A Graph- and Object-Oriented Counterpart for SQL

Marc Andries* Jan Paredaens Jan Van den Bussche†

University of Antwerp (UIA), Dept. of Math. & Comp. Science
Universiteitsplein 1, B-2610 Antwerp, Belgium
E-mail: {andries,pareda,vdbuss}@ccu.uia.ac.be

Abstract

We introduce a graph- and object-oriented database query language, GOQL, inspired by a number of characteristics of SQL. GOQL describes a two-dimensional user-interface for constructing queries and database transformations. We show that GOQL compares favorably with SQL.

1 Introduction

During the past decade, database research has witnessed the rise of quite a number of database models, based on a variety of modeling paradigms, the most prominent of which are probably the object-oriented, the deductive and the functional approaches.

It was realized several years ago that in practice, database systems based on new paradigms will only be able to compete with commercial systems currently in use (especially relational DBMS) if the gap to be bridged by the user is kept sufficiently small. E.g., in [ABD+89] it is argued that future object-oriented DBMS must provide the functionality of ad hoc query languages, allowing users to pose simple queries to the database in a simple manner. A declarative nature, efficiency and application independence are mentioned as the three most prominent characteristics of such a facility. "Typical" relational queries are proposed as a yardstick for measuring the degree in which a newly proposed model comes up to this requirement. The authors of [CfADF90] even make a stronger statement by claiming that SQL [C+76] (which they refer to as "intergalactic dataspeak") has become so universally accepted that it must not be overlooked in the development of any novel OODBMS.

At the same time, the need for better and easier-to-use database end-user interfaces is widely being acknowledged. To this end, the two-dimensional nature of a computer screen should be fully exploited. The most natural starting-point for the development of a profound theoretical basis for such two-dimensional interfaces is the usage of *graphs* as the basic data type.

Graphs have indeed been an integral part of the database design process ever since the introduction of semantic and, more recently, object-oriented data models [Ban88, HK87]. Their usage in data manipulation languages, however, is more recent [AH87, CM90, GPG90, PPT91].

The first graphical database end-user interfaces were developed for the relational model (e.g., Query-By Example). Later on, many graphical database end-user interfaces were developed for the Entity-Relationship Model (as illustrated in the series of Proceedings of the International Conferences on Entity-Relationship Approach), as well as for more complex semantic and object-oriented database models [KKS88]. Although some proposals also deal with recursive data objects and queries [CMW87], most interfaces using graphs as their central tool are rather limited in expressive power.

*Supported by the I.W.O.N.L.
†Aspirant N.F.W.O.

Synthesizing the above two observations, we believe that currently a need is felt for SQL-like query languages for OODB's, as well as for graphical database user interfaces. The objective of this paper is therefore the development of a simple object-oriented query language (primarily oriented towards *retrieval* of information present in the database), inspired by the relational query language SQL on one hand, and by graph-oriented database models on the other hand. In the latter category of models, a database is conceptually looked upon as a graph, allowing the definition of a natural graphical syntax for database operations.

We focus on the following SQL-concepts:

1. building queries modularly (cfr. nesting of subqueries);
2. **Group By**- and **Having**-clauses;
3. join-select-project sequences;
4. aggregate functions.

From the area of object-oriented databases, we support the concepts of class hierarchy, encapsulation and message passing (or method calls).

We stress that this paper does not contain a fully fledged operational specification for a query language, but rather indicates some relevant concepts and techniques. We present a number of ideas on what a user-interface inspired by the ideas outlined above should look like, although the language we present may require some extensions and adaptations in order to be fit to serve as the basis of a complete end-user interface.

The rest of this paper is organized as follows. In Section 2 we introduce the Graph- and Object-oriented Query Language, using an elementary data model. In Section 3 we illustrate how the most prominent features of SQL may be expressed using GOQL.

2 The Graph- and Object-oriented Query Language

2.1 The Data Model

In this Section we introduce a very general model for object base schemes and instances, that will facilitate the development of the Graph- and Object-oriented Query Language. It relies only on the most basic object-oriented data modeling features of classes and methods, hence the concepts to be introduced in the sequel are largely system-independent and generally applicable.

Our notion of object base scheme is quite compatible to what is used in most object-oriented models introduced over the past few years in both the area of databases and programming languages [Bee90]. Basically, we look upon a scheme as a collection of class definitions, each of which consists of three components:

1. a class *name*;
2. a list of names of the class's *direct superclasses*;
3. a list of signatures of *methods*: such a signature consists of the method's name, a list of class names for the arguments (as well as a proper name for each argument), and the class name of the resulting objects.

Figure 1 shows the object-base scheme we will use as a running example throughout the remainder of this paper. To clarify the applied notations, let's look at the signature of the method "teaches" in the definition of class `Employee`, which can be used to test whether an employee teaches a given course.[1] In this signature, *course* is the proper name for the argument, while `Course` is the name of the class of objects for this argument (cfr. item 3 in the description of class definitions). Note also the application of multiple inheritance in the definition of class `Working_Student`, as well as its empty **Methods** section.

Since we focus on languages for data *retrieval*, we restrict ourselves to *side effect-free* methods. Although methods are the only means by which relationships between objects can be modeled, it is clear that attributes may be seen as a special kind of methods:

Single-valued Attributes: Methods having no arguments besides the receiver object (called *unary methods* in the sequel) express a single-valued property or attribute of that receiver;

[1] We assume that each scheme contains the classes `Boolean`, `String`, `Integer`,....

Class Person **is**
  Superclasses:
  Methods:
    name: → String
    street: → String
    town: → String

Class Department **is**
  Superclasses:
  Methods:
    name: → String
    building: → String
    president: → Employee

Class Employee **is**
  Superclasses: Person
  Methods:
    function: → String
    department: → Department
    salary: → Integer
    teaches: [*course:* Course] → Boolean

Class Course **is**
  Superclasses:
  Methods:
    name: → String
    department: → Department

Class Student **is**
  Superclasses: Person
  Methods:
    department: → Department
    takes: [*course:* Course] → Boolean
    score: [*course:* Course, *year:* Integer]
        → String

Class Working_Student **is**
  Superclasses: Student, Employee
  Methods:

Figure 1: An object-base scheme for Departments and Courses.

Set-valued Attributes: Methods with exactly one argument besides the receiver, resulting in an object of class **Boolean** (called *binary boolean-valued methods* in the sequel) express arbitrary object-to-object relationships, or, from the point of view of the receiver object, *set*-valued properties. For instance, the method "teaches" expresses a set-valued property of employees. In this respect, it will turn out convenient to make the following assumptions. For each such binary boolean-valued method (viewed as a set-valued attribute), we assume the existence of a number of corresponding unary methods (viewed as single-valued attributes), which express certain properties of the set-valued attribute under consideration. For example, for the set-valued attribute "teaches" of class **Employee**, also a single-valued attribute "$\text{count}_{\text{teaches}}$" is defined, returning a natural number. If in addition, the objects in the set are values on which simple arithmetic can be performed, we also assume the existence of methods for the typical SQL aggregate functions like **sum**, **max**, **min** and **average**.

Due to inheritance, methods of a class's superclass are applicable to all objects of that class. Concepts such as overriding and overloading are not considered in this paper, hence we assume that no method name may appear in both the definition of a class and one of its (direct or indirect) superclasses.

Object-base instances over a given object-base scheme are conceptually looked upon as *directed, labeled graphs.* Two kinds of nodes are distinguished within the instance graph. First, nodes representing *objects* are called *object-nodes*, and are labeled by the name of their class. Second, nodes representing method-calls are called *message-nodes*, and are labeled by the name of the method. Edges in the graph always go from message-nodes to object-nodes.

Figure 2 (which shows a partial instance over the scheme of Figure 1) illustrates the conventions used for the graphical representation of instance-graphs.

Object-nodes are graphically represented by rectangles. Objects representing *values* (like strings, numbers, bitmaps,...) may, in addition to their

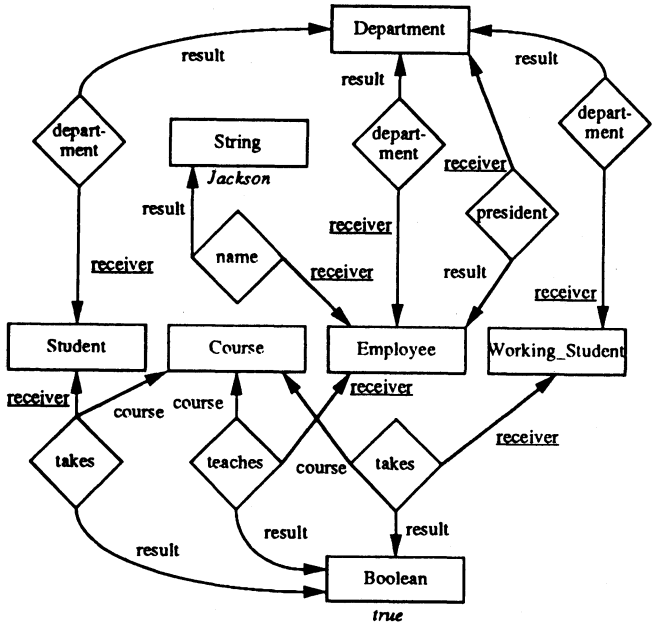

Figure 2: An object-base instance of Departments and Courses.

class name, be labeled by this value. Each value may occur only once in an instance. For instance, there is one node in the instance of Figure 2 representing the string "Jackson".

Message-nodes are represented using diamonds. The subgraph of the instance determined by such a diamond and its outgoing edges must correspond exactly to the signature of some method in the scheme. Consider e.g., the message represented in the bottom left corner of Figure 2, which represents a call to the method "takes", listed in the definition of class **Student**. The diamond representing that message has an outgoing edge, labeled *receiver*, to a node in the instance labeled **Student**. Besides, it has an outgoing edge labeled *result* to a node labeled **Boolean**, i.e., the class name of the resulting objects, as well as an outgoing edge labeled *course* to a node labeled **Course**. This means that the result of applying the method "takes" to the single student in the database, with the designated course in the database as argument, results in the boolean value *true*.

2.2 The Query Language

As already mentioned, we wish to offer the possibility to model queries in a modular fashion. Therefore, in general a GOQL-query consists of several *query steps*. Since graph-orientation is another major design goal, GOQL offers a natural graphical representation for the specification of such query steps.

Patterns

A first question that arises is how to specify those portions of the database to which a query step must be applied, or that must be examined by the application. For example, in the particular case of the relational database model with an SQL-like interface, the relevant database portions are specified using **From**- and **Where**-clauses, while the actual operations are expressed by, e.g., **Select** or **Delete** commands.

Our answer is to use *graph patterns* as a natural means for describing those parts of the database the user wants to access. Syntactically, a pattern is any subgraph of an instance. E.g., the pattern of Figure 3 describes those employees that teach a course to a student that studies at the same department they work for.

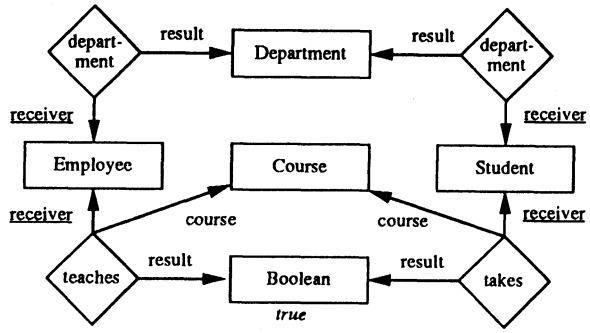

Figure 3: An example pattern.

Informally, when a query step is applied to some instance, it is applied to each occurrence or *matching* of the pattern in that instance. More formally, a matching is a mapping from the node-set of the pattern to the node-set of the instance, that satisfies two conditions. First, labels must be preserved, i.e., the image of a message-node must have the same label as that message-node, while the image of an object-node must be a member of the same class as that node, or it must be a member of a subclass. Second, if an edge exists between two nodes in the pattern, then an edge with the same label should exist between the images of these nodes in the instance.

The reader is invited to verify that there exist two matchings of the pattern of Figure 3 in the (partial) instance of Figure 2. Note that for one of the matchings, the Student-node in the pattern is mapped to a Working_Student-node in the instance.

However, Figure 3 also illustrates that, given the limited number of modeling primitives available (i.e., object- and message-nodes), patterns may rapidly become quite large and unsurveyable, even if simple configurations are being modeled.

Therefore, we now introduce a number of *syntactical shortcuts* for messages, which we believe are useful in many situations, as well as a simple extension of patterns to incorporate negation.

Single-valued Attributes in Patterns

In Section 2.1, it was already mentioned that unary methods express single-valued attributes. Since these are so often needed in patterns, we allow a call to such a method to be specified by an edge from the receiver node to the result node, labeled by the name of the method. This way, the pattern of Figure 4, top, describing persons and their names, can be replaced by the pattern of Figure 4, bottom.

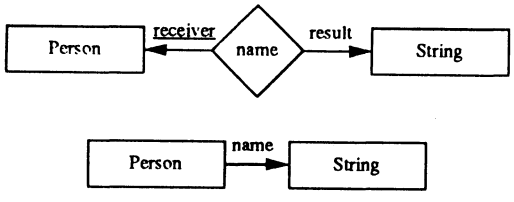

Figure 4: A shortcut for single-valued attributes.

An important feature of SQL from a practical point of view is the possibility to apply *aggregate-functions* to a set of values. Recalling the assumption made in Section 2.1 about aggregate functions of set-valued attributes, and using the shortcut for unary methods, calls to such functions may now be expressed in GOQL-patterns in a very elegant manner. For example, the pattern of Figure 5 describes those employees teaching exactly five courses.

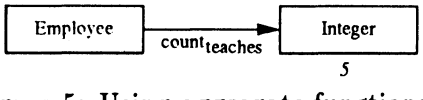

Figure 5: Using aggregate functions.

Set-valued Attributes in Patterns

In Section 2.1, it was already mentioned that binary boolean-valued methods express set-valued attributes. Therefore, in analogy to the shortcut for single-valued attributes, we allow calls to binary boolean-valued methods that evaluate to *true*, to be replaced by an edge from the receiver to the argument, labeled by the name of the method. To indicate that it is a shortcut for a set-valued attribute, this edge must have a *double* arrowhead. E.g., Figure 6 shows how the pattern of Figure 3 may be specified much more elegantly using the shortcuts for both single- and set-valued attributes.

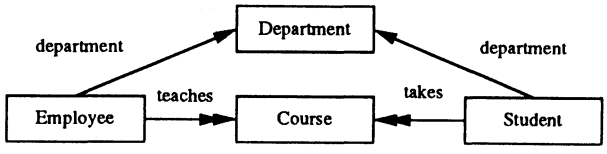

Figure 6: A pattern using shortcuts for attributes.

Another concept that can be expressed in GOQL patterns in an elegant manner using the shortcut for set-valued attributes, is that of comparison between values. For instance, in object-oriented programming [GR85], the comparison $x < y$ is seen as the boolean-valued result of sending a message $<$ to receiver x with argument y. Such a comparison may now be expressed in a GOQL-pattern using an edge between the objects representing the two values, labeled by the comparison operator under consideration. E.g., the pattern of Figure 7 determines those employees that earn more than employee Jackson.

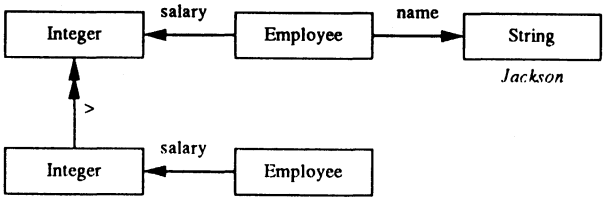

Figure 7: A shortcut for arithmetic comparison.

Negation in Patterns

Quite often, the specification of a method used in a pattern will involve some form of negation. A typical example is that of a method returning true or false, depending on whether or not a relationship between two objects does *not* hold. Although this

is a very simple condition on object-to-object relationships, we cannot use the shortcuts introduced earlier on in this Section because of the negation. This motivates the introduction of an extra feature for patterns, by allowing nodes and edges to be marked as being negated,[2] on the condition that the non-negated part still remains a syntactically correct pattern. The embeddings of a pattern using negation are those embeddings of the non-negated part, that cannot be "extended" to embeddings of the entire pattern.

E.g., the non-negated part of the pattern of Figure 8 (in which we use the shortcuts introduced above), has two embeddings in the instance of Figure 2, since employee Jackson teaches two courses. One of these embeddings however can be extended to an embedding of the entire pattern, since one of those courses is taken by a working student. Hence the complete pattern only matches the courses taught by employee Jackson, that are *not* taken by a working student.

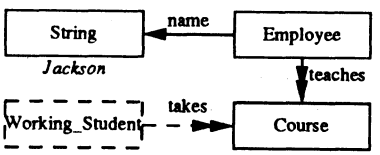

Figure 8: A shortcut for negation.

Operators

The first component being a pattern, the second component of a GOQL query step is an *operator*, which determines the actual operation to be applied to each of the matchings of the pattern of the query step. This operator corresponds roughly to the **Select**-clause of an SQL-query, or even to the **Update**-, **Modify**- or **Create View**-clauses of other SQL-statements. Since the major topic of this paper is the development of a query language, in this Section we only introduce two operators that are necessary and sufficient to express a broad variety of queries. This conjecture will be exhaustively illustrated in Section 3.

The main operator is CREATE SET. We first illustrate its semantics using the sample application of Figure 9. The effect of this operation is that first, for each employee who teaches a course, an object of class **Set of Students** is created as a (single-valued) "co-students"-attribute of that employee. Moreover, this **Set of Students**-object has a set-valued attribute "contains", by means of which it is linked to all students that take the considered course and study at the employee's department.

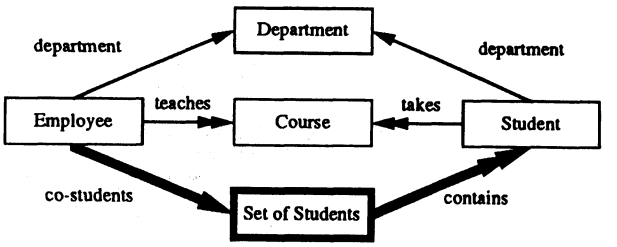

Figure 9: A sample application of CREATE SET.

More generally, a CREATE SET operation is applied to two object-nodes of the pattern, called its *owner*- and *member*-node (in the example these are the nodes with respective labels **Employee** and **Student**). The operation has three additional arguments, being a single-valued attribute-name A ("co-students" in the example), a class name **C** (**Set of Students** in the example) and a set-valued attribute-name S ("contains" in the example).[3] The effect of the CREATE SET operation is as follows. First, the matchings of the pattern are grouped according to the database object to which they map the owner node. Then, for each such object, an A-attribute of class **C** is created, if it does not exist already. Furthermore, the set of all database objects to which the member-node is mapped by a matching in the group corresponding to the considered owner-object, is added as the S-attribute of this owner-object.

We also allow certain arguments of CREATE SET to be omitted. First, the owner-node may be omitted. For example, the operation of Figure 10 groups all students taking the course on Databases as set-valued attributes of a newly created object of class **Set of Students**, unless an object of that class already exists. In the latter case, all **Student** objects in some matching of the pattern are added as "contains" attributes to each **Set of Students** object. This variation on CREATE SET allows sets

[2] Graphically, negated nodes and edges are represented using dashed lines.

[3] Naturally, the signature of these methods should not conflict with the original database scheme.

to be added to the instance, even if they are not a single-valued attribute of some other object.

If no member-node is specified, the database objects matched to the owner-node get a new A-attribute of class C (unless such an object already exists). This variation is useful for "tagging" objects that satisfy a certain condition (expressed by the pattern).

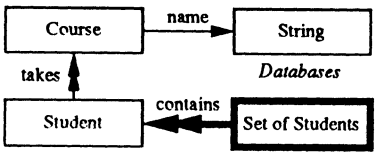

Figure 10: Grouping the database-students.

CREATE SET is the basic operator of GOQL. It allows the addition to the database of all kinds of derived information, by creating and manipulating sets. At some point, the result of the query must of course be presented to the user. In GOQL, this is done as shown in Figure 11. In general, PRINT may be applied to an arbitrary number of object-nodes in the pattern. Its effect is that for each matching of the pattern, some appropriate representation for the database objects to which these nodes are matched is presented to the user.

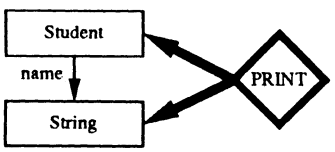

Figure 11: Showing students and their names.

We also allow several PRINT- and CREATE SET-operations to be specified together on a single pattern. This may render the formulation of certain queries in GOQL more concise. The semantics of such a "combined" operation is that first, the set of matchings of the pattern is determined, after which the operations may be applied to this set in any order. For example, the operation of Figure 12 associates to each department sets containing its students and courses. In most cases, the order of application is irrelevant.[4].

[4]The only case where a problem may arise is in the singular case where two CREATE SET-operations create objects of the same class, while exactly one of them has no member-node indicated

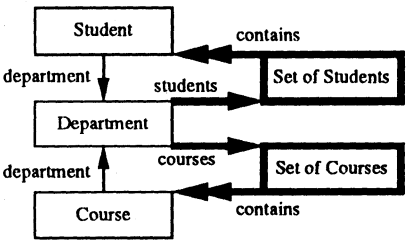

Figure 12: Multiple operations on one patte

This concludes the introduction of GOQL q steps. General GOQL queries are built by sir sequencing query steps, where the pattern of step may use classes and methods from the orig database scheme, as well as classes and met introduced by previous applications of CRE SET.

This close correspondence between patterns schemes may well be exploited by an actual interface in the way it lets the user specify que If patterns can only be built by constantly "r ring" to parts of the scheme, (syntactically) in rect queries cannot be formulated.

For example, object-nodes might be added pattern by picking class-names from a menu, v message-nodes are added by picking method-na from a menu corresponding to the class of a ignated object-node. An alternative is to let user copy and compose subgraphs from a gra cal representation of the object-base scheme, u the same conventions as introduced for insta and patterns [AGP+92]. A full treatment of techniques is outside the scope of this paper.

Consequently, graph-oriented object manip tion formalisms that are pattern-based allow *syntax-directed* way of working, in a much r natural way than text-based interfaces.

In order to be fit to serve as the basis for a plete end-user interface, GOQL requires a nur of extensions. By allowing operations to cal bitrary methods, the GOQL-model may easil enhanced to incorporate other database operat besides querying, such as updates, restructur view definitions,.... Besides an operator for ating sets, a facility for creating tuples of ob is also required for expressing arbitrary restru ings. In addition to our simple PRINT-opera other means may be provided for I/O.

3 GOQL compared to SQL

In this Section, we will show that GOQL is sufficiently powerful to express all prominent features of SQL. Since GOQL uses an object-oriented data model, while SQL uses the relational model, we use a straightforward relational representation of the object-base scheme of Figure 1, shown in Figure 13 (all attributes are of type String, except "year" and "salary" which are of type Integer).

Person	(name, street, town)
Student	(name, street, town, takes, department)
Working_Student	(name, street, town, takes, department, salary)
Employee	(name, street, town, salary, function, teaches, department)
Department	(name, building, president)
Course	(name, department)
Score	(receiver, course, year, result)

Figure 13: A relational scheme for the database of Departments and Courses

Just to give an initial example, consider Figure 14. The SQL query shown at the top of the figure returns the names of those students that got some score on the Databases course in a year where nobody got an A on that course. Below it is shown how this query is expressed in GOQL. In the remainder of this section, we will make clear that actually, SQL-queries can be translated to GOQL in a systematic manner.

Consider the SQL-query of Figure 15, top. It retrieves the names of those students that take a course, taught by some employee that lives in the same town as they do. This query illustrates how GOQL deals with the nested subqueries of SQL. The first operation, an application of CREATE SET, corresponds to the nested subquery.

In the next example, we illustrate the use of set-comparison operators. The SQL-query at the top of Figure 16 returns all pairs of employees in which the first employee teaches at least all the courses the second employee teaches. This example also illustrates how the *naming* of subqueries (by means of the class name `His Courses`), a feature not present

```
Select  name
From    Student, Score
Where course='Databases' and
      takes=course and
      receiver=name and
      year not in
      (Select year
      From   Score
      Where course='Databases'
            and result='A')
```

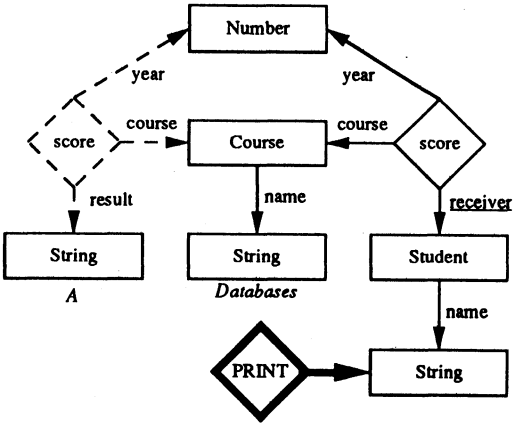

Figure 14: GOQL compared to SQL.

```
Select  name
From    Student S
Where takes in (Select  teaches
                From    Employee E
                Where S.town = E.town)
```

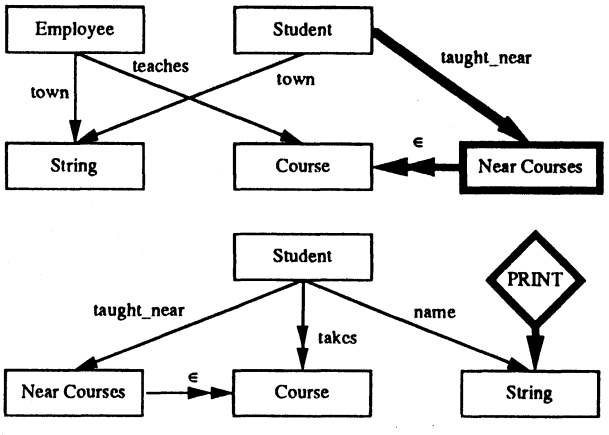

Figure 15: Nested subqueries in GOQL

```
Select  E1.name, E2.name
From    Employee E1, Employee E2
Where (Select  teaches
       From    Employee
       Where name = E1.name)
      contains
      (Select  teaches
       From    Employee
       Where name = E2.name)
```

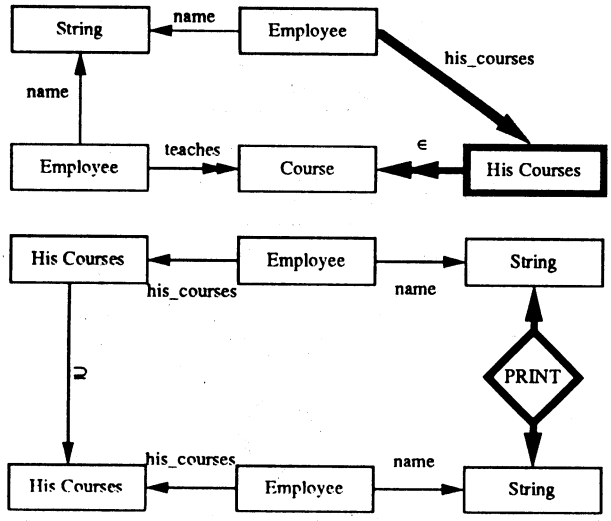

Figure 16: Set Comparison in GOQL

in SQL, allows a form of "code-reuse". In GOQL, we can namely make full usage of the syntactical similarity between the two subqueries of the SQL-query, which has to be specified only once.

In the next example, we illustrate how GOQL deals with the **Group by**- and **Having**-clauses as well as the aggregate-functions of SQL. [5] The SQL-query at the top of Figure 17 returns for each department its name and the average salary of its employees living in Antwerp, on the condition that the department employs at least hundred of these.

We now illustrate how the logical operands "and", "or" and "not" are dealt with. The SQL-query at the top of Figure 18 returns the pairs of students and employees that either live in a different town, or in the same town but in a different street. For simplicity, we apply PRINT to the students and employees, rather than to their names.

[5] For a correct treatment of duplicates in the computation of aggregate functions, the semantics of methods as well as of the procedure CREATE SET should be adapted to the multiset-semantics as used in SQL.

```
Select     department, avg(salary)
From       Employee
Where      town = "Antwerp"
Group By   department
Having     count(employee)>100
```

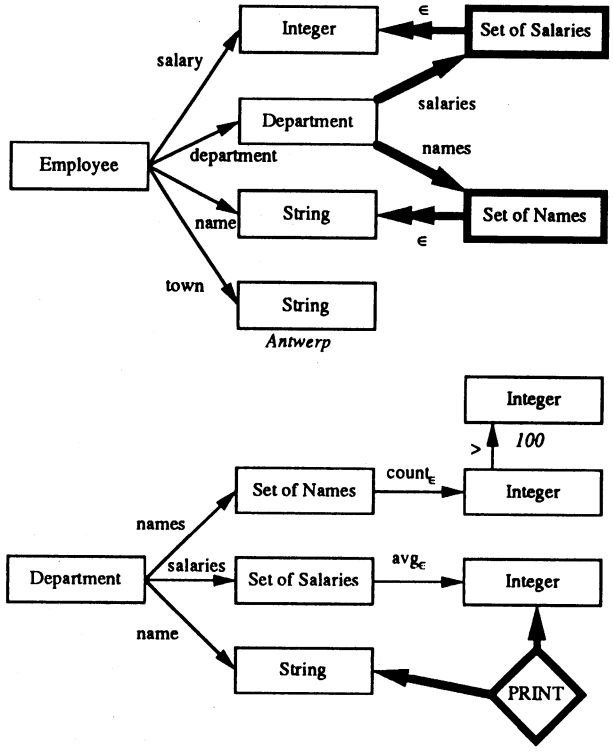

Figure 17: Grouping in GOQL

Note that conjunction can be expressed directly in a GOQL-pattern, while for negation we explicitly extended the definition of patterns. Furthermore, disjunction cannot be expressed in a single pattern at all. Similarly, universal quantification cannot be modeled directly, but has to be "simulated" using negation and existential quantification. The development of an appropriate graphical syntax for all these features is a topic of recent research [WMSB90].

In a final example, we illustrate how GOQL can express set difference. The SQL-query at the top of Figure 19 returns the names of students that are not employees. Note that we assume in this query that "name" is a key for persons.

Although it was our objective in this Section to show how SQL queries can be systematically trans-

Select E.name, S.name
From Employee E, Student S
Where not (S.town=E.town) **or**
((E.town=S.town) **and not**
(E.street=S.street))

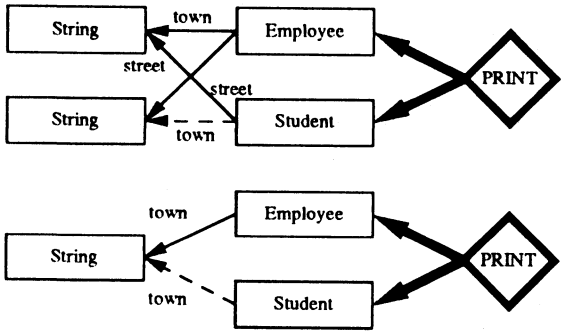

Figure 18: Logic in GOQL

lated to GOQL, GOQL can express certain queries in a more concise way than SQL. It is namely possible in some cases to translate a query which cannot be formulated in SQL without subqueries, into a GOQL query consisting of a single query step. The previous query (illustrating the translation of set difference) may be used to illustrate this claim.

Select name
From Student
Where name **not in** (**Select** name
From Employee)

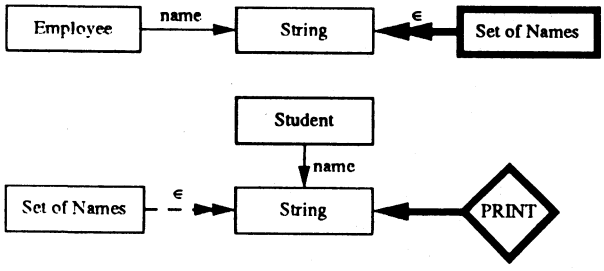

Figure 19: Set difference in GOQL

First, note that the one-step GOQL query of Figure 20 expresses exactly the same query as the one of Figure 19. On the other hand, set-difference is a *non-monotonous* operation, in the sense that it does not preserve the inclusion of relations. Since in SQL queries without subqueries, only monotonous queries may be expressed, the SQL-representation

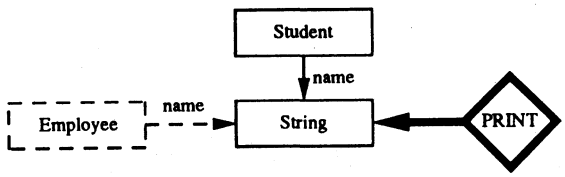

Figure 20: The operation of Figure 19 in a single GOQL query-step

of the considered query cannot be rewritten without using subqueries.

References

[ABD+89] M. Atkinson et al. The object-oriented database system manifesto. *Proc. 1st DOOD*, pp. 40–57.

[AGP+92] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. *Proc. EDBT'92* (to appear).

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *TODS*, 12(4):525–565.

[Ban88] F. Bancilhon. Object-oriented database systems. In *Proc. 7th PODS*, pp. 152–162.

[Bee90] C. Beeri. A formal approach to object-oriented databases. *DKE*, 5(4):353–382.

[C+76] D. Chamberlain et al. SEQUEL 2: A unified approach to data definition, manipulation and control. *IBM J. Res. Dev.*, 20(6):560–575.

[CfADF90] The Committee for Advanced DBMS Function. Third-generation database system manifesto. *SIGMOD Record*, 19(3):31–44.

[CM90] M. Consens and A. Mendelzon. GraphLog: a visual formalism for real life recursion. *Proc. 9th PODS*, pp. 404–416.

[CMW87] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. A graphical query language supporting recursion. *Proc. 1987 SIGMOD*, pp. 323–330.

[GPG90] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. *Proc. 1990 SIGMOD*, pages 417–424.

[GR85] A. Goldberg and D. Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1985.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260.

[KKS88] H. J. Kim, H. F. Korth, and A. Silberschatz. PICASSO: A graphical query language. *Software Practice and Experience*, 18(3):169–203.

[PPT91] P. Peelman, J. Paredaens, and L. Tanca. G-Log: A declarative graphical query language. *Proc. 2nd DOOD*, pp. 108–128.

[WMSB90] K.-Y. Whang et al. Supporting universal quantification in a two-dimensional database query language. *Proc. 6th Data Eng. Conf.*, pp. 68–75.