Object-Oriented Querying of Existing Relational Databases

Daniel A. Keim, Hans-Peter Kriegel, Andreas Miethsam

Institute for Computer Science, University of Munich
Leopoldstr. 11B, D-8000 Munich 40

Abstract: In this paper, we present algorithms which allow an object-oriented querying of existing relational databases. Our goal is to provide an improved query interface for relational systems with better query facilities than SQL. This seems to be very important since, in real world applications, relational systems are most commonly used and their dominance will remain in the near future. To overcome the drawbacks of relational systems, especially the poor query facilities of SQL, we propose a schema transformation and a query translation algorithm. The schema transformation algorithm uses additional semantic information to enhance the relational schema and transform it into a corresponding object-oriented schema. If the additional semantic information can be deducted from an underlying entity-relationship design schema, the schema transformation may be done fully automatically. To query the created object-oriented schema, we use the Structured Object Query Language (SOQL) which provides declarative query facilities on objects. SOQL queries using the created object-oriented schema are much shorter, easier to write and understand and more intuitive than corresponding SQL queries leading to an enhanced usability and an improved querying of the database. The query translation algorithm automatically translates SOQL queries into equivalent SQL queries for the original relational schema.

1 Introduction

Relational database systems are widely used in research and industry. For traditional application areas like accounting, reservation systems, etc., the relational data model seems to be adequate providing suitable modeling and performance characteristics. The main reasons for using the relational data model are: It is well known, easy to use and has a firm theoretical basis. The SQL query language, however, with its linear syntax was developed two decades ago and has not changed substantially since then. SQL has rather poor query facilities compared to the query facilities of today's object-oriented database systems. In spite of major advances in research, little has been done to improve the functionality and expressiveness of SQL. By defining the SQL2 standard [11] some of the deficiencies and inconsistencies of SQL have been removed but no major improvement of the query language has been achieved. A major problem is still the lack of an intuitive way to specify complex queries. Practical experiments with novice and experienced users show that essential and powerful concepts of SQL, such as nested queries or set operators are rarely used in a correct way [10] degrading SQL to a query language which is only useful for simple ad hoc queries. Additionally, the sometimes quite unnatural normalization of the relational data model and the missing semantic modeling capabilities make querying of relational databases even more difficult. The standardization of SQL3 [17] which shall be completed in 1996, the earliest, is aimed to improve the modelling capabilities and the query language by introducing object-oriented features. However, it is not clear how the additional features of SQL3 can be used in conjunction with existing databases.

From a practical point of view, it is very important to design query languages that allow novice and unexperienced users to query databases with little background other than some basic understanding of the schema and data. The design of graphical database interfaces is one approach to provide this kind of easy-to-use query interfaces [15]. While graphical user interfaces can greatly enhance the specification process, they can not overcome the limited capabilities of the relational model to express semantic aspects, i.e. relationships, structured entities and procedural aspects. A lot of research has been going on over the last decade to improve data models and query languages. As a result, major advances in database technology have been made, e.g. the object-oriented and extended relational database systems with their extended semantic modeling capabilities (e.g. [23], [13], [21], [16], [3]), advanced query languages (e.g. [8], [3]) and graphical user interfaces (e.g. [20], [1]). A problem, however, is the poor propagation of these systems in real world applications. Although commercial object-oriented database systems are available for some time they are rarely used in production environments. The main reason is the proliferation of relational systems. The effort and costs for migrating into a new system are very high since the application programs which have been implemented over the years and the training of users present high investments.

Our approach is a more pragmatic one and directed towards practical applications. Our starting point is the fact that, in the near future, we will be using relational systems for practical reasons; however, we need to improve the query specification process. It is possible to narrow the gap between the user's way of expressing queries and database manipulation languages like SQL without changing the system itself. Considering many examples, we found that using an object-oriented schema and query specification greatly enhances the readability and understandability of queries making it similar to the user's 'natural' view of the problem. Our idea is to automatically create an object-oriented schema from the relational one and to provide an object-oriented database query language which can be translated automatically into SQL. To query the created schema, we provide the Structured Object Query Language (SOQL), a declarative language for querying object-oriented databases. In SOQL, the user has full SQL-like access to the underlying relational database. Many object-oriented database query languages have been proposed in the literature [14], [7], [2], [22], [9], [3]. By introducing SOQL, we do not want to propose just another object-oriented query language. The main point in introducing SOQL is to define an object-oriented query language which is easy-to-use and allows to specify short and intuitively understandable queries but can be automatically translated into SQL.

At this point, we want to stress that the object-oriented schema we create is only a virtual one without having instances. The data itself completely remains in the relational system. Neither the schema transformation nor the query translation algorithm require any change to the data or the relational system. This is important since it will greatly enhance the practical applicability making our system useful for most areas where relational systems are used today.

The rest of the paper is organized as follows: Section 2 introduces the overall architecture of the system. Section 3 elaborates on the automatic transformation of relational schemas into object-oriented ones using meta information deducted from the underlying entity-relationship schema. In section 4, we introduce our Structured Object Query Language (SOQL) which provides declarative query facilities for the created object-oriented schema. In section 5, we will briefly describe the automatic query translation of SOQL into equivalent SQL queries. Section 6 summarizes our approach and points out some problems.

2 System Architecture

In this section, we want to introduce the overall architecture of our system. We designed our architecture to be used in real world environments and therefore, we had to

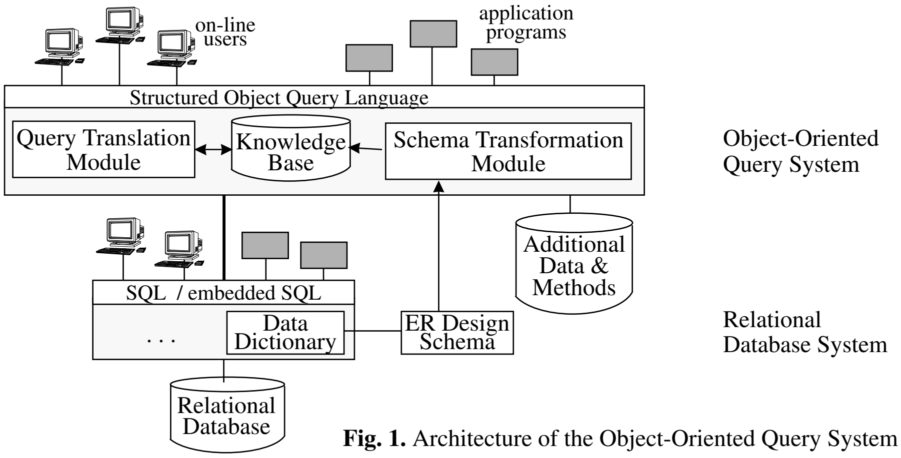

Fig. 1. Architecture of the Object-Oriented Query System

build it on top of existing relational systems like Oracle, Ingres, Sybase or others. It is important to note that, in general, such systems are used on-line with many application programs running permanently on a daily basis. In real world environments, it is important that changes of the architecture or the system do not have any impact on existing application programs because it is not feasible to rewrite them in the short- or mid-term range. Therefore, in our system we propose an additional layer which is built on top of the existing relational systems with their query language SQL (see figure 1). Our goal is to provide an advanced query interface for relational databases allowing an object-oriented querying of the database without migrating and transforming data or changing existing application programs. As shown in figure 1, in our system, pre-existing access structures remain unchanged while, at the same time, additional on-line and application program access to the database is provided by the object-oriented query system.

The main components of our system are the schema transformation module, the query translation module and the knowledge base. The schema transformation module is necessary to create an object-oriented schema from the relational schema. In general, it is not feasible to automatically create more structured and semantically richer object-oriented schemas from flat relational ones. Therefore, additional semantic information is needed, e.g. on tables implementing n-ary relationships (m:n, 1:n or 1:1), on connecting attributes implementing relationships between tables, on subtypes and so on. This additional semantic information is not modeled explicitly in the relational model but may be deducted from an underlying entity-relationship design schema. Consequently, in our schema transformation algorithm we use both, the relational schema and the entity-relationship (ER) design schema, to create the object-oriented schema. Meta information on the relational schema is usually stored in some kind of data dictionary, information about the ER schema is mostly available in the database design tool (see figure 1). Since format and access to both types of information may vary from one system to another, specific access procedures have to be implemented for the specific relational system and its design tool.

The knowledge base component is used to store all the additional semantic information deducted from the ER schema together with mapping information relating ER and relational model on one side with the object-oriented schema on the other side. The knowledge base is built during the semantic schema enrichment, the first step of the schema transformation algorithm, and provides the basis for an adequate schema transformation as well as for an automatic querying of the database based on the transformed schema. Furthermore, the knowledge base can be extended by the user to also allow schema extensions or changes and to define additional methods. User defined methods may be used in the same way as system defined methods allowing SOQL to be uniform and consistent even if extended by new classes and methods.

The query translation module uses the information stored in the knowledge base to translate SOQL queries based on the created object-oriented schema into equivalent SQL queries based on the original relational schema. As already indicated, SOQL allows to express any 'semantically meaningful' SQL query and the translation algorithm guarantees a fully automatic translation of such queries into SQL. Only if user-defined methods or additional classes are used, an SOQL query can not be translated directly into an SQL query. As will be described in section 5, the data needed to execute user-defined methods has to be selected iteratively from the relational database before such methods can be executed by our object-oriented query system.

3 Schema Enrichment and Transformation

In this section, we investigate how a relational database schema can be transformed into object-oriented class definitions. Usually a good object-oriented schema contains more semantics than a relational schema for the same application domain. If an automatic transformation process is aimed to produce adequate, well-structured object-oriented class definitions, more input than the pure relational schema is needed.

For illustrating the schema enrichment and transformation process and as a basis for the query examples in section 4, we will use the following example. Consider a relational database *FlightDB* containing information on passengers, departures, airlines, planes, planetypes and their relationships.

Passenger (<u>pid</u>: Integer; name: String; address: String)
Departure (<u>did</u>: Integer; start: Date; flight: Integer; airline-id: String; plane-id: Integer)
Pass_Dept (<u>did</u>: Integer; <u>pid</u>: Integer; booking: Date)
Airline (<u>airline-id</u>: String; name: String)
Plane (<u>serial-nr</u>: Integer; yr-built: Date; manufacturer: String; model: Integer)
Planetype (<u>model</u>: Integer; manufacturer: String; capacity: Integer; range: Integer)

To transform this database schema, we need additional semantic information, e.g. that *Pass_Dept* establishes an *m*:*n*-relationship between *Passenger* and *Departure*. Generally, we need additional semantic knowledge such as tables representing relationships (connecting tables), the type of the relationship (1:1, 1:n, n:m), attributes or groups of attributes representing foreign keys (connecting attributes), etc.

This additional semantic information is crucial for the schema transformation process to be able to replace connecting attributes and connecting tables by direct object references. Very often the domain of interest is formalized using an entity-relationship (ER) model [4]. The ER model contains the semantic information needed for our schema enrichment. If there is a formalized and standardized semantic design model together with an also standardized mapping which entity and which relationship lead to which table, a fully automatic schema enrichment is possible. If no standard ER model and no standardized mapping is available, support by the designer or administrator of the database will be necessary. In any case, part of the additional semantic information can be automatically deducted [6], [19], [18] and the user may be guided in the process of relating the ER design schema to the relational schema.

In the following, we assume that we are able to extract an ER model from the given relational schema with the following properties: Each entity E in the ER model corresponds to a table E in the relational schema, for each functional relationship R: E $\rightarrow$ F table E contains the (foreign) key of F, all other relationships R correspond to a table R connecting the respective 'entity' tables. This corresponds to the normal transformation when creating a relational schema from an ER design schema. In the following, we formally describe the transformation of the ER schema into an object-oriented schema:

1. for each entity E with attributes A_i of domain D_i, i=1,...,n and key K(E)
   $\Rightarrow$ **Class** E **with attributes** A_1:D_1; ...; A_n:D_n; **key is** (A_1, ..., A_m); **end**; is created.
2. for each functional relationship R: E $\rightarrow$ F
   $\Rightarrow$ class E is extended by a method R: $\rightarrow$ F, which applied to an object e of class E yields the corresponding object f of class F: e.R = f.
   $\Rightarrow$ class F is extended by a method R: $\rightarrow$ Set(E), which applied to an object f of class F yields the corresponding set of objects $\{e_1, ..., e_n\}$ of class E: f.R = $\{e_1, ..., e_n\}$.
3. for all other relationships R between entities E_1, ..., E_p, with possible relationship attributes A_k of domain D_k, k=1,...,q
   $\Rightarrow$ class E_i is extended by the following q+1 methods:
   - R: $\rightarrow$ Set(E_1 x ··· x E_{i-1} x E_{i+1} x ··· x E_n), which applied to an object e_i of class E_i yields the corresponding object tuples, e_i is in R-relationship with: e_i.R = $\{(\bar{e}_i) \mid$ R-relationship holds for $(e_1, ..., e_n)\}$, where $\bar{e}_i$ denotes the tuple $(e_1, ..., e_{i-1}, e_{i+1}, ..., e_n)$.
   - A_k: E_1 x ··· x E_{i-1} x E_{i+1} x ··· x $E_n \rightarrow D_k$, which applied to an object e_i of class E_i and object tuple $\bar{e}_i$ as parameter yields the k-th attribute $a_k \in D_k$ of this relationship: e_i.$A_k(\bar{e}_i) = a_k$.

   Example: Let p be an object of class Passenger and d_1, ..., d_n be its departures, where p booked departure d_2 at January 1st, 1993. Then p.departures = $\{d_1, ..., d_n\}$ and, p.booking(d_2) = '01/01/93' (see also class definitions below).

Together with the creation of the object oriented schema, mapping information is stored in the knowledge base, relating classes and tables which originate at the same entity. Furthermore, each method reflecting a relationship of the ER schema is related to the corresponding connecting attributes and, if existing, connecting tables of the relational schema. We establish the mapping at class creation time by automatically linking each new class definition to its corresponding relational table. Having the mapping information, we can determine which table corresponds to which class and whether a given class attribute has to be translated to a join on the relational side. This mapping is needed for the automatic translation of queries on the object-oriented schema.

Let us now consider the schema transformation of our example database. There we have functional relationships between Plane and Planetype and between Plane and Departure and a connecting table Pass_Dept connecting Passenger and Departure, which has an attribute specifying the date of booking. The schema transformation algorithm described above will produce the class definitions given in figure 2 which represent the semantically enriched object-oriented schema.

At this point, it should be mentioned that our schema transformation may not provide a perfect object-oriented schema. There are additional possibilities e.g. identifying subtype relationships within the relational schema [18], using the aggregation paradigm of the object-oriented system more extensively[5] and so on [12]. But as it will be shown in section 4 and section 5, the schema created by our schema transformation allows to state SOQL queries which are often significantly shorter and more intuitive than corresponding SQL queries using the original tables. At this point, let us emphasize that we only generate class definitions in the object-oriented database system, whereas the instances remain in the relational database. Thus, access operations to instances of object-oriented classes have to be translated into accesses to the corresponding relational tuples which will be described in section 5.

Class *Passenger* **with**
attributes *pid: Integer;*
name: String;
address: String; key is (pid);
methods *departures: → Set (Departure);*
booking: Departure → Date; **end;**

Class *Departure* **with**
attributes *did: Integer;*
start: Date;
flight: Integer; key is (did);
methods *airline: → Airline;*
plane: → Plane;
passengers: → Set (Passenger);
booking: Passenger → Date; **end;**

Class *Planetype* **with**
attributes *model: Integer;*
manufacturer: String;
capacity: Integer;
range: Integer; key is (model);
methods *planes: → Set (Plane);* **end;**

Class *Plane* **with**
attributes *serial-nr: Integer;*
yr-built: Date; key is (serial-nr);
methods *departures: → Set (Departure);*
planetype: → Planetype; **end;**

Class *Airline* **with**
attributes *airline-id: String;*
name: String; key is (airline-id);
methods *departures: → Set (Departure);* **end;**

Fig. 2. Object-Oriented Schema

4 Structured Object Query Language

In this section, we will give a short overview of our query language SOQL. SOQL is a declarative query language for querying the created object-oriented schema. It is similar to other declarative query languages for object-oriented database systems such as O_2SQL [3], OSQL [7] and Object SQL [9]. In addition to features available in these object-oriented database query languages, SOQL provides concepts which greatly enhance the query specification process making it more intuitive. The main point, however, in introducing SOQL is to present an object-oriented database query language which can be translated automatically into SQL (c.f. section 5).

4.1 The Query Language

As already indicated by the name, SOQL is similar to SQL. SOQL provides declarative query facilities for objects as SQL does for relations. The basic query format can be indicated by the following description

select {<range_var>{.<method>}* {.struct_expr}$^{0/1}$ }$^+$
for each {<classname>{.<method>}* <range_var>}$^+$
{ where <condition> }$^{0/1}$.

In the ‘select’ clause, the user has to specify the desired output of the query. According to the expression in the ‘select’ clause, automatically a new (temporary) object class is created. As a result of the query, all tuples fulfilling the condition are available as virtual instances of this class. The result is also available as a (nested) set and, therefore, can be directly used in subqueries. To allow an easier specification of queries with structured results, we introduce the notion of ‘structured expressions’. Structured expressions extend the select-clause by providing the possibility to define the structure of the desired result which is indicated by square brackets. As we will show later in the examples, structured expressions do not only help to structure the result but may also help to avoid joins. Since structured expressions are a unique feature of SOQL, we give the exact syntax definition in the following

struct_expr ::= [{<struct_expr>}$^+$] | [{<method>}{.<method>}* {.struct_expr}$^{0/1}$] |
[<range_var>{.<method>}* {.struct_expr}$^{0/1}$]

The ‘for each’ clause is similar to the ‘from’ clause in SQL. It is necessary to define and type the class variables used in a query. The ‘for each’ indicates that the condition is checked for each instance of the corresponding class and in the case, an instance fulfills the condition, the desired output is created. In the 'where' clause, a condition may be specified. The condition is an expression with result type 'Boolean'. All methods, including the created access methods to attributes, may be used in the condition as long as the result of the whole expression is of type Boolean. As already mentioned, the result of a subquery may also be considered as a set. Therefore, set operations can be used to specify nested SOQL queries.

As most object-oriented systems, our system provides a set of basic object classes (*Boolean, String, Numbers, Integer, Real* and the generic classes *Set* and *List*) together with a set of basic methods. Special methods are defined for Set(Numbers) including the aggregate operations *sum*, *avg*, *min*, *max* (Set(Numbers) → Numbers). As already mentioned, the user may extend the system provided set of methods by additional ones. Such user-defined methods may be used in the same way as system defined methods allowing SOQL to be uniform and consistent even if it is extended by new classes and methods. In the case of user-defined methods, however, there is no automatic translation to a single SQL query based on the underlying relational database (c.f. section 5).

As indicated in the query format definition, methods are applied to class or range variables using dot-notation. For convenience, the standard infix notation is allowed for the predefined methods of the basic classes. Chains of methods may be connected in dot-notation, which allows to directly access one object class from another one without explicitly joining them. It is some kind of schema navigation in the created object-oriented schema. An advantage of the dot-notation compared to database query languages like OSQL or O_2SQL is that our queries are structured in the way the user is thinking and, therefore, they are easier to write and understand. A problem of our approach, however, is that complex methods may have many arguments which may result in queries that are hard to read. In the case of creating the object-oriented from the relational schema, most access functions do not have any argument except their class and, therefore, the problem only occurs in the rare cases of methods deducted from relationship attributes or user-defined methods.

To further illustrate the possibilities of our query language, in the following we will give some examples for SOQL queries. We will show the advantages of SOQL over SQL by comparing SOQL queries based on the created object-oriented schema with equivalent SQL queries using the original schema. For the example queries, we use the transformed example database as presented in section 3. A simple query selecting all flight numbers with a list of the corresponding passenger names for the airline "Lufthansa" on the 02/18/93 would be expressed as

Example 1: select D.flight D.passengers.name for each Departure D
where D.start = "02/18/93" and D.airline.name = "Lufthansa"

Note, that the result of the query is of the complex type Set([Integer, Set(String)]). To store the result, a temporary class with two attributes of type Integer and Set(String) is created. The nested structure of the result is a consequence of using the generalization of the dot-notation to sets. Since 'D.passengers' provides a 'Set(Passenger)' for each departure, the method 'name' is not applicable since it is only defined for objects of class 'Passenger'. The generalization of the dot-notation to sets, however, allows methods which are defined for a class O to be also used with Set(O). As a consequence, the method name in our example can also be used with 'Set(Passenger)' providing a set of passenger names for each departure. 'D.passengers.name' is equivalent to $\{x.name \mid x \in D.passengers\}$. The generalization of the dot-notation to sets will be described formally in the next subsection. In the relational system, even for the simple query example 1, four tables need to be joined in order to execute the query. An equivalent SQL query is given as the result of the query translation algorithm in section 5.

Another simple query would be to find all passengers who have at least one flight together with a passenger named "Andy Meier". In SOQL, we can write

Example 2: select P for each Passenger P
where "Andy Meier" in P.departures.passengers.name

In this query, again we use the generalization of the dot-notation to sets. The result of P.departures. passengers.name is a set of sets of strings. The method 'in' with parameter "Andy Meier", however, requires a set of strings since it is only defined for O x Set(O) → Boolean and not for O x Set(Set(O)) → Boolean. According to the generalization of the dot-notation, we shift the method 'in' into the inner brackets until it is applicable for the first time. In the example, instead of ' "Andy Meier" in P.departures.passengers.name ', we execute { "Andy Meier" in {x.name | x ∈ d.passengers} | d ∈ P.departures} resulting in a set of booleans. Like in IRIS [7], sets of booleans in conditions are implicitly 'or'-connected providing true if at least one element is true.

While the SOQL query is still intuitive and easy to understand, corresponding SQL queries are quite difficult to read and to write. A corresponding SQL queries requires a join of at least four tables with the need to know the connecting tables and attributes

```
select distinct P.name, P.address from Passenger P, Pass_Dept PD
where P.pid = PD.pid and PD.did in
      select PD1.did from Passenger P1 Pass_Dept PD1
      where P1.pid = PD1.pid and P1.name = "Andy Meier"
```

The next query is an example of a nested query. If we want to select name and address of all passengers which have flown with all types of planes, we may use the query

Example 3: select P.name P.address for each Passenger P
where P.departures.plane.planetype contains (select PT for each Planetype PT)

This query may be expressed in SQL as follows

```
select distinct P.name, P.address
from Passenger P
where not exists
      (select * from Planetype PT
       where not exists
             (select * from Pass_Dept PD, Departure D, Plane PL
              where P.pid = PD.pid and PD.did = D.did and
              D.plane-id = PL.serial-nr and PL.model = PT.model))
```

Another interesting query is to determine the seat utilization of all "Lufthansa" flights. The following SOQL query provides the desired result

Example 4: select (D.plane.planetype.capacity - D.passengers.count) for each Departure D
where D.airline.name = "Lufthansa"

A corresponding SQL query is far more complicated. One possibility is

```
select D.dno, (PT.capacity - count(PD.pid))
from Departure D, Pass_Dept PD, Plane PL, Planetype PT, Airline A
where A.name = "Lufthansa" and A.aid = D.airline-id and D.did = PD.did
           and D.plane-id = PL.serial-nr and PL.model = PT.model
group by D.dno, PT.capacity
```

Note, that in the SQL query we have to select more information than actually required. We need the additional information to do the grouping which is only implicit in the SOQL query. In general, if the result for a query is a nested set with more than one nesting level, there is no one-to-one translation to an SQL query. Nested results may occur as answer for queries with structured expressions or queries where the generalization of the dot-notation is used more than once in a row.

Our last example is such an SOQL query with a nested structured expression. To select the names of all passengers who have Andy as part of their name together with all their flights as well as name and address of all co-passengers we can write in SOQL

Example 5: select P.[name, departures.[flight, date, passengers.[name, address]]]
for each Passenger P
where P.name.substring("Andy")

Since, in this case, the result has more than one nesting level, there is no possibility to express the query in SQL. As we will show in section 5, in such cases we translate the SOQL query into an SQL query which provides a superset of the data necessary to answer the query.

To summarize the advantages of SOQL over SQL: SOQL queries are much shorter, easier to write and understand and more intuitive than corresponding SQL queries. Since the created class definitions are more structured, in most cases, joins do not have to be specified explicitly and complex queries are avoided. Additionally, the results of SOQL queries can be arbitrarily structured and user-defined methods may be used like system-provided ones. In general, we believe that the created object-oriented schema together with the SOQL query language are closer to the users view of the application domain which leads to an enhanced usability and an improved querying of the database.

4.2 Semantic Issues

Before presenting the automatic query translation algorithm (c.f. section 5), in this subsection we first need to formally describe the semantics of special features of SOQL, particularly of the generalization of the dot-notation and of structured expressions.

The semantics of the 'select' clause is straightforward as long as only the system created access methods for attributes are used. For all other methods, we have to apply the method to all instances fulfilling the condition. More exactly, a query

select $a_1.op_1, ..., a_n.op_n$ for each ... with $op_i \in$ *Object-Class(a_i)* for i=1..n

results in a set of objects ($a_1.op_1, ..., a_n.op_n$). Even in the case of having chains of methods connected in dot-notation, there is no problem as long as the methods are defined for the class to which they are applied. We found, however, that this condition is quite restrictive for practical purposes and leads to queries which are more complex than necessary. Often, it seems to be intuitive to apply methods of a class O to sets of that class (Set(O)) or even to $Set^i(O)$. Therefore, we relax the condition by generalizing the dot-notation to sets. If, for example, a method is applied to objects of class $Set^n(O)$, but is not defined within this class, we try to apply the method to each member of the outmost set. If the method is not defined for $Set^{n-1}(O)$, we try to apply the method to each member of this set and so on until the method is defined for one level. Formally the generalization of the dot-notation is defined recursively

$$m(Set^i(O)) := \{m(obj) \mid obj \in Set^{i-1}(O)\}.$$

This step is repeated as long as the method m is not applicable to $Set^i(O)$. Using this recursive definition the nesting structure of the whole expression is preserved.

The semantics of structured expressions is that all attributes on the same nesting level are related to each other if possible. As we have shown in the previous subsection, structured expressions do not only help to intuitively specify structured results, but also to avoid complicated join conditions. More formally, the semantics of a structured expression can be described as follows: Let $O.[m_1, ..., m_n]$ be a structured expression. If there is no m_i which is applicable to O and O is an object of set type, we generate the set $\{obj.[m_1, ..., m_n] \mid obj \in O\}$. This step is repeated as long as obj itself is a set and no m_i is applicable. If an m_i is applicable at the level described in the above structured expression, the final result is $\{(obj.m_1, ..., obj.m_n) \mid obj \in O\}$. According to this definition, the result type for the query in example 5 can be described as { (String, { (Integer, Date, { (String, String)}) })}. The above definition for resolving structured expressions and method applications may be used for arbitrary structured expressions.

5 Translation of SOQL Queries

This section describes the translation of SOQL queries into SQL queries and the restructuring of the result according to the complex answer type given by the SOQL select clause. It is obvious that all queries expressed in SQL over the relational schema can also be expressed by an SOQL query over the created object oriented schema, since information is added during the schema enrichment and transformation process and SOQL has more expressive power than SQL. By providing a translation *t*, we show constructively how an SOQL query Q is translated into an SQL query S = *t* (Q). The result of S may be formatted by a function *f* into the desired answer format specified by Q, where *f* basically consists of sorting and projection operations.

The main task of *t* is to resolve chains of method applications by adequate joins and subqueries on the relational side and to correctly replace the SOQL condition part by equivalent SQL constructs. In the following, we describe the translation *t* of a given SOQL-statement Q into an SQL-statement S and illustrate this process using Example 1 from section 4.1. We assume that all class variables occurring in the 'for each' clause of the query and its subqueries have pairwise distinct names; otherwise, they will be consistently renamed. New variables introduced during the transformation are denoted by V_i.

1. First, the SOQL-statement Q is transformed into a nested set expression by evaluating the chains of method applications and structured expressions as described in section 4.2. The result is an equivalent (same result) specification of the query Q, with resolved dot generalizations and resolved structured expressions.

   $$\{(\text{D.flight, D.passengers.name}) \mid \text{D} \in \text{Departure} \wedge \text{D.start} = \text{“02/18/93”} \wedge \text{D.airline.name} = \text{“Lufthansa”}\} \equiv \{(\text{D.flight}, \{\text{V}_1\text{.name} \mid \text{V}_1 \in \text{D.passengers}\}) \mid \text{D} \in \text{Departure} \wedge \text{D.start} = \text{“02/18/93”} \wedge \exists\, \text{V}_2\text{: V}_2 = \text{D.airline} \wedge \text{V}_2\text{.name} = \text{“Lufthansa”}\}$$

2. The remaining object references are resolved in the following way: V_i *op* X.m $\equiv>$ $V_i \in$ Type(X.m) $\wedge$ join(X, V_i), where *op* stands for '$\in$' or '=' depending on whether X.m is set or single valued. In this step, join predicates join(X, V_i) are introduced with the intended meaning: join(X, V_i) = True, if there is an object reference from the current instance of X to V_i.

   $$\{(\text{D.flight}, \{\text{V}_1\text{.name} \mid \text{V}_1 \in \text{Passenger} \wedge \text{join(D, V}_1)\}) \mid \exists\, \text{V}_2\text{: D} \in \text{Departure} \wedge \text{D.start} = \text{“02/18/93”} \wedge \text{V}_2 \in \text{Airline} \wedge \text{join(D, V}_2) \wedge \text{V}_2\text{.name} = \text{“Lufthansa”}\}$$

3. Then, the nesting of result tuples is resolved by shifting set conditions onto the outer level and adding object identity / key information until the result tuple is flat. The structure of the result will be flattened by this transformation but can be easily reconstructed using the additional key attributes.

   $$\{(\text{D.flight, D.key, V}_1\text{.name}) \mid \exists\, \text{V}_2\text{: V}_1 \in \text{Passenger} \wedge \text{join(D, V}_1) \wedge \text{D} \in \text{Departure} \wedge \text{D.start} = \text{“02/18/93”} \wedge \text{V}_2 \in \text{Airline} \wedge \text{join(D, V}_2) \wedge \text{V}_2\text{.name} = \text{“Lufthansa”}\}$$

4. In the next step, we transform the above tuple-calculus-like expression into an SQL statement. The attributes to be specified in the 'select' clause can be directly taken from the result part of the expression. All parts 'X $\in$ Class' of the condition are transformed into the 'from' clause. If one of these variables is existentially quantified, the 'select' clause is extended by 'X.key' for all variables occurring in the 'select' clause

and the key word 'distinct' is added to remove duplicates which are not intended. The remaining condition part has to be transformed into a permissible SQL condition. The methods on set types such as 'el in set', 'isempty(set)', 'set1 contains set2' are replaced by computing these sets in a subquery and applying the SQL constructs 'el = some (select ...)', 'exists(select ...)', 'not exists(select ... where not exists (select ...))'.

```
select distinct  D.flight, D.key, V1.name, V1.key
from  Departure D, Passenger V1, Airline V2
where  join(D, V2) and join(D, V1) and D.start = "02/18/93" and V2.name = "Lufthansa"
```

5. The join predicates join(R, S) and key expressions S.key are replaced according to the mapping information.

```
select distinct  D.flight, D.did, V1.name, V1.pid
from  Departure D, Passenger V1, Airline V2, Pass_Dept V3
where  D.airline-id = V2.airline-id and D.did = V3.did and V3.pid = V1.pid and
       D.start = "02/18/93" and V2.name = "Lufthansa"
```

This is the final SQL statement to be executed on the relational database. The formatting function *f* has to sort the result by D.did and then eliminate D.did and V_1.pid.

Since SOQL has more expressive power than SQL, there are some cases where SOQL queries do not have corresponding SQL queries. Problems in the process of query translation may occur e.g. if set operations are used in conjunction with structured tuples or nested sets in the 'where' clause, if variables in the 'for each' clause range over nested sets and, as already mentioned, if user extensions (e.g. additional attributes or user-defined methods) are used in a query. In general, for such SOQL queries there is no translation to a single SQL query. Note, that the problems are only caused in cases where, in general, there is no corresponding SQL query. The details of the query translation algorithm are beyond the scope of this paper and will be presented in a future paper.

6 Summary and Conclusions

Relational database systems are widely used in research and industry. A major problem of relational systems are the poor query facilities of SQL. In this paper, we described a system which enhances the functionality and usability of existing relational databases and allows to query them like object-oriented databases. Using additional information deduced from the underlying ER schema, we automatically create a semantically enriched object-oriented schema together with the necessary mapping information relating object-oriented and relational schema. Our query language SOQL provides a uniform and convenient query interface to the database which, in addition, is easily extensible. The presented query translation algorithm is performing the automatic translation of SOQL queries into equivalent SQL queries for the original relational schema. We believe that our approach is simple, elegant and of high practical importance. We do not require any change to the relational system, the data or existing applications and therefore, systems like ours may be used in practice within a short period of time.

In our current implementation, we use the object-oriented database system O_2 as the basis for the additional layer. In O_2, we store the necessary semantic information as well as additional classes, methods and data. The created object-oriented schema is also available as O_2-schema. The implementation of the schema transformation and operation translation algorithms with complete support of user-defined methods and additional classes is currently on the way, but not yet finished. One open problem is the optimization of queries which involve user extensions to the schema or arbitrarily structured results. In such SOQL queries which have no one-to-one correspondence to an SQL query, the query optimization cannot be done on the relational side. Therefore, we have to optimize the query execution plan to reduce the amount of data which needs to be transferred between our and the relational system. Performance issues will be of high importance for such a system to be used in real world applications.

In our future work, we plan to extend the schema enrichment and query translation algorithms to cover the automatic detection and creation of subtype hierarchies or complex methods. We will further work on the optimization issue trying to provide an acceptable performance even in complicated cases. Finally, we intend to use our system as a basis for an advanced integration of relational systems into a heterogeneous multidatabase system and we plan to integrate the system itself into a network of interoperating databases.

References

[1] Agrawal R. et al.: *'OdeView: The Graphical Interface to Ode'*, Proc. ACM-SIGMOD Int. Conf. on Management of Data, Atlantic City, 1990.

[2] Alashqur A.M., Su S.Y., Lam H.: *'OQL: A Query Language for Manipulating Object-oriented Databases',* Proc. 5th Int. Conf. on Very Large Data Bases, Amsterdam, 1989, pp. 433-442.

[3] Bancilhon F., Delobel C, Kanellakis P. (editors): *'Building an Object-Oriented Database System - The Story of O_2'*, Morgan Kaufmann, San Mateo, CA, 1992.

[4] Chen P.P.-S.: *'The Entity-Relationship Model - Toward a Unified View of Data',* Proc. ACM Trans. on Database Systems, Vol. 1, No. 1, 1976.

[5] Castellanos M., Saltor F.: *'Semantic Enrichment of Database Schemas: An Object Oriented Approach',* Proc. 1st Int. Workshop on Interoperability in Multidatabase Systems, Kyoto, 1991, pp. 71-78.

[6] Davis, Arora: *'Converting a Relational Database Model into an Entity-Relationship Model',* Proc. 6th ER Conf., New York, 1987.

[7] Fishman D.H. et al: *'Overview of the Iris DBMS'*, chapter 10 in: *Object-Oriented Concepts, Databases and Applications* by Kim W. and Lochovsky F.H. (editors), ACM Press Frontier Series, Addison Wesley, Reading, MA, 1989, pp. 219-250.

[8] Haas L.M., Freytag J.C., Lohman G.M., Pirahesh H.: *'Extensible Query Processing in Starburst'*, Proc. ACM-SIGMOD Int. Conf. on Management of Data, 1989, pp. 377-388.

[9] Harris C., Duhl J.: *'Object SQL'*, chapter 11 in: *Object-Oriented Databases with Applications to CASE, Networks, and VLSI Design* by Gupta H. and Horowitz E., Prentice Hall, 1991, pp. 199-215.

[10] Hansen G.W., Hansen J.V.: *'Human Performance in Relational Algebra, Tuple Calculus and Domain Calculus'*, Int. Journal of Man-Machine Studies, Vol. 29, 1988, pp. 503-516.

[11] ISO/IEC: *'Database Language SQL'*, ISO/IEC 9075:1992 (German Standardization: DIN 66315).

[12] Kaul M., Drosten K., Neuhold E.J.: *'ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views',* Proc. IEEE Int. Conf. on Data Engineering, 1990.

[13] Kim W., Ballou N., Chou H.-T., Garza J.F., Woelk D.: *'Features of the ORION Object-Oriented Database System'*, chapter 11 in: *Object-Oriented Concepts, Databases and Applications* by Kim W. and Lochovsky F.H. (editors), ACM Press Frontier Series, Addison Wesley, MA, 1989, pp. 251-282

[14] Kim W.: *'A Model of Queries for Object-Oriented Databases'*, Proc. 5th Int. Conf. on Very Large Data Bases, Amsterdam, 1989, pp. 423-432.

[15] Keim D.A., Lum V.: *'Visual Query Specification in a Multimedia Database System'*, Proc. Conf. Visualization, CS Press, Los Alamitos, CA., 1992.

[16] Lohman G.M., Lindsay B., Pirahesh H., Schiefer K.B.: *'Extensions to Starburst: Objects, Types, Functions and Rules'*, Comm. of the ACM, Vol. 34, No. 10, 1991, pp. 94-109.

[17] Melton J. (editor): '*Database Language SQL (SQL3)'*, ISO/ANSI working draft, X3H2-92-055 DBL CNB-003, July 1992.

[18] Markowitz V., Makowsky J.: *'Identifying Extended Entity-Relationship Object Structures in Relational Schemas',* IEEE Tran. on Software Engineering, Vol. 16, No. 8, 1990.

[19] Navathe S., Awong: *'Abstracting Relational and Hierarchical Data with a Semantic Data Model',* Proc. 6th ER Conf., New York, 1987.

[20] Rogers T.R. , Cattell R.G.G.: *' Entity-Relationship Database User Interfaces'*, in: Readings in Database Systems, ed. M. Stonebraker, 1988.

[21] Rowe L.A., Stonebraker M.R.: *'The POSTGRES Data Model'*, in: *Readings in Object-Oriented Database Systems* by Zdonik S.B. and Maier D., Morgan Kaufmann, CA., 1990, pp. 461-473.

[22] Servio Logic Development Corporation: *'Gemstone V 2.0: Programming in OPAL'*, 1990.

[23] Shipman D.W.: *'The Functional Data Model and the Data Language DAPLEX'*, ACM Trans. on Database Systems, Vol. 6, 1981, pp. 140-173.