Evaluation and optimization of complex object selections. In Ref. 47, pp. 226–243 (1991).