Database Transformation from Relational to Object-Oriented Database and Corresponding Query Translation[1]

Predrag Stanisic
Faculty of Computing Mathematics and Cybernetics
Moscow State University
Moscow, Russia
predrags@cs.msu.su

Abstract

The problem of database transformation from relational to object-oriented database occurs on using heterogeneous databases and when moving from a relational to an object-oriented database management system. The process of database transformation from one model to another can be divided into two phases: schema transformation from one model to another and data migration in accordance with schema transformation. Converting applications that work with a relational database into applications that work with an object-oriented database, the query translation problem occurs. In this paper we describe the main ideas of algorithms for implementation of all three phases, that enables automatization of the conversion processes.

1. Introduction

The necessity of conversion of an existing relational database into object-oriented one arises as a consequence of the appearance of generation of database management systems based on object-oriented model. The transformation problem also exists in heterogeneous systems, that contain various types of database management systems.

In this paper, the standard relational model is accepted and as an object-oriented model, we accept the core object-oriented model ([7]). The core model contains all basic object-oriented concepts. Many existing object-oriented data models expand the core model by adding some variations to concrete interpretations of core concepts. We accept the language shortly described in chapter 3 (see [5]) as the object-oriented query language because it well enough uses the path expression concept, which is an important advantage of the object-oriented model. In this paper the relational SQL-queries without path expressions are being transformed into object-oriented queries, which contain path expressions.

The paper is organized as follows. After introduction, in chapter 2, the process of schema transformation is described. The chapter 3 describes the process of data migration, and chapter 4 the process of translation of relational SQL-queries into equivalent object-oriented queries. At the end we give a short conclusion. In this paper we describe some of the main ideas of methods and illustrate them by the simple examples; all these methods are described in detail in [8,9,10,11].

2. Schema transformation

Schema transformation is a set of rules, which map concepts, existing in data model M_1, into the concepts of data model M_2 ([1]). For our task of transformation of a relational database schema into the object-oriented one, first of all we have to identify those concepts of object-oriented data model, which we want to obtain from relational database. Object-oriented data model describes not only the structure (static aspect), but the behavior (dynamical aspect) of objects from the real world, also. That means that object-oriented data model has, in this sense, better modeling abilities than relational data model. In paper [9] we describe an algorithm for obtaining the static aspect of object-oriented data model from relational schema, i.e. we are interested in the following object-oriented concepts: (a) object, (b) class, (c) inheritance and (d) references (complex attributes).

2.1. Description of the schema transformation procedure

For the transformation of relational database schema into object-oriented one, it is necessary to have information about foreign and candidate keys. In this paper we use the definition of candidate keys which point out not only on the

[1] This research is supported by Russian Foundation for Basic Research, grant ¹ 9607-89110

Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999

relation's primary key, but also on its arbitrary candidate key ([2]).

In our algorithm, relations are being mapped into classes and tuples from relations are being mapped into objects, which are instances of classes. Relationships between classes, i.e. inheritance and referencing, we identify by using foreign and candidate keys in the corresponding relations and using relationships between those keys. Relations can be directly mapped into classes but the semantics of the relational database will be lost. The goal of the paper [9] is to construct an algorithm that works automatically, i.e. that analyses relational database semantics and transforms it into object-oriented one, completely without any help from the user.

At the beginning, for each relation in database, a class with the same name has to be defined, containing all attributes from relation. In transformation process of a relational database schema to object-oriented one, some attributes, in this way defined classes, have to be removed and some of them have to be replaced with some other attributes. Information about that, which attributes are removed and which attributes are replaced with some other attributes, is necessary for the next phases of transformation of relational database into object-oriented one. That information has to be saved during schema transformation process ([9]).

Paper [9] describes seven characteristic cases of mutual relationship between foreign and candidate keys, which occur in schema transformation. They are described in an order in which they are being processed in the schema transformation algorithm. The established order of case processing resolves a large number of conflicts, which can occur. The order of processing of the cases is changed only in few situations. From these seven cases of mutual relationship of foreign and candidate keys, five cases are known in the literature ([3]), and two of them are proposed by author (see [9], 2.1.4 and 2.1.5). Differing from [3], where only an informal description is given, in [9] we analyze all cases in detail. In [9] we also give an analysis of conflicts that can occur, and the resolving procedures for each conflict are also described. These cases cover all candidate and foreign keys in relations of the relational database. Case 3 deals with candidate keys, which are, at the same time, foreign keys. Case 6 deals with candidate keys, which contain foreign keys, and Case 7 deals with all remaining foreign keys. Case 2 deals with relations, which have identical set of candidate keys. Cases 4 and 5 deal with some special cases (foreign key pointing on the same relation, to which the key belongs (Case 4), and two or more foreign keys which belong to one relation and which point on the another relation (Case 5)). Case 1 illustrates the use of internal data semantics.

It is necessary to make an additional remark for the Case 1. There are many different ways to make an internal analysis of data and method, described in [4,9], is only one of them. This case is used because it illustrates the possibilities of preliminary changes in relational database schema in such a way, that it contains more complete information about database semantics. The goal of this is to obtain an object-oriented database, which will better correspond to the initial relational database semantics. The following three examples illustrate some aspects of the process of schema transformation. In all of them, underlined attributes indicate candidate keys.

Example 1

Let be given the following relations: *student(id, name)* and *grad_student(gsid, thesis).* The domain of *id* and *gsid* attributes is *integer*, and the domain of the other attributes is *string*. In relation *grad_student,* attribute *gsid* is a foreign key, which points on candidate key *id* in relation *student*. Transformed object-oriented schema is:

```
class student              class grad_student
    id :integer;               inherits student;
    name :string;              thesis :string;
end;                       end;
```

In this example the inheritance relationship between two classes is being created. In such relationship, to each instance from subclass corresponds one instance from its superclass. These two instances have to be connected. Because of that, we assume that subclass contains one (implicit) attribute, whose name is *superclass_name-OID* for each one of its direct superclasses. Hence, domain of that attribute is the set of object identificators (OIDs) of instances of class *superclass_name*. For example, class *grad_student* contains attribute *student-OID*. This situation is being processed in Case 3 ([9]).

Example 2

Let be given the following three relations:

author(author#, name),
book(book#, publisher) and
author_book(authorNo,bookNo, date).

The domain of attributes *author#, authorNo, book#* and *bookNo* is *integer*, the domain of attributes *name* and *publisher* is *string* and the domain of attribute *date* is *datetime*. Attributes *authorNo* and *bookNo* are foreign keys from relation *author_book* which point out on candidate keys *author#* and *book#* of relations author and book, respectively. Transformed schema is:

```
class author               class book
    author# : integer          book# : integer;
    name : string;             publisher : string;
end;                       end;

class author_book
    author : ref author;
    book : ref book;
    date : datetime;
end;
```

Class *author_book* is created from relation *author_book* whose candidate key contains foreign keys, which point out on candidate keys in relations *author* and *book*. The attributes of the foreign keys are being transformed into references (complex attributes) which, in essence, are object identificators of instances of classes, which are created from relations on which those foreign keys were pointing. This case is examined in Case 6 of schema transformation process ([9]).

Example 3

Let be given relations *office*(*office#, building)* and *employee*(*name*, *adress*, *office_no)*. The domain of attributes *office#* and *office_no* is *integer*, the domain of attributes *name*, *adress* and *building* is *string*. Atribute *office_no* is a foreign key from relation *employee* to relation *office*. The transformed schema is:

```
class employee                  class office
    name :string;                   office# :integer;
    address :string;                building :string;
    emp_office: ref office;     end;
end;
```

The attributes of a foreign key, which is not contained in any candidate key, are being transformed into a reference (complex attribute) which is the object identificator of instances of the class, created from relation on which that foreign key was pointing to. This case is examined in Case 7 of schema transformation process ([9]).

3. Data migration in accordance with schema transformation

In the process of schema transformation, the relations are being mapped to classes. Tuples from those relations have to be mapped into objects, which are instances of corresponding classes. Tuples from some relations can be directly mapped into objects, which are instances of corresponding classes, but with some relations, this can not be done. The problem is that attributes of classes, which are created in the process of schema transformation of a relational database into an object-oriented database, can have domains, which can be different from domains in relational schema. This happens with attributes, that are object identificators of instances of the classes. By those attributes, as it was mentioned before, object-oriented concepts of inheritance and referencing (complex attributes) are being represented. In paper [10] we describe two algorithms for solution of data migration problem. Here we describe their main ideas.

3.1 The first data migration algorithm

This algorithm, in contrast to the second one (3.2), is based on the assumption that classes, in the object-oriented schema, contain exactly those attributes that were described in schema transformation process, i.e. there is no additional attributes that contain redundant information. This way, object-oriented schema is fully equivalent to initial relational database schema. This fact is a problem during data migration, because, in some situations, there is no possibility to uniquely identify the needed class instance.

We can notice in example 1 that instances of class *grad_student* can not be created before instances of the class *student* are created. The reason is that a unique candidate key *gsid* of the *grad_student* relation is not contained in the corresponding class *grad_student* and, because of that, we are not able to uniquely identify the wanted instance of this class if, for example, it is needed to change the value of *thesis* attribute. In example 2 we can also notice that the instances of the *author_book* class can not be created before instances of classes *author* and *book* are created. This is a direct consequence of the fact that candidate key (*authorNo, bookNo, date)* of relation *author_book*, contains attributes *authorNo* and *bookNo, which* are foreign keys pointing out on relations *author* and *book* and from which complex attributes *author* and *book* in class *author_book* are being created. On the other hand, in example 3 we can notice that instances of class *employee* can be created before instances of class *office* are created. This is a direct consequence of the fact that a single foreign key *office_no* in relation *employee* is not contained in any of candidate keys of that relation. Instances of classes *employee* can temporarily contain null values on attribute *emp_office* and that values can be updated later, i.e. instances can be later connected with the instances of class *office*.

3.1.1. Description of the first data migration algorithm

These examples show that it is not possible to construct instances of classes in arbitrary order. Instances can sometimes be constructed without all attributes, i.e. with null values on some of them. Values of those attributes can be updated later, i.e. when instances of classes, to which they refer, are constructed. For later update of class instances, we must have a possibility to uniquely identify an instance. Because of that, there should be present all attributes of that class, which are created from at least one candidate key of relation, from which that class is created. Only then, with corresponding object-oriented query, we have the possibility to identify uniquely the wanted instance. Attributes of candidate keys, as it can be concluded from introductory examples, are being transformed into OID attributes of superclasses, complex attributes (which are also OIDs) and simple attributes. We conclude that an instance of class can be created only if at least one of following three conditions is satisfied:

(a) the instances of at least one superclass of that class are created;

(b) there are created instances of all classes on which the complex attributes from that class reference and which are made from attributes of one candidate key of relation, from which that class is created;

(c) class contains all attributes of at least one candidate key of relation from which that class is created.

Data migration procedure is based on three sets of classes: *Start*, *Half-Finished* and *Finished*. If a class is contained in the set *Finished*, it means that all attributes of that class are present, i.e. all instances of that class are completely calculated. Of course, the data migration process is finished when all classes are in the set *Finished*. Class is contained in the set *Half-Finished* when all attributes of that class are not present yet, i.e. when some attributes of that class have null values on all instances of that class but instances are nevertheless created and they can be identified uniquely, because there are present all crucial attributes described in (a), (b) and (c). Set *Start* contains classes for which none of three conditions (a), (b) or (c) is satisfied, i.e. classes whose instances still can not be created. Data conversion process (in basic terms) can be described as follows.

First (Step 1), for each class, which is contained in the set *Start*, we have to check does that class satisfies at least one of the conditions (a), (b) or (c), or not. If it does, instances of that class can be created with all, in that moment, present attributes. Instances are created using INSERT operation on object-oriented system. Depending on the fact that, are all attributes present or only some of them, we add a class to the set *Finished* or *Half-Finished* and remove it from the *Start* set. Depending on which one of conditions (a), (b) and (c) that class satisfies, we have to define a way by which we will later identify instances of that class.

In Step 2, for each class that is contained in the set *Half-Finished*, we have to check which of formerly absent attributes are present now, i.e. which of them are object identificators of instances of classes which, in that moment, are contained in the set *Finished* or *Half-Finished*. All that attributes (if any) have to be updated, i.e. for each tuple in relation from which that class is created, has to be identified the corresponding instance of that class and attributes of that instance, that became present now, have to be calculated with corresponding UPDATE operation of object-oriented system. If all attributes of a class are present now, that class is being removed from the set *Half-Finished* and added to the set *Finished*. These two steps are being repeated until all classes are contained in the set *Finished*.

We point out that input of data migration algorithm is the initial relational database and data structures, in which the information on the process of schema transformation process is saved. The output is a sequence of INSERT and UPDATE operations, which can be directly executed by the object-oriented system.

3.1.2. Construction of INSERT and UPDATE expressions

Now we have to consider creation and updating of instances of the classes. As we said earlier, this can be done by INSERT and UPDATE operations on object-oriented system.

We assume the object-oriented query syntax is similar with relational SQL. Instead of formal definition of query language, we shall describe some of its main features in few examples. The query (in some hypothetical database) ″Find all graduated students whose average mark is above 9 and whose science adviser has an office in the building with number 3″ can be expressed as ‘*SELECT Student.Adviser.Name FROM Student WHERE (Student: Average>9).(Adviser:).(Office:).(Building: Number=3)*‘. Here, *Adviser* is a complex attribute of class *Student*. Example for insert operation is: ‘*INSERT INTO student(Name, Average, Adviser) VALUES (John, 8.7, SELECT Adviser FROM Adviser WHERE (Adviser: Name =Peter).(Degree =Ph.D.)’*. By this query, we create an instance of class *Student*, for a student with name John, whose average mark is 8.7 and whose adviser is a Ph.D. with name Peter. If we want to change advisor of student John, we use an update operation: ‘*UPDATE student(Adviser) VALUES (SELECT Adviser FROM Adviser WHERE (Adviser: Name =Jack)) WHERE Name =John’*. The part of this query ‘*WHERE Name =John’* is used to locate wanted instance. For a more formal definition of some aspects of this object-oriented query language see, for example, [5].

Now, we can proceed with the explanation of creation and updating of instances of the classes. To the execution of update operation on OID attribute in one class, which is made from candidate or foreign key of relation from which that class is created, or, in the other words, to assigning to that attribute a value of object identificator of instance of some other class, corresponds, in essence, an execution of join operation on candidate keys in relations (or on a foreign key in one relation and a candidate key in the other relation), from which those two classes are created. But, the attributes of candidate key in second relation may not be present in definition of class which is made from that relation. Their role can be played by attributes from some of superclasses of that class or attributes of class to which corresponding attribute refer. If we know values of those attributes from first relation, by equalizing these attributes and attributes from the second class, we can identify corresponding instance from the second class, i.e. we can find its OID and assign that OID to the complex attribute of the first class. Process of getting information about which attributes correspond to each other, goes together with data migration process. That information has to be obtained for all candidate keys and it is saved in so called *query-sets* [10], which we use for construction of INSERT an UPDATE expressions. In this paper, due to space limit, we do not formally describe this data migration algorithm. Instead, we consider a simple example of its work.

3.1.3 Example of the first data migration algorithm

Let be given the relational database showed on Fig. 1.

sport_car

sa_id	*Sa_tires*
1	1
3	2

car

id	*Name*
1	Jaguar
2	Ford
3	Ferari

sport_tires

st_no
1
2

tires

no	*manufacturer*	*t_president*
1	bridgestone	4
2	good year	3
3	tiger	2

driving

d_driver	*d_car*
123	1
234	3

race_driver

rd_id	*rd_name*
123	hill
234	prost
345	alesi

president

p_no	*name*	*p_car*
2	mark	1
3	john	1
4	peter	2

Figure 1. Sample relational database

Candidate keys are the first attributes from the left in all relations, except in relation *driving* where the candidate key is pair *(d_driver, d_car).*
Foreign keys are *sport_car.sa_id Ì car.id, sport_car.sa_tires Ì sport_tires.st_No, sport_tires.st_No Ì tires.No, tires.t_president Ì president.p_No, president.p_car Ì car.id, driving.d_driver Ì race_driver.rd_id* and *driving.d_car Ì car.id* .

Transformed schema is:

```
class sport_car                    class car
    inherits car;                      id : integer;
    racing_tires: ref sport_tires;     name : string;
end;                               end;

class sport_tires          class tires
    inherits tires;            no : integer;
end;                           manufacturer :string;
                               director :ref president;
                           end;

class driving                  class race_driver
     driver : ref race_driver;     rd_id :integer;
     vehicle: ref car;             rd_name: string;
end;                           end;

class president
     p_no  : integer;
     name  : string;
     director_car : ref car;
end;
```

We assume that the algorithm investigates relations/classes in above order. The algorithm creates the following queries:

```
INSERT INTO car(id, name) VALUES (1, jaguar);
INSERT INTO car(id, name) VALUES (2, ford);
INSERT INTO car(id, name) VALUES (3, ferrari);
INSERT INTO tires(no, manufacturer, director) VALUES (1,
Bridgestone, NULL);
INSERT INTO tires(no, manufacturer, director) VALUES (2,
good year, NULL);
INSERT INTO tires(no, manufacturer, director) VALUES (3,
tiger, NULL);
INSERT INTO race_driver(rd_id, rd_name) VALUES (123,
hill);
INSERT INTO race_driver(rd_id, rd_name) VALUES (234,
prost);
INSERT INTO race_driver(rd_id, rd_name) VALUES (345,
alesi);
INSERT INTO president(p_no, name, director_car) VALUES
(2, mark, SELECT car FROM car WHERE (car: id=1) );
INSERT INTO president(p_no, name, director_car) VALUES
(3, john, SELECT car FROM car
          WHERE (car: id=1) );
INSERT INTO president(p_no, name, director_car) VALUES
(4, peter, SELECT car FROM car
          WHERE (car: id=2) );
UPDATE tires
   SET director= (SELECT president FROM president
          WHERE (president: id=4)
WHERE tires.no=1;
UPDATE tires
   SET director=(SELECT president FROM president
          WHERE (president: id=3) )
WHERE tires.no=2;
UPDATE tires
   SET director=(SELECT president FROM president
          WHERE (president: id=2) )
WHERE tires.no=3;
INSERT INTO sport_car(car-OID, racing_tires) VALUES
(SELECT car FROM car
          WHERE (car: id=1), NULL);
INSERT INTO sport_car(car-OID, racing_tires) VALUES
(SELECT car FROM car
          WHERE (car: id=3), NULL );
INSERT INTO sport_tires(tires-OID) VALUES
  (SELECT tires FROM tires WHERE (tires: no=1) );
INSERT INTO sport_tires(tires-OID) VALUES
  (SELECT tires FROM tires WHERE (tires: no=2) );
INSERT INTO driving(driver, vehicle) VALUES
```

```
        ( SELECT race_driver FROM race_driver
                    WHERE (race_driver: rd_id=123 ),
         SELECT car FROM car WHERE (car: id=1) );
INSERT INTO driving(driver, vehicle) VALUES
        ( SELECT race_driver FROM race_driver
                    WHERE (race_driver: rd_id=234 ),
         SELECT car FROM car WHERE (car: id=3) );
UPDATE  sport_car
    SET racing_tires= (SELECT sport_tires
          FROM sport_tires WHERE (sport_tires: no=1))
 WHERE id=1;
 UPDATE  sport_car
     SET racing_tires=(SELECT sport_tires
          FROM sport_tires WHERE (sport_tires: no=2))
 WHERE id=3;
```

3.2 The second data migration algorithm

This algorithm is based on the assumption that each class in object-oriented schema contains not only those attributes, which are described in schema transformation process, but also all the other attributes from initial relation in the relational database, from which that class is created in schema transformation process. Those are the attributes, which are deleted from the class declarations in the end of the schema transformation process.

The main idea of this algorithm is to add to the class description some redundant attributes, which contain all necessary information for fast unique identification of instances of that class. After that, connection of instances of this class with instances of other classes can be made with the help of these attributes. First, all instances of all classes are being created, and after that instances are being connected. When an instance of some class is being created, the object-oriented database management system assigns to it the unique OID. Irrelevant of the situations, described earlier (3.1), every instance can always be identified here, because it now contains all necessary attributes from corresponding relation. In this case, instances can be connected to each other in arbitrary order, i.e. the order of their creation is not important. The algorithm can be described as follows.

First, at Step 1, using appropriate expression all attributes existing in the corresponding relation *r*, and which do not exist in the definition of the class $\tilde{N}$ are being added to the definition of class $\tilde{N}$, for each class *C* in object-oriented schema. Syntax of that expression can be, for example, *ALTER CLASS* $\tilde{N}_i$ *ADD ...* Attributes which are being added are those that were contained in foreign or candidate keys which created inheritance relationship between two classes.

Further, at Step 2, all instances of all classes are being created. In essence, at this step, tuples from relational database are directly being copied into class instances in the object-oriented database.

At Step 3 corresponding class instances are being connected to each other, i.e. using UPDATE expressions of the object-oriented query language, values to OID-attributes are being assigned. Using only one UPDATE operation all instances of one class can be connected to all necessary instances of other classes.

Finally, at Step 4 all attributes that were added in Step 1, are being deleted, using appropriate expression (for example, DROP).

3.2.1. Example of the second data migration algorithm

Let's consider an example of execution of this algorithm (we use the database from example in 3.1.3, Fig.1).

At Step 1 the following attributes are being added to the object-oriented schema:

sport_car.sa_id, sport_car.sa_tires, sport_tires.st_no, tires.t_president, president.p_car, driving.d_driver and driving.d_car.

At Step 2 the following expressions are being created:

```
INSERT INTO sport_car(sa_id, sa_tires, car-OID,
racing_tires) VALUES (1, 1, NULL, NULL);
INSERT INTO sport_car(sa_id, sa_tires, car-OID,
racing_tires) VALUES (3, 2, NULL, NULL);
INSERT INTO car(id, name) VALUES (1, jaguar);
INSERT INTO car(id, name) VALUES (2, ford);
INSERT INTO car(id, name) VALUES (3, ferrari);
INSERT INTO sport_tires(st_no, tires-OID) VALUES (1,
NULL);
INSERT INTO sport_tires(st_no, tires-OID) VALUES (2,
NULL);
INSERT INTO tires(no, manufacturer, t_president, director)
VALUES (1, bridgestone, 4, NULL);
INSERT INTO tires (no, manufacturer, t_president, director)
VALUES (2, good year, 3, NULL);
INSERT INTO tires (no, manufacturer, t_president, director)
VALUES (3, tiger, 2, NULL);
INSERT INTO driving (d_driver, d_car, driver, vehicle)
VALUES (123, 1, NULL, NULL);
INSERT INTO driving (d_driver, d_car, driver, vehicle)
VALUES (234, 3, NULL, NULL);
INSERT INTO race_driver(rd_id, rd_name) VALUES (123,
hill);
INSERT INTO race_driver(rd_id, rd_name) VALUES
(234, prost);
INSERT INTO race_driver(rd_id, rd_name) VALUES (345,
alesi);
INSERT INTO president(p_no, name, p_car, director_car)
VALUES (2, mark, 1, NULL);
INSERT INTO president(p_no, name, p_car, director_car)
VALUES (3, john, 1, NULL);
INSERT INTO president(p_no, name, p_car, director_car)
VALUES (4, peter, 2, NULL);
```

At Step 3 the following expressions are being created:

```
UPDATE sport_car
```

```
    SET car-OID=(SELECT car FROM car
                 WHERE (car: id=sport_car.sa_id)),
    SET racing_tires=( SELECT sport_tires FROM
                 sport_tires WHERE
              (sport_tires: st_no= sport_car.sa_tires));
UPDATE sport_tires
    SET tires-OID= (SELECT tires FROM tires
          WHERE (tires: no=sport_tires.st_no) );
UPDATE tires
    SET director= (SELECT president FROM president
          WHERE (president:p_no= tires.t_president) );
UPDATE driving
  SET driver=( SELECT race_driver FROM race_driver
       WHERE (race_driver: rd_id= driving.d_driver )),
  SET vehicle=(SELECT car FROM car
                 WHERE (car: id= driving.d_car) ) ;
UPDATE president
   SET director_car= (SELECT car FROM car
        WHERE (car: id= president.p_car) );
```

At Step 4, attributes, which were added in Step 1, are being deleted.

It is obvious, that this algorithm is optimal in terms of speed. On the one hand, one INSERT operation creates each instance directly from appropriate relation tuple, and, on the other hand, all instances of some class are being connected to all necessary instances of the other classes using only one UPDATE expression.

In an object-oriented database, concepts of complex attributes and inheritance are represented by OID-attributes. In relational database those concepts are represented by foreign and candidate keys, which can contain a large number of attributes. Because of that, object-oriented database, typically, has smaller size than the relational database. A disadvantage of the presented algorithm is the creation of the large amount of redundant information, which in extreme situations can be approximately near the size of the relational database. It can be a disadvantage when using very large databases.

It can be concluded that the first algorithm is optimal in terms of memory use (i.e., the size of database) and the second one is optimal in terms of speed. The advantage of the second algorithm is in its relative simplicity.

4. Translation of relational queries to equivalent object-oriented queries

We describe here a method for translation of SELECT-queries into corresponding object-oriented queries. We assume that both the relational and object-oriented queries are in SELECT-FROM-WHERE format. SELECT-clause of query specifies the names of relations/classes attributes, which are the result of query execution. FROM-clause of query specifies relation tuple variables (RTV) in case of relational query, and class instance variables (CIV) in case of object-oriented query. With the help of R(RTV) (C(CIV)) we shall designate the relation (class), for which RTV (CIV) is defined. WHERE-clause of query specifies a qualifying condition of query, i.e. condition, which have to be satisfied by current values of RTV (CIV) in order to include these values into the result. WHERE-clause of object-oriented query can also contain the *path expressions*. We assume that WHERE-clause of relational SQL query is given in conjunctive form. Without loss of generality, we assume that relational queries are not nested since nested relational queries can always be translated to equivalent unnested queries ([6]). Since relational WHERE predicate has conjunctive form, the translated object-oriented predicate is also in conjunctive form. The major problem in translation of a relational SQL query to equivalent object-oriented query, is the WHERE-clause translation, because there is a drastic difference between relational predicates and object-oriented predicates. The relational predicates, which consist of joins and selections and which do not contain path expressions, are being translated into object-oriented predicates, which contain path expressions.

Translation of the WHERE-predicate is being carried in three steps. First, a relational predicate graph is being created from the relational predicate. At second step, this graph is being converted to the corresponding object-oriented predicate graph. Finally, the object-oriented predicate is being obtained from the object-oriented predicate graph. Some ideas of this process, with significant modifications, are taken from [5]. In difference to [5], during the translation of the relational WHERE-predicate, the SELECT and FROM-clauses of query are being translated, also.

4.1. Construction of relational predicate graph from WHERE-clause of SQL query

For relational predicate RW in WHERE-clause of relational SQL-query we define its graph: G(RW) = (RV, RE), where each node from the set of nodes RV represents one RTV, which appears in RW-predicate (i.e. it also appears in FROM-clause of SQL-query), and each edge between two nodes, which is in RE, represents some join predicate between two RTV in RW. Each node is annotated by all selection predicates of corresponding RTV. Each edge between two nodes, for example, edge between RTV_1 and RTV_2, is annotated by a join predicate of the form '$RTV_1 \cdot A_1$ *comparator* $RTV_2 \cdot A_2$', where A_1 and A_2 are attributes of the relations $r_1=R(RTV_1)$ and $r_2=R(RTV_2)$. The relational predicate graph of WHERE-clause of SQL-query contains the same information as the predicate itself.

Example

Let be given the query: "Find the average mark for all graduated students, which passed a math exam at June, 15." Corresponding SQL-query is:

```
SELECT student.average
FROM exam, graduated_student, student
WHERE (student.student_id=graduated_student.student_id)
 AND (student.department=graduated_student.department)
    AND (exam.student_no=graduated_student.student_id)
```

AND (exam.department=graduated_student.department)
AND (exam.date=15 jun)
AND (exam.course_name=mathematics)

It is assumed here, that relation *graduated_student* and relation *student* are being joined by candidate keys, and relation *exam* is being joined with the relation *graduated_student* using a foreign key. It is also assumed, that attribute *course_name* has been deleted from class *exam* during schema transformation process and that it is now belonging to the class *course*. From WHERE-predicate of this query we obtain the graph shown in Fig. 2.

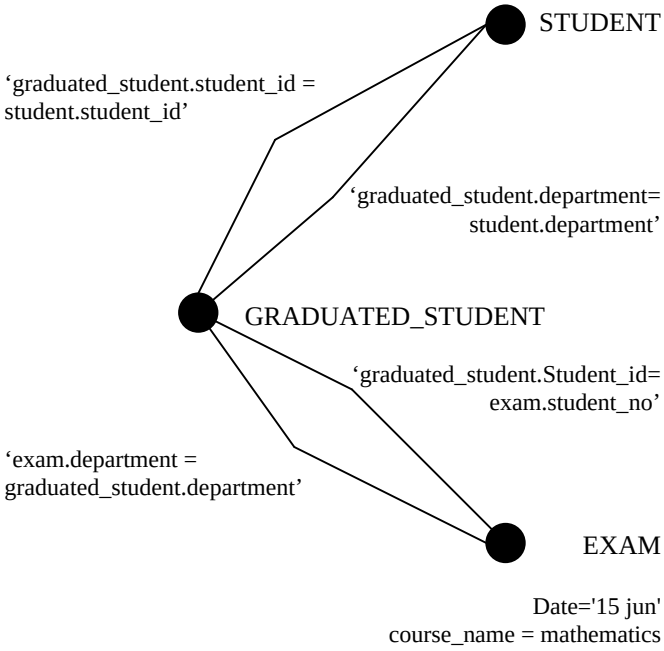


Figure 2. Example of relational predicate graph

4.2. Construction of the object-oriented predicate graph from the relational predicate graph

Let RW be a WHERE-clause of relational SQL-query of the relational schema. The graph of this relational predicate G(RW)=(RV, RE) is being transformed into annotated graph OG(RW)=(OV, OE_1, OE_2), where OV is a set of nodes, OE_1 is a set of directed edges, and OE_2 is a set of undirected edges. Each node from OV corresponds to one class instance variable (CIV), which corresponds to the some class in the object-oriented schema.

There are three kinds of edges in the object-oriented predicate graph: *directed edge with annotation*, *directed edge without annotation*, and *undirected edges with annotation*. A directed edge from CIV_1 to CIV_2 with annotation *ANNOTATION* corresponds to implicit join of $C(CIV_1)$ and $C(CIV_2)$ classes by complex attribute named *ANNOTATION* in $C(CIV_1)$ class, whose domain is class $C(CIV_2)$. A directed edge from CIV_1 to CIV_2 without annotation corresponds to implicit join of an instance of class $C(CIV_1)$ and corresponding instance of its superclass $C(CIV_2)$. An undirected edge from CIV_1 to CIV_2 with the annotation *ANNOTATION*, where *ANNOTATION* is an expression of form '$CIV_1.A_1$ *comparator* $CIV_2.A_2$', represents the explicit join between instances of classes $C(CIV_1)$ and $C(CIV_2)$ on attributes A_1 and A_2.

Transformation of G(RW) graph to OG(RW) graph consists of three parts: transformation of nodes, transformation of edges and transformation of node annotations.

Transformation of nodes

Transformation of nodes is being performed as follows. First, for each node with name RTV in RV-set of nodes of G(RW) graph, it has to be created a node in OV, with the same name. This node corresponds to instance variable of the class, which was created from relation, to which the node from set RV corresponds. The new node in OV receives the same annotations as the node of the graph GW, from which it has been created. In our example we obtain the graph with three nodes: EXAM, GRADUATED_STUDENT and STUDENT.

Transformation of edges

Transformation of edges is being performed as follows. For each pair of nodes, RTV_1 and RTV_2, in RV, let E be the set of edges between these two nodes. This set is being divided on the partitions, from which the edges of the object-oriented graph are being created.

First, those edges that correspond to implicit join between the class and superclass, are being transformed to directed edge without annotation. In our example, the directed edge without annotation from node GRADUATED_STUDENT to node STUDENT, is being created. Further, there have to be converted the edges that correspond to implicit join between two classes, where one of them is the domain of complex attribute of the other one. Those edges are being converted to directed edge with annotation. In our example the directed edge with *graduated_student* annotation (the name of the complex attribute) from node EXAM to node GRADUATED_STUDENT, is being created. All remaining edges are being directly transformed to undirected edges with annotation of the object-oriented graph, i.e. they are included in set OE_2. These edges, in essence, correspond to explicit join.

Transformation of node annotations

Finally, it is necessary to transform the node annotations. The node annotations of the relational graph correspond to selections in relational WHERE-predicate. Their transformation is being performed depending on whether an attribute was removed from the definition of a class, which was created during schema transformation from a relation,

which contained that attribute, on which selection is being done.

The procedure ([11]) is recursive. In our example, for selection *'course_name = mathematics'*, first, a new CIV for the class course, which we can name COURSE, node COURSE and a directed edge with annotation *exam_course* from node EXAM to node COURSE are being created. The procedure is being repeated for node COURSE and attribute course_name. The attribute *course_name* remains in the class *course*, and the node COURSE receives the annotation *'course_name =mathematics'*.

The following object-oriented predicate graph is being obtained (Fig 3):

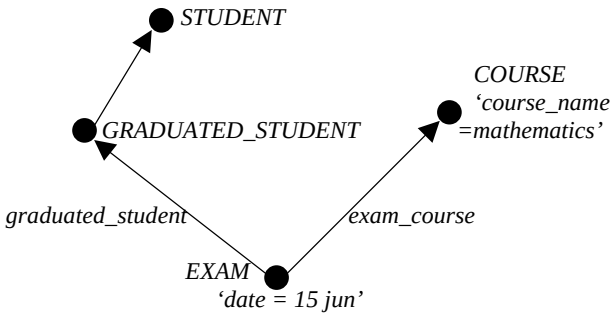
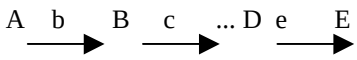


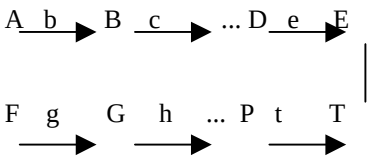
Figure 3. Example of the object-oriented WHERE-predicate graph

4.3. Construction of object-oriented predicate from its graph

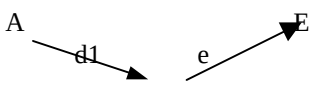
Let us describe all basic constructions, which can occur in the object predicate graph, and explain their semantics. The main constructions are shown in a Fig. 4. In each construction class the instance variables (CIV) are indicated by large characters and the complex attributes, whose domains are the classes appropriate to class instance variables, which are situated at the ends of directed edges, are indicated by small characters. Some of the annotations may be empty strings.

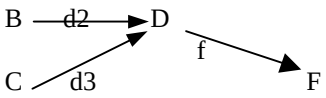


(a) *A directed path*

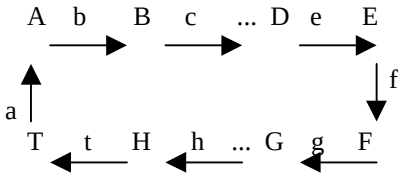


(b) *An undirected bridge*





(c) *A node associated with 3 or more edges*



(d) A directed cycle

Figure 4 Basic constructions in object-oriented predicate graph

Construction 1

In a Fig. 4(a) the following is assumed: (1) there is no directed edge, which ends in the node A; (2) except node A, no other node is connected to the any other node, except its predecessor in the directed path; (3) no node appears more than once in the directed path. Semantics of this construction is that a traversal from A to Å is needed. In the object-oriented WHERE-predicate definition it corresponds to the expression:

(A: Selection-on-A).(b: Selection-on-B).... (e: Selection-on-E) .

Construction 2

In a fig. 4(b) two directed paths are connected with an undirected edge. It is possible that one or two of directed paths have length equal to zero. Suppose that *"E.Attribute1 Comparator T.Attribute2"* is the annotation on the undirected edge. This construction is translated to explicit join on simple attributes.

(A: Selection-on-A).(b: Selection-on-B)... (e: Selection-on-E).Attribute1

Comparator

(F: Selection-on-F).(g: Selection-on-G)....(t: Selection-on-T).Attribute2

Construction 3

This construction is represented in Fig. 4(c). If there are three or more edges connected to a node, then all these edges refer to the same instance of class, to which CIV, corresponding to this node corresponds. For example, d1, d2, d3, e, f must reference the same instance of class C(D). This can be achieved by specifying a reference variable, which corresponds to one of incoming edges. For example, in the situation shown in Fig. 4(c), it is possible to define, that the reference variable X corresponds to the path expression *(A: Selection-on-A).(d1: Selection-on-D).* Of course, this has to be added to FROM-clause of query. Then we obtain the following object-oriented predicates:

*(B: Selection-on-B).d2=**X** AND (c: Selection-on-C).d3=**X** AND **X**. (e: Selection-on-E) AND **X**. (f: Selection-on-F)*

Construction 4

This construction is represented in Fig. 4(d). If we start the evaluation, for example, from node A, then complex attribute *a* of the class corresponding to CIV T, i.e. class C(T), references the same instance of the class C(A). Therefore it is possible to translate this construction into the following object-oriented predicate:

(A: Selection-on-A).(b: Selection-on-B).... (d: Selection-on-D).(e: Selection-on-E).(F: Selection-on-F).(g: Selection-on-G)....(h: Selection-on-H).(t: Selection-on-T) (a:)= A.OID

We obtain various object-oriented predicates, if we start from different nodes. Of course, all these predicates are equivalent. An algorithm which traverses the graph according to the semantics and which creates the object-oriented predicate and whole object-oriented query is described in [11]. In our example the following object-oriented query is being obtained:

***SELECT** exam.graduated_student.average*
***FROM** exam*
***WHERE** (exam: date='15 jun').(exam_course: course_name*

=mathematics)

We have described some of the main ideas of a method for translation of SELECT queries. Using this result, we have also developed methods for translation of UPDATE, INSERT and DELETE queries.

5. Conclusion

In this paper we have discussed some aspects of database transformation from relational to object-oriented database, including query translation. We have described some of the main ideas of a schema transformation algorithm, two algorithms for data conversion and proposed a method for translation of SELECT queries.

Acknowledgments

The author wishes to thank to Dr. Gordana Pavlovich-Lazetich, from University of Belgrade, and to Dr. Lev Nikolaevich Korolev, from Moscow State University, for their help in his work.

References

1. Fong J. ″Converting Relational to Object-Oriented Databases″. SIGMOD Record, Vol.26, No. 1, March 1997
2. Date, C.J. ″An Introduction to Database Systems ″, (6th edition) Addison-Wesley Systems Programming Series. 1995
3. Ramanathan S., Hodges J. ″Extraction of Object-Oriented Structures from Existing Relational Databases″. SIGMOD Record, Vol.26, No. 1, March 1997
4. Piatetsky-Shapiro G., Frawley W. ″Knowledge Discovery in Databases ″. AAAI Press / MIT Press, California, 1991
5. Meng W., Yu C., Kim W., et al. ″Construction of a Relational Front-end for Object-Oriented Database Systems″. Proceedings of Ninth International Conference on Data Engineering, Vienna, Austria, IEEE Computer Society, 1993
6. Kim W. ″On Optimizing an SQL-like Nested Query″. ACM TODS, 1982
7. Kim W. ″Introduction to Object Oriented Databases″. The MIT Press, 1990
8. Stanisic P. "Transformacija relacionih baza podataka u objektno-orijentisane”. MD thesis. University of Belgrade, Belgrade, 1998
9. Stanisic P. "Schema Transformation from Relational to Object-Oriented Database as a part of Database Reverse Engineering Process". Mathematica Montisnigri, No. 9, 1998
10. Stanisic P. "Data Conversion from Relational to Object-oriented Database" (in Russian). Vestnik Moskovskogo Universita, Ser.15, No. 1, 1999 (accepted)
11. Stanisic P. "A Method for Translation of Relational Queries into Equivalent Queries in Transformed Object-Oriented Database" (in Russian). Numerical methods and computational experiment, edited by A.A. Samarsky, V.I. Dmitriev, 1998 (accepted)