The VLDB Journal (1997) 6: 261–281

The VLDB Journal
© Springer-Verlag 1997

Query processing over object views of relational data

Gustav Fahl, Tore Risch

Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden; e-mail: {gusfa, torri}@ida.liu.se

Edited by: M.T. Özsu. Received April 12, 1995 / Accepted April 22, 1996

Abstract. This paper presents an approach to *object view* management for relational databases. Such a view mechanism makes it possible for users to transparently work with data in a relational database as if it was stored in an object-oriented (OO) database. A query against the object view is translated to one or several queries against the relational database. The results of these queries are then processed to form an answer to the initial query. The approach is not restricted to a 'pure' object view mechanism for the relational data, since the object view can also store its own data and methods. Therefore it must be possible to process queries that combine local data residing in the object view with data retrieved from the relational database. We discuss the key issues when object views of relational databases are developed, namely: how to map relational structures to sub-type/supertype hierarchies in the view, how to represent relational database access in OO query plans, how to provide the concept of object identity in the view, how to handle the fact that the extension of types in the view depends on the state of the relational database, and how to process and optimize queries against the object view. The results are based on experiences from a running prototype implementation.

Key words: Object views – Relational databases – Object-oriented query processing – Object-oriented federated databases – Query optimization

1 Introduction

An object view of a relational database makes it possible for users to transparently work with data in a relational database as if it was stored in an object-oriented (OO) database. When the term 'OO database' is used in this paper, it refers to a database system with an OO data model and a query language that is at least as powerful as SQL[1]. Queries against the object view are translated to queries against the relational database. The results of these queries are then processed to form the answer to the initial query. In this paper, we concentrate on *access* to relational databases via object views, not updates.

Object views of relational databases are an important component of multidatabase systems[2] (Bright et al. 1992; Litwin et al. 1990; Sheth and Larson 1990). Most multidatabase systems use a canonical data model (CDM) to deal with the problem of data model heterogeneity. It is generally agreed that object-oriented data models are appropriate as the CDM in a multidatabase system (Saltor et al. 1991). Using the terminology of Sheth and Larson (1990), if an object-oriented CDM is used, the different local schemas must be mapped to object-oriented structures in the component schemas, i.e., object views must be established for the different types of component databases. Since relational databases have such a dominating position on the database market, techniques for developing object views of relational databases are especially important.

The results in this paper are based on experiences from the development of an object view mechanism for the relational DBMS Sybase. The approach presented in this paper is not restricted to 'pure' object views of relational databases, since the software component implementing an object view can also store its own data and methods. In fact, the object view mechanism has been implemented by extending an existing OO DBMS (AMOS). The result is an OO DBMS which can handle existing relational data in a general and efficient way. We will use the term *Translator* for this software component. Part of the schema of the Translator is an object view of a relational database. Figure 1 compares the architecture of an OO DBMS to that of a Translator[3].

Figure 2 shows the design phases in our approach. First, the schema of the object view is decided. This means modelling the information in the relational database using our OO data model. OO data models are semantically richer than the relational data model, and the object view schema explicitly

Correspondence to: Tore Risch

[1] The term 'object-relational' is becoming more and more used for this kind of system (Stonebraker and Moore 1996)

[2] We use the term 'multidatabase system' for the general concept of a system in which it is possible to access data from multiple databases, which may be distributed, heterogeneous, and autonomous.

[3] Note that we are assuming a 'bottom-up' development – the object view is created for an *existing* relational database, we do not use a relational database as persistent storage for an OO DBMS

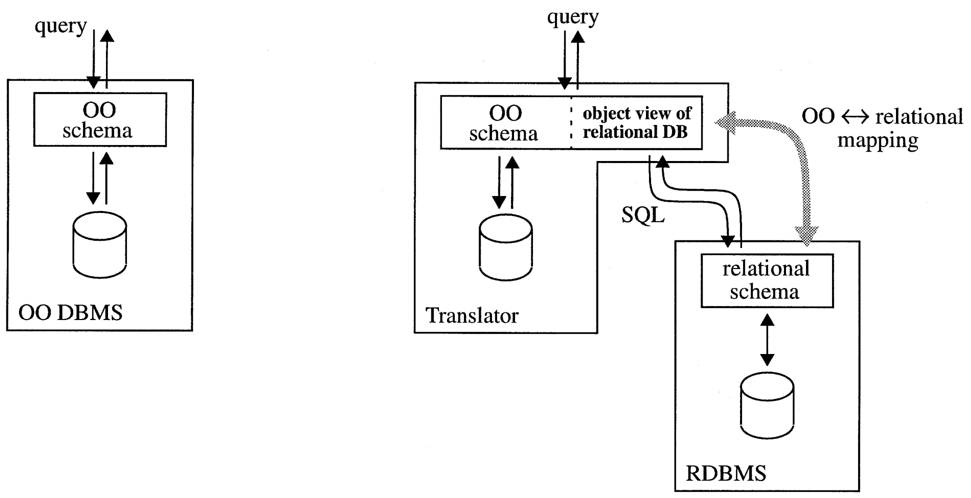


Fig. 1. Architectures of OO DBMSs (left) and Translators (right)

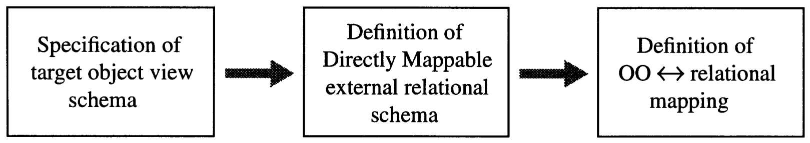

Specification of target object view schema
Definition of Directly Mappable external relational schema
Definition of OO ↔ relational mapping

Fig. 2. Object view design phases

captures semantics which was only implicitly represented in the relational schema.

To simplify the mapping between the relational database and the object view, we assume that the relational database is structured in a particular way. Relational databases structured this way are said to be *directly mappable* to the target object view. When the relational database *is not* directly mappable to the object view, we first define an external relational schema, i.e., a set of relational views, that *is* directly mappable.

During the third phase, the mapping between the object view and the relational database is defined using an *object view definition language*. The types and functions in the object view are created, and a set of object view resolution rules are generated.

The focus of this paper is on *query processing* over object views of relational data. Figure 3 illustrates the conventional database query processing methodology (Jarke and Koch 1984) adapted for OO databases. A high-level query language is used to formulate the query. This is converted to an internal declarative representation – an object calculus expression. The calculus optimizer applies syntactic and semantic rewrite rules to simplify the calculus expression. The calculus expression can be translated to many equivalent, procedural, algebraic expressions. This translation is performed by the algebra generator. The execution cost for each of these algebraic expressions is then estimated and the cheapest is selected for execution[4].

[4] Of course, this is only a schematic overview. For example, the entire set of possible algebraic expressions is seldom generated, it may even be impossible to do so in finite time. One way to deal with the complexity is to use a randomized algorithm for exploration of the search space. Another common algebraic optimization methodology is to generate *one* initial algebra expression, and then apply equivalence-preserving rewrite rules to translate the algebra expression into a cheaper one.

Figure 4 gives the corresponding overview of query processing in our Translators. A new phase is added: object view resolution. Some predicates in the object calculus are defined in terms of relational database access. During object view resolution, these predicates are replaced by their definitions[5].

A Translator needs a way to represent relational database access in query plans. We use domain relational calculus (DRC) predicates for this. Thus, the output from object view resolution is an object calculus expression where some predicates are specially marked predicates (DRC predicates) representing queries to the relational database.

The optimization phases face new requirements due to the fact that part of the Translator schema is a view of a relational database:

- The calculus optimizer needs to be extended with new semantic rules for simplifying the calculus expression.
- During algebra generation, the DRC predicates must be replaced with function calls which send SQL queries to the relational database. The algebra generator must be extended with knowledge of the different ways to generate these SQL queries.
- The cost estimator must be extended with knowledge of the cost of sending different SQL queries to the relational database.

The reader might find it useful to skim through appendix C at this point. It describes the query processing steps for an example query to the Translator, and even though the details will be hard to follow, it might assist in providing an overview of the problems addressed in this paper.

[5] A useful analogy is view expansion during query processing in a relational database system. There, relational calculus predicates referring to a view are replaced with predicates re ferring to base relations.

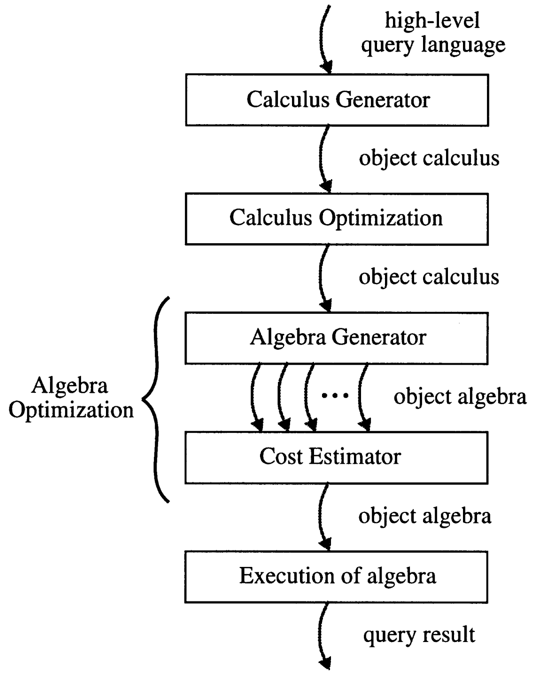


Fig. 3. Query processing in OO DBMSs

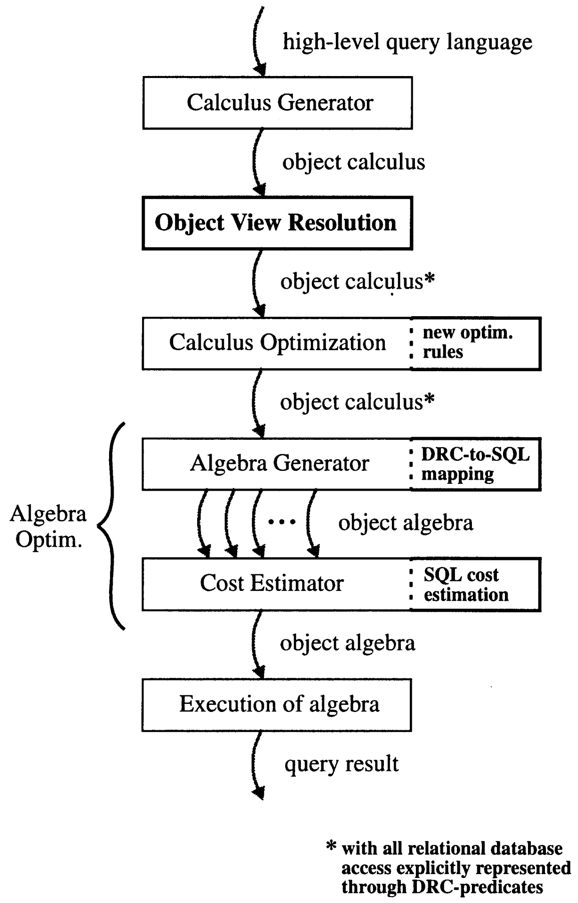


Fig. 4. Query processing in a Translator

We will end this introductory section by listing the contributions of the paper (Sect. 1.1) and by presenting an example (Sect. 1.2) which will be used throughout the paper. The rest of the paper is organized as follows.

Section 2 discusses related work. Section 3 gives a brief overview of the AMOS project (Fahl et al. 1993), which is the framework in which this work has been performed. The section describes the AMOS data model and query language, and gives an overview of query processing in AMOS, which includes presenting the object calculus and object algebra used in this paper. Section 4 covers the three design phases in Fig. 2. The first phase – specification of the target object view schema – is only discussed very briefly, since it has been covered extensively in previous work. Section 4.1 concerns the second phase – creation of directly mappable external relational schemas. Sections 4.2–4.4 cover the third phase – creation of the OO $\leftrightarrow$ relational mapping. Section 4.2 discusses ways to represent relational database access in OO query plans. Section 4.3 shows how the concept of object identity can be provided in the view. Section 4.4 discusses type membership tests. Section 5 covers query optimization. Calculus optimization is discussed in Sect. 5.1 and algebra optimization is discussed in Sect. 5.2. Section 6, finally, gives a summary of the paper and discusses future work.

1.1 Contributions

Some of the discussion in this paper on the relationship between the relational data model and OO data models, and the semantics of object views of relational data has been covered in previous work, especially in the Pegasus project (Shan et al. 1995). See the section on related work (Sect. 2).

Our main contributions are in the area of *query processing* in object views of relational data. The paper aims to give a complete overview of the problem, including query processing. More specifically, the following aspects have not been covered in previous work:

- The concept of a relational database being directly mappable (DM) to an object view and the use of relational views for relational databases that are not DM.
- The importance of having all relational database access explicitly represented in object view query plans, the use of DRC predicates to accomplish this, and the applications of this to function calls.
- Explicit coverage of the role of object identity and instance-of relationships during query processing.
- Semantic optimization rules for simplification of subqueries which are overlapping due to the definition of object views, and for removal of unnecessary translations between OIDs and primary key values.
- A discussion on the additional complexity introduced to query optimization by the presence of relational database access in query plans, and a heuristic rule suitable as an initial way to manage this new complexity.

Since our Translators can store their own data and methods, the paper also addresses a special case of query processing in heterogeneous multidatabase systems, namely integration of relational and OO databases.

The functional data model used in this paper is in some aspects closer to the relational data model than many other object models. However, the problems associated with subtype/supertype hierarchies, object identity, the instance-of relationship, and function/method application, are common to all object models. The relative simplicity of our object model should hopefully make the discussion on these general concepts clearer.

1.2 The company example

Throughout the paper we will use the same example of a relational database. We will use this example to show the relationship between a relational database and the corresponding object view, and to show how queries against the object view are processed. The examples are intentionally kept very simple to enable us to focus on the essentials. The impact that complex databases and queries would have on our approach is discussed in the section on future work (Sect. 6.1). However, note that scalability regarding database and schema size is not a problem, since our approach is based on non-materialized views and query translation. No relational data is stored in the object view, only schema information and the mapping to the relational database. The example relational database is shown in Fig. 5.

We will use the AMOS data model as our example of an OO data model. It is a functional and OO data model which is based on the IRIS data model (Fishman et al. 1989; Lyngbaek et al. 1991), which in its turn is based on the functional data model DAPLEX (Shipman 1981). The subset of the AMOS data model used in this paper is identical to the IRIS data model. It is further described in Sect. 3.1. Figure 6 shows the schema of the corresponding object view of the company database. The types `secretary` and `salesman` are subtypes to `employee`. The properties of employees, secretaries, and salesmen are modelled by functions. The hollow arrowhead for the `hobby` function indicates that this is a multi-valued function.

Figure 7 shows the extension of the object view. The relational database stores information about four employees. Two of these (‘anne’ and ‘bob’) are salesmen and one (‘doris’) is a secretary. One of them (‘colin’) is neither a salesman nor a secretary, but still an employee. Accordingly, there should be four objects in the object view (`:e1`, `:e2`, `:e3`, and `:e4`). Two of them should be direct instances of `salesman`, one a direct instance of `secretary`, and one a direct instance of `employee`[6].

Informally, the semantics of the mapping between tuples in the relational database and objects in the object view is as follows. There is one object for each tuple in the `employee` table. The primary key `enr` is used to define the correspondence between tuples and objects. For example, the enr 314 corresponds to the object `:e1`. All objects are instances of the `employee` type. An object is also an instance of the type `secretary` (`salesman`) if there is a tuple in the `secretary` (`salesman`) table with the enr that corresponds to the object. For example, the object `:e1` is an instance of the type `salesman`, since there is a tuple with enr=314 in the `salesman` table.

[6] We distinguish between an object being a direct instance of a type and being an instance of a type by generalization. For example, the object :e1 is a *direct instance* of the type salesman, and it is an *instance by generalization* of the supertypes to salesman.

2 Related work

Multibase (Landers and Rosenberg 1982) has an architecture similar to that of AMOS and uses the functional data model DAPLEX (Shipman 1981) as the CDM. AMOSQL is a DAPLEX derivative, but an important difference is that AMOSQL is object-oriented. Queries in AMOSQL can return object identifiers (OIDs). Another difference is the role of the translation component. An AMOS Translator captures as much of the semantics of the data source as possible, whereas, in Multibase, the translated schema is the simplest possible and all semantic enrichment is performed in the integration modules.

The Pegasus project (Albert et al. 1993; Ahmed et al. 1993; Shan et al. 1995) uses the IRIS data model as the CDM and an extension to OSQL as the data manipulation language. In the area of object views of relational data, the Pegasus project has concentrated on techniques for automatic generation of the *schema* of the object view.

The relationship between schemas in the relational data model and schemas in semantically rich data models, such as an OO data model or an extended entity-relationship (EER) model, is fairly well understood. Translation of an EER schema into a relational database schema is a central part of many database design methodologies (Elmasri and Navathe 1989). Recent work has shown how to identify semantic modelling constructs in a relational database schema, and how it can be (semi-) automatically transformed into an EER/OO schema (Albert et al. 1993; Johannesson and Kalman 1989; Markowitz and Markowsky 1990; Navathe and Awong 1987; Litwin et al. 1990).

The focus of our work is on *query processing* in non-materialized object views of relational data. We show how queries against the object view are translated and optimized.

Query processing in object views of relational data relates to query processing in OO database systems (Özsu and Blakeley 1994). Object algebras and algebra optimization are discussed in Straube and Özsu (1990), Shaw and Zdonik (1990), and Demuth et al. (1994). Other related research areas are views in OO database systems (Abiteboul and Bonner 1991; Bertino and Martino 1991; Chomicki and Litwin 1994; Heiler and Zdonik 1990), and work on providing a more general view mechanism for relational databases (Krishnamurthy et al. 1991).

The work most closely related to ours that we are aware of is reported in (Kemp et al. 1994). The main differences in their approach are that object identifiers (OIDs) are stored in the relational database, that new objects cannot be added to the relational database, and that queries that need to combine relational data with other data causes the execution of many small SQL queries. We will further relate our work to theirs later in the paper. Methods to completely translate OO queries to the corresponding relational ones have been proposed recently by (Qian and Raschid 1995) and (Yu et al. 1995), but neither of these proposals handle OO queries employee

enr	name	salary	manager
314	anne	20000	265
159	bob	15000	314
265	colin	25000	NULL
358	doris	13500	314

emp_hobbies

emp	hobby
314	sailing
314	golf
159	golf
265	tennis
265	fishing
265	golf
358	tennis

secretary

enr	typingspeed
358	1100

salesman

enr	district	sales
159	charlotte	70
314	raleigh	40

Fig. 5. The company database (example of a relational database)

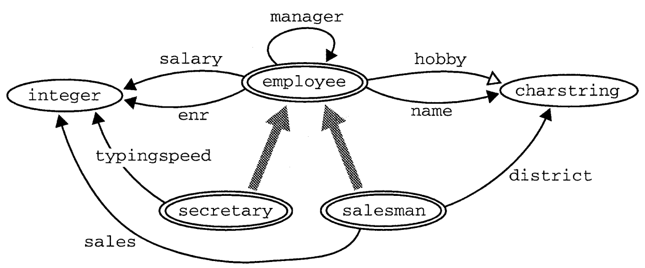


Fig. 6. Schema of the object view of the company database

that return OIDs or that combine object view data with data in the relational database.

3 AMOS

This section gives an overview of the AMOS project (Fahl et al. 1993) which is the framework in which this work has been performed. Section 3.1 presents the data model and query language used in AMOS. Readers familiar with the IRIS data model (Fishman et al. 1989) and the OSQL query language (Lyngbaek et al. 1991) may skip this section since the subset of the AMOS data model and query language that is used in this paper is identical with IRIS/OSQL. Section 3.2 describes the multidatabase aspects of AMOS. In this paper, we concentrate on the AMOS components called Translators, i.e., the software that implements an object view of a relational database. Translators are the subject of Sect. 3.3. Section 3.4 briefly describes query processing in AMOS and the object calculus and object algebra used in this paper.

3.1 The AMOS data model and query language

There are three basic constructs in the AMOS data model; *objects*, *types* and *functions*. Objects are used to model entities in the domain of interest. Types are used to classify objects; an object is an instance of one or more types. Properties of objects and relationships between objects are modelled by functions.

Types are divided into *literal* types and *surrogate* types. The extension of a literal type is fixed (often not enumerable), and instances of a literal type are self-identifying; no extra object identifier is needed. Examples of literal types are `integer`, `charstring`, and `real`. Surrogate types and instances of surrogate types are created by the system or by users. Instances of surrogate types are identified by a unique, immutable, system-generated OID. Examples of surrogate types are `person`, `document`, `country`, etc.

Types are organized in a subtype/supertype graph. Figure 8 shows part of the type graph of AMOS. The most general type is `object`; all other types are subtypes of `object`. User-defined types are subtypes of a special type called `usertypeobject`. Types are objects too, and are instances of the type `type`. User-defined types are also instances of the type `usertype`.

A function is implemented in one of three different ways; it may be *stored*, *derived*, or *foreign*. For stored functions, the extension is stored directly in the database. A derived function uses the AMOSQL query language to calculate the extension. A foreign function is implemented in a general programming language, such as C or LISP. Functions can be overloaded and they are invertible.

The general syntax for queries is:

```
select <result>
for each <type declarations for local
         variables>
where <condition>
```

For example:

```
select name(e)
for each employee e
where hobby(e)='sailing'
```

AMOS functions and queries return *bags of tuples of objects*, denoted $\{| < \ldots >, \ \ldots, \ < \ldots > |\}$. The semantics for nested function calls, which will be referred to as ‘DAPLEX semantics’ (Shipman 1981; Gray 1984), is defined as:

$f(g(x)) = \{|\ y$ such that y is in $f(z)$ and z is in $g(x)\ |\}$

3.2 The AMOS multidatabase system architecture

The software components of the AMOS multidatabase system architecture (Fahl 1994) are shown in Fig. 9. Using the terminology of Sheth and Larson (1990), it can be seen as a combination of a loosely coupled and a tightly coupled

```
direct_instance_of(:typeEmployee)=:e3
direct_instance_of(:typeSecretary)=:e4
direct_instance_of(:typeSalesman)=:e2
direct_instance_of(:typeSalesman)=:e1

enr(:e1)=314               enr(:e2)=159                 enr(:e3)=265               enr(:e4)=358
name(:e1)='anne'           name(:e2)='bob'              name(:e3)='colin'          name(:e4)='doris'
salary(:e1)=20000          salary(:e2)=15000            salary(:e3)=25000          salary(:e4)=13500
manager(:e1)=:e3           manager(:e2)=:e1             hobby(:e3)='tennis'        manager(:e4)=:e1
hobby(:e1)='sailing'       hobby(:e2)='golf'            hobby(:e3)='fishing'       hobby(:e4)='tennis'
hobby(:e1)='golf'          district(:e2)='charlotte'    hobby(:e3)='golf'          typingspeed(:e4)=1100
district(:e1)='raleigh'    sales(:e2)=70
sales(:e1)=40
```

Fig. 7. Extension of the object view of the company database

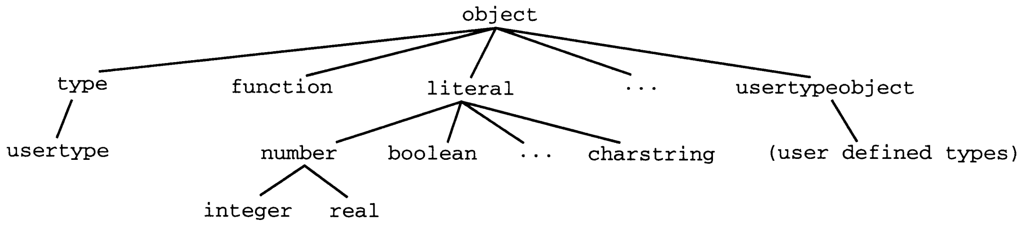


Fig. 8. Part of the AMOS subtype/supertype graph. Each *line* represents a subtype/supertype relationship. `object` is the most general type

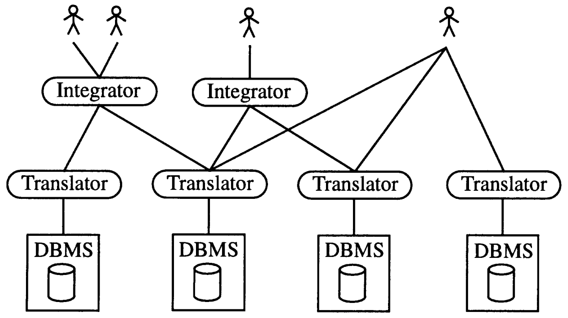


Fig. 9. Software components in the AMOS multidatabase system architecture

federated database system. The basic way to access data in AMOS is through a multidatabase language, but it is also possible to provide integrated views of multiple data sources.

In this paper, we concentrate on Translators for relational databases, i.e., the software component that implements an object view of a relational database.

3.3 Translators

A Translator provides the functionality of an AMOS DBMS augmented with the notions of *mapped types* and *mapped objects*. A mapped type is a type for which the extension is defined in terms of the state of an external database. In our case, the external database is relational, and the extension of a mapped type is defined so that there is a one-to-one mapping between instances of the mapped type and tuples in some relation or view in the external database. The instances of mapped types are called mapped objects[7].

Figure 10 shows the subtype/supertype graph for Translators. Mapped types are subtypes to the type `usertypeobject` and are instances of the type `mappedtype`. Figure 11 shows how mapped types fit into the subtype/supertype graph.

We will use the term *most general mapped type* (*mgmt* for short) for mapped types that are direct subtypes to `usertypeobject`. In Fig. 11, the types $\texttt{MT}_1$ and $\texttt{MT}_6$ are mgmts. In the company example, the type `employee` is the only mgmt.

We will also use a function called `mgmt`. The function takes a mapped type MT as argument and returns the supertype to MT which is a mgmt. For example:

```
mgmt(:typeSalesman) = :typeEmployee
mgmt(:typeEmployee) = :typeEmployee
mgmt(direct_instance_of(:e4)) =
  = mgmt(:typeSecretary) = :typeEmployee
```

3.4 Query processing in AMOS – object calculus and object algebra

Figure 3 in Sect. 1 gives an overview of query processing in AMOS. This section uses an example query to describe the query processing steps and the object calculus and algebra used in AMOS. A more thorough description of query processing in AMOS can be found in (Litwin and Risch 1992)[8].

Consider the following AMOSQL query:

```
select s, salary(manager(s))          (Q1)
for each salesman s
where hobby(s)='golf'
```

[7] The notions of mapped types and mapped objects are similar to what is called *virtual classes* and *imaginary objects* in (Abiteboul and Bonner 1991). The main difference is that the object view in Abiteboul and Bonner (1991) is defined over an OO database, rather than over a relational database.

[8] (Litwin and Risch 1992) use the logical language ObjectLog to describe internal query processing. Object calculus expressions in this paper correspond to type-resolved ObjectLog rules in (Litwin and Risch 1992). Object algebra expressions in this paper correspond to type- and binding-pattern-resolved ObjectLog rules in (Litwin and Risch 1992).

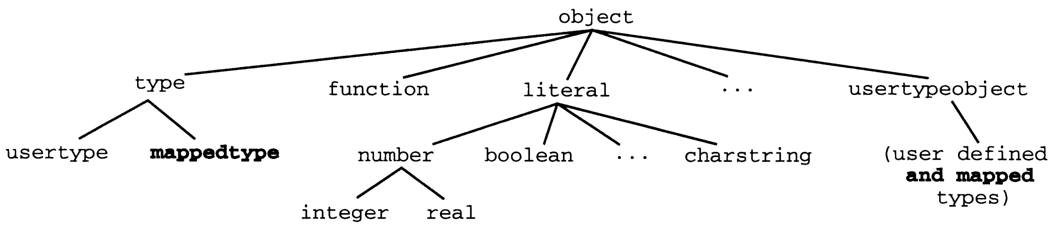


Fig. 10. Subtype/supertype graph for Translators

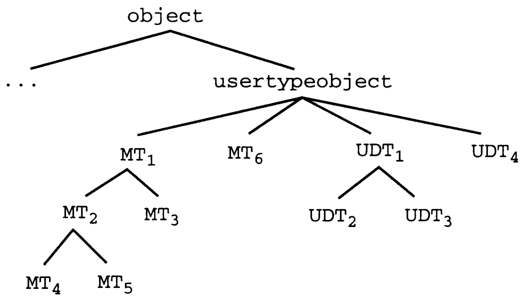


Fig. 11. Mapped types (MT_{1-6}) and user-defined types (UDT_{1-4}) in the subtype/supertype graph for Translators

A natural language formulation of this query would be something like: 'for each salesman that has golf as a hobby, retrieve that salesman together with the salary of his/her manager'.

The calculus generator transforms the query to a normalized[9], type-resolved[10] object calculus expression (existential quantifiers are implicit):

$$
\begin{aligned}
&\{\ s, sal\ |\\
&\quad Employee(m)\ \wedge\\
&\quad sal = salary_{employee \rightarrow integer}(m)\ \wedge\\
&\quad m = manager_{employee \rightarrow employee}(s)\ \wedge\\
&\quad 'golf' = hobby_{employee \rightarrow charstring}(s)\ \wedge\\
&\quad Salesman(s)\ \}
\end{aligned}
$$

The calculus optimizer applies syntactic and semantic rules to simplify the calculus expression. In this case no such rules are applicable.

The algebra generator transforms the calculus expression to equivalent procedural representations – object algebra expressions. Figure 12 shows two of the possible object algebra equivalents. The input to and output from an algebra operator is a bag of tuples of objects. For simplicity reasons, we do not show the full signature of the type-resolved functions in the algebra trees.

The leaf nodes in an algebra tree are enumerable types, producing their extent as output. An algebra operator is one of $\{\pi, \sigma, \times, \cup, \cap, \bowtie, \gamma\}$. The π, $\times$, $\cup$, $\cap$, and $\bowtie$ operators have the same semantics as their relational counterparts (Elmasri and Navathe 1989), with the exception that input and output are bags (denoted $\{|\ldots|\}$) rather than sets. The γ (generate) operator is a new operator that performs function application. It can thereby introduce objects other than those produced by the leaf nodes into the query plan. In this way, the γ operator is similar to the generate operator in Straube and Özsu (1990), and the image operators in Shaw and Zdonik (1990) and Demuth et al. (1994). The σ (select) operator is identical to the relational selection operator with the exceptions that input and output are bags, and that the selection condition may contain function calls. The γ and σ operators are defined formally in appendix A.

Figure 13 shows the flow of data[11] if our example query is executed according to the left plan in Fig. 12 (assuming the database extension in Fig. 7).

Let us use the node marked (*) to illustrate the 'DAPLEX semantics' of the γ operator. The employee object (:e1) of the first input tuple (<:e1, 20000>) is the manager of two employees (:e2 and :e4). This means that the first input tuple results in two output tuples (<:e1, 20000, :e2> and <:e1, 20000, :e4>). The employee object of the second input tuple is not the manager of any employees, which means that the second input tuple results in zero output tuples. The third and fourth input tuples result in one and zero output tuples, respectively, giving a total of three output tuples.

To simplify the expressions in our examples, we allow the γ operator to exclude some of the input variables from the result, i.e., perform projection. This is used in the top two nodes in the query plan to the right in Fig. 12.

The optimizer uses a cost model (Litwin and Risch 1992) to select the cheapest algebra expression.

4 Object view design

This section covers the three design phases of our approach illustrated in Fig. 2.

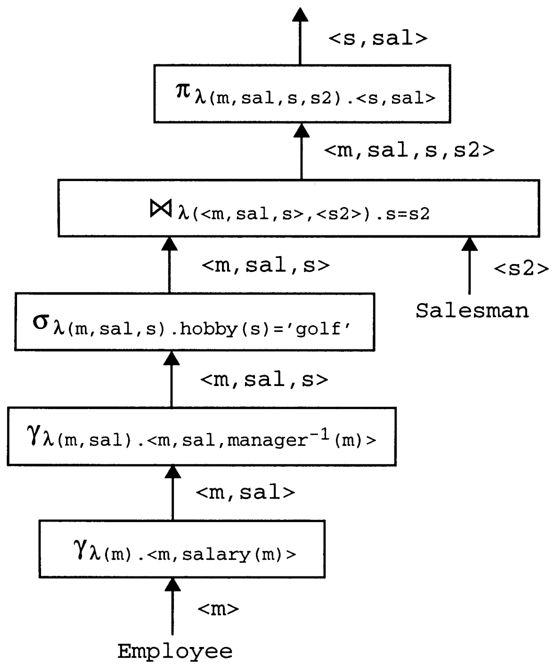
The first step when developing an object view for a relational database is to identify semantic modelling constructs in the relational database, and to define the OO schema corresponding to the relational database. Information that could be used as an aid in this process includes the database schema, the database content, and functional dependencies. This has been covered extensively in previous work (see Sect. 2) and will not be discussed further here.

Section 4.1 concerns the second phase – creation of external relational schemas that are DM to an object view. We assume that subtype/supertype relationships are represented in a particular way in the relational database, and define relational views otherwise.

9 Disjunctions of conjunctions, no nested function calls, existential quantifiers moved out as far as possible.

10 I.e. where functions are annotated with their type signatures

11 Since AMOS uses stream-oriented query processing (Straube and Özsu 1995), the intermediate results are not actually materialized. On the physical level, the input to and output from an algebra operator is a *stream of tuples of objects*.



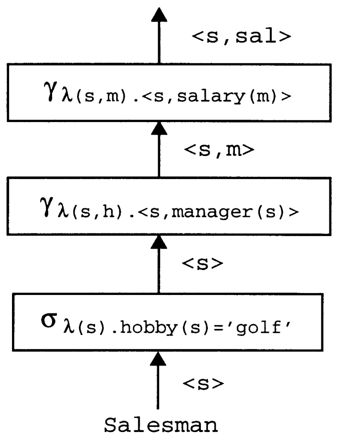


Fig. 12. Two of the possible object algebra representations for the example query: straightforward, but non-optimal translation from calculus (*left*), and one of the candidates for optimal execution strategy (*right*)

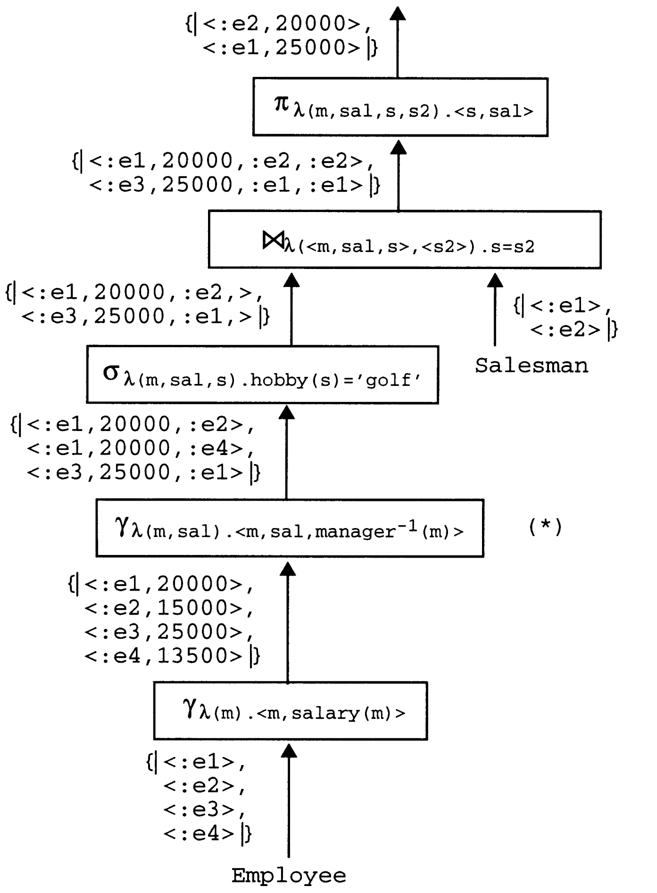


Fig. 13. Data flow during query processing according to the left plan in Fig. 12

During the third design phase, the mapping between the relational database and the object view is created. In our prototype implementation, the mapping is defined using a declarative object view definition language like the one in (Albert et al. 1993). For example:

```
declare primary key of 'employee'
  is 'enr';
create type employee as
  mapped to relation 'employee';
create function
  salary(employee)->integer
  as mapped to attribute
  'employee.salary';
```

The Object View Definition Language is not discussed further here.

The view definition commands result in the creation of the object view types and functions, and in a set of object view resolution rules. Sections 4.2–4.4 concern the internal representation of the OO ↔ relational mapping.

Section 4.2 discusses the importance of having all relational database access explicitly represented in query plans in the object view. We use DRC predicates for this. Section 4.3 shows how the concept of object identity can be provided in the object view even though there is no such concept in the relational data model. Section 4.4 discusses type membership tests in object views. The extension of types in the object view depends on the state of the relational database. The consequences of this for query processing are discussed.

4.1 DM relational views

The concept of subtype/supertype relationships in OO data models has no corresponding concept in the relational data

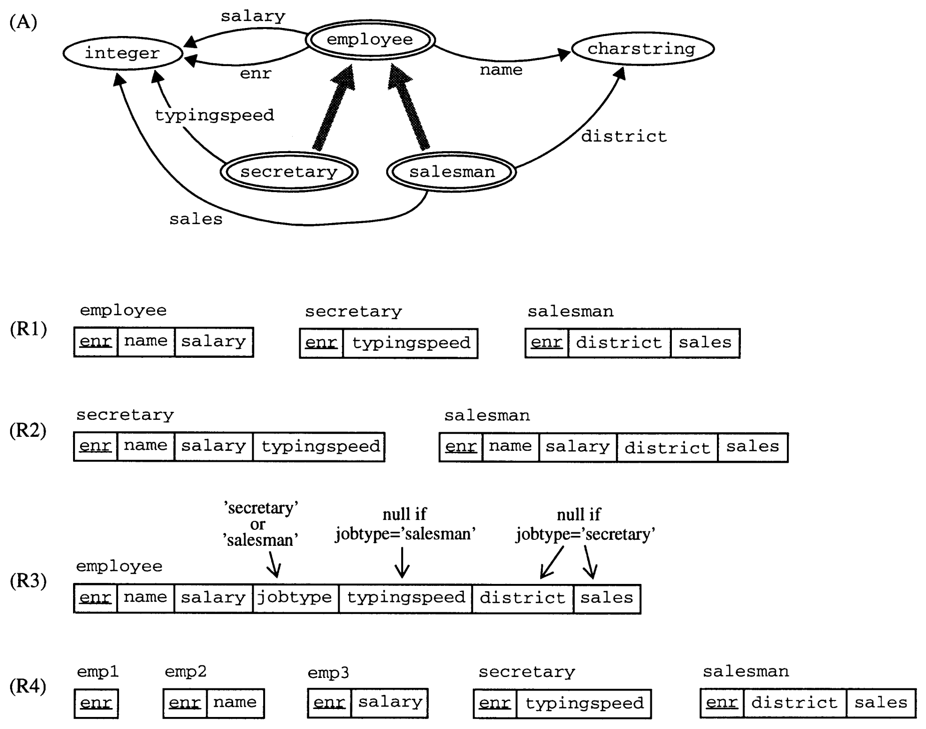


Fig. 14. Representing subtype/supertype relationships in relational schemas. Four alternative mappings from the AMOS schema (A) to relational schemas (R1, R2, R3, R4) are given

model. When subtype/supertype relationships exist between objects in the domain which is modelled, this is only represented implicitly in relational databases.

Figure 14 shows four alternative mappings from an AMOS schema, where subtype/supertype relationships are represented explicitly, to a relational database schema [adapted from Elmasri and Navathe (1989)]. Alternatives (R1) and (R2) are probably the most common. Note that the AMOS schema is a subset of the company example used elsewhere in the paper.

In alternative (R1) all types have their own relation. In alternative (R2) there is no relation for the supertype employee. The attributes (functions) of the supertype are duplicated in all the relations representing the subtypes. Alternative (R3) has one relation for all types. The `jobtype` attribute specifies the type of the employee (secretary or salesman)[12]. The relation schema contains all the attrib utes of all the subtypes. If an attribute is not applicable (e.g., `typingspeed` for salesmen) it is given the value `null`. Alternative (R4) is similar to (R1), but the name and salary for employees are stored in separate relations. This kind of vertical fragmentation is often used to avoid `null` values in a relation. There are inclusion dependencies from `emp2.enr` to `emp1.enr`, and from `emp3.enr` to `emp1.enr`.

Alternatives (R1) and (R4) can be used for all kinds of specialization (overlapping/disjoint and total/partial)[13]. Alternative (R2) cannot be used for partial specialization. Alternative (R3) cannot be used for overlapping specialization.

To simplify the mapping between the relational database and the object view, we will assume that subtype/supertype relationships are represented in a particular way in the relational database. Relational databases represented in this particular way are said to be DM to the object view.

When an object view is defined over a relational database that is not DM to the object view, the first step is to use the relational view mechanism to define an external relational schema that *is* DM.

We define DM as follows:

Let OS be an EER/OO schema and RS the corresponding relational database schema. RS is DM to OS if:

for each type TP in OS there exists a relation R in RS such that there is a one-to-one mapping between instances of TP and tuples in R.

(End of definition.)

In Fig. 14, schemas (R1) and (R4) are DM to schema (A), whereas schemas (R2) and (R3) are not.

The main benefit of having the relational schema DM is that the object view mechanism does not need to handle all possible cases of relational database schemas. The reason for choosing our particular representation is that it simplifies management of the instance-of relationship. To check whether an object is an instance of a particular type, or to

[12] Or *null*, if the employee is neither a secretary nor a salesman ('just' an employee).

[13] A specialization is *overlapping* if an instance of the supertype can be an instance of more than one of the subtypes. Otherwise it is *disjoint*. A specialization is *total* if an instance of the supertype must be an instance of some of the subtypes. Otherwise it is *partial*. The overlapping/disjoint and total/partial criteria are orthogonal.

retrieve all instances of a particular type, it suffices to examine a single relation[14].

When an object view over a relational database that is not DM is going to be created, the developer must first create an external relational schema that is DM to the object view. Consider, for example, schema (R3) in Fig. 14. The following SQL statements could be used to define an external schema DM to schema (A):

```
create view employee* as
  select enr, name, salary from employee

create view secretary* as
  select enr, typingspeed
  from employee
  where jobtype='secretary'

create view salesman* as
  select enr, district, sales
  from employee
  where jobtype='salesman'
```

The attribute(s) we use as the basis for the instance-of relationship may contain duplicates. This may happen if the attribute is part of a composite key, or if key constraints are not enforced. These cases require relational view definitions with a 'distinct' specification to produce DM external schemas.

Note that it is not possible to create DM external relational schemas for all kinds of relational database schemas. For example, the DM external schema over schema (R3) above could only be created because we assumed that the domain of the `jobtype` attribute was fixed. Suppose that this assumption could not be made, i.e., that the domain of the `jobtype` attribute was character strings in general rather than the two specific character strings 'secretary' and 'salesman'. In that case, the DM extern l schema should have one relation for each distinct value that occurred in the jobtype column. In other words, the number of relations in the external schema would depend on the state of the underlying database. Such views cannot be created in current relational database systems. More general view mechanisms for relational databases are discussed in Krishnamurthy et al. (1991) and Litwin et al. (1991).

4.2 Representing relational database access in query plans

In most cases, it is advantageous to translate AMOSQL queries to as few and as large SQL queries as possible. A naive translation method that leads to a large number of small queries against the relational database would result in unnecessary communication between the Translator and the relational database.

For example, consider the following AMOSQL query:

```
select name(e)                         (Q2)
for each employee e
where salary(e)=15000
```

[14] The *pivot* relation plays a similar role in PENGUIN (Barsalou et al. 1991), which basically is a system where views can contain nested relations.

A straightforward, but naive, OO-to-relational mapping would be to view the name and salary functions as one atomic unit each, and implement them as foreign functions which made calls to the relational database. This would result in a query plan which would start with the salary function, which would execute the following SQL query:

```
select enr from employee
  where salary=15000
```

For each of the results (say X) of this query, the name function would be called, resulting in SQL queries of the following kind:

```
select name from employee where enr=X
```

It is obvious that an optimal translation should result in a single query against the relational database:

```
select name from employee
  where salary=15000
```

The problem with the simple approach taken above is that access to the relational database is embedded in the code of the `salary` and `name` functions. For the optimizer to be able to reason about and optimize access to the relational database, it is essential that all relational database access is represented explicitly in query plans.

A general requirement on all kinds of Translators should be that atomic units of access to the foreign data source are represented in Translator query plans in some way. The query processor of the Translator must be extended with knowledge about the foreign data source, so that it can decide in which cases it is possible and advantageous to combine multiple atomic units of access into a single call to the foreign data source.

In our case, where the foreign data source is a relational database, we can take advantage of the fact that on a syntactic level, predicates in the DRC are a subset of the predicates allowed in the object calculus used here. Access to the relational database is therefore represented in the Translator query plan with DRC predicates. The calculus optimizer of the Translator treats these predicates just like any other object calculus predicates, whereas the algebra generator is extended with knowledge of how to combine DRC predicates to replace them with SQL calls to the relational database.

4.2.1 DRC predicates

The object calculus representation of a query to the Translator will contain specially marked predicates representing access to the relational database. The semantics of these DRC predicates is defined as follows:

a DRC predicate $*r(x_1, \ldots, x_n)$ is true if and only if there is a tuple $< x_1, \ldots, x_n >$ in the relation named r in the relational database.

(End of definition)

All functions in the object view which require access to the relational database are defined in terms of one or more DRC predicates. During object view resolution, predicates using such a function are replaced by their definitions. For example[15]:

$$salary_{employee \rightarrow integer}(e) = sal \;\Leftrightarrow$$
$$\ldots \;\wedge\; *employee(enr, \_, sal, \_)$$

We now return to example query (Q2). The object calculus representation of this query will contain two DRC predicates[16]:

$$\{\, n \mid \ldots \;\wedge\; *employee(enr, n, \_, \_) \wedge$$
$$\ldots \;\wedge\; *employee(enr, \_, 15000, \_) \;\wedge\; \ldots \,\},$$

which in the executable query plan will have been replaced with the desired, single, SQL query:

```
select name from employee
  where salary=15000
```

4.3 Object identity

A major difference between relational and OO databases is that relational databases are *value based*, whereas OO databases are *identity based*. The reason for calling OO databases 'identity based' is that objects have an existence independent of the values of their attributes. Each object is uniquely identified by an OID which can always be used to refer to it. In contrast, if two tuples in a relational database have identical values for all attributes, the tuples are considered identical. This is usually handled by having a set of attributes (the primary key) whose values are always different for different real-world objects[17].

In an object view of a relational database, there will be a correspondence between primary key values in the relational database and OIDs in the view. The Translator must generate OIDs which correspond to the different primary key values and guarantee that a primary key value is mapped to the same OID each time it is accessed. This is a general problem for object views (Abiteboul and Bonner 1991). Suppose, for example, that an application issues a query which returns an OID (let us call this *:obj*). The application disconnects from the Translator but maintains the reference to :obj. The next time the application connects to the Translator, it issues a query which retrieves some property of :obj. Now, the Translator must map :obj to the same primary key value as when it was retrieved[18].

Two approaches to generation and maintenance of OIDs in object views can be distinguished; *algorithmic generation* of OIDs, and the use of OID *mapping tables*.

[15] The complete definition of this function is given in Sect. 4.3.1.
The variable '_' can be read as "don't care". For readability, we use this notation for variables that only occur once in the query plan.

[16] A complete description of the query processing steps for this query is given in appendix D.

[17] Most commercial relational databases do not enforce tuple uniqueness. See Sect. 4.1.

[18] In Kemp et al. (1994), OIDs are stored in the relational database. Extra 'OID columns' are added for this purpose. We want to avoid this approach, since it assumes permission to modify the relational database.

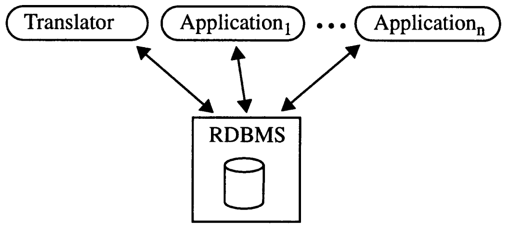


Fig. 15. The Translator works in parallel with existing applications

Algorithmic generation of OIDs means that the OID is somehow calculated based on the value of the primary key. One way to guarantee that different values always generate different OIDs is to represent OIDs by a concatenation of the relation name and the primary key value. This is proposed as a plausible implementation in the Pegasus project (Ahmed et al. 1993).

In the mapping tables approach (Fahl 1994), which is used in AMOS, there is no mathematical correspondence between the OID and the primary key value. OIDs for mapped objects are generated dynamically the first time they are needed and are thereafter maintained by the Translator. The mapping between OIDs and primary key values is stored in internal tables in the Translator.

The approach to OID management described in this paper does not depend on whether the algorithmic or mapping tables approach is used.

The mapping between OIDs and primary key values is modelled with system-generated functions called *oid_map* functions.

Since the Translator provides an object view of an *existing* relational database, it may have to coexist with other applications of that database. This is illustrated in Fig. 15. This makes it not feasible for the Translator to keep locks on all relations from which OIDs have been generated, since that would soon end up locking large parts of the database from other applications. Transactions in the relational database started by the Translator can therefore be characterized as 'access and commit immediately'[19]. This means that oid_map functions must access the relational database to check that the primary key value is still there, since it may have been deleted by another application.

The oid_map functions are defined in terms of DRC predicates and another kind of system-generated functions called *oid_translate* functions. This avoids that relational database access is embedded within the code of the oid_map functions. The oid_translate functions translate between OIDs and primary keys but do not access the relational database.

Section 4.3.1 describes oid_map functions and Sect. 4.3.2 describes oid_translate functions. Section 4.3.3 discusses the semantics when primary key values for which OIDs have been generated are deleted from the relational database.

For ease of presentation, we will only consider primary keys consisting of a single attribute. This can easily be extended to primary keys consisting of multiple attributes.

[19] We assume that the relational database *schema* used when defining the mapping to the object view is not changed.

4.3.1 oid_map functions

The oid_map function is overloaded and there is one resolvent function for each mapped type. An oid_map function takes a mapped object as argument and returns the primary key value that the object is mapped to. Let *MT* be a mapped type, *REL* be the relation that MT is mapped to, and *LT* be the literal type corresponding to the domain of the primary key attribute of REL. For each MT, the system generates the function

$oid\_map_{MT \rightarrow LT}$

For example:

$oid\_map_{employee \rightarrow integer}$

$oid\_map_{salesman \rightarrow integer}$

All functions in the object view that reference a mapped type in their signature are implemented as derived functions which make a call to the oid_map function. For example, the $salary_{employee \rightarrow integer}$ function is defined in terms of the $oid\_map_{employee \rightarrow integer}$ function and a DRC predicate:

$$\begin{aligned} &salary_{employee \rightarrow integer}(e) = sal \;\Leftrightarrow \\ &\quad oid\_map_{employee \rightarrow integer}(e) = enr \;\wedge \\ &\quad *employee(enr, \_, sal, \_) \end{aligned}$$

We assume that the semantics for oid_map functions implies that a result value must be present in the relational database in order to be returned. This means that apart from translating between OIDs and primary key values, an oid_map function must access the relational database. This is true regardless of whether it is used in the forward or backward (inverse) direction.

As was discussed in Sect. 4.2, good optimization requires that all relational database access is represented explicitly in query plans. We therefore define all oid_map functions as derived functions:

$$\begin{aligned} &oid\_map_{MT \rightarrow LT}(OBJ) = VAL \;\Leftrightarrow \\ &\quad oid\_translate_{mgmt(MT) \rightarrow LT}(OBJ) = VAL \;\wedge \\ &\quad *REL(\ldots, VAL, \ldots) \end{aligned}$$

For example:

$$\begin{aligned} &oid\_map_{employee \rightarrow integer}(e) = enr \;\Leftrightarrow \\ &\quad oid\_translate_{employee \rightarrow integer}(e) = enr \;\wedge \\ &\quad *employee(enr, \_, \_, \_) \end{aligned}$$

$$\begin{aligned} &oid\_map_{salesman \rightarrow integer}(s) = enr \;\Leftrightarrow \\ &\quad oid\_translate_{employee \rightarrow integer}(s) = enr \;\wedge \\ &\quad *salesman(enr, \_, \_) \end{aligned}$$

The DRC predicate ensures that the result value is a primary key in the relation that MT is mapped to. The oid_translate function handles the translation between OIDs and primary key values.

4.3.2 oid_translate functions

Just like the oid_map functions, an oid_translate function takes a mapped object as argument and returns the primary key value that the object is translated to. However, unlike oid_map functions, oid_translate functions do not involve access to the relational database. They are only concerned with the translation between OIDs and primary key values. The oid_translate function is overloaded and there is one resolvent function for each mgmt:

$oid\_translate_{MGMT \rightarrow LT}$

For example:

$oid\_translate_{employee \rightarrow integer}$

Oid_translate functions are only defined for mgmts. Subtypes to these types inherit the oid_translate function.

4.3.3 Deletion semantics

Since the Translator does not keep locks on relations from which OIDs have been generated, primary key values in these relations may be deleted by other applications. This section discusses possible semantics for the object view when this happens. In the following discussion, *:obj* is a mapped object and *MT* is the mapped type of which :obj is a direct instance.

Let us first consider the case when the primary key value is deleted from a relation which is mapped to a type which is not an mgmt. This causes no problems – the only thing that happens is that the type membership for :obj changes. Instead of being a direct instance of MT, it will be a direct instance of the supertype of MT. For example, if the tuple where enr=314 is deleted from the salesman relation, the object :e1 will no longer be an instance of the type salesman, but it will still be an instance of the type employee.

The problems start when (1) MT is an mgmt, and (2) some application has a handle to :obj. Suppose, for example, that some application has issued a query against the object view which returned the object :e1, that the application keeps a reference to this object, and that later the tuple where enr=314 is deleted from the employee relation. The question is what should happen when the application issues a new query which involves the object :e1, for example:

```
select salary(:e1)
```

In the current prototype, a query like this will simply return an empty result. The Translator will first execute

$oid\_translate_{employee \rightarrow integer}(:e1)$

to get the primary key value (314) that :e1 is mapped to. It will then send the following SQL query to the relational database:

```
select salary where enr=314
```

Since there is no tuple where enr=314, the initial query will not return anything.

Other semantics are conceivable for a situation like this. The Translator could notify the application that the reference to :obj is obsolete, rather than just return an empty result for queries like the one above. The notification could either be performed the first time the application uses :obj in a query, or as soon as the primary key value is deleted. The latter case would require an active database mechanism (Hanson and Widom 1993; Sköld and Risch 1996) or some kind of monitoring (Risch 1989) of the relational database.

A related question is what should happen if the primary key value is added to the relational database again. In the current prototype, the primary key value is mapped to the same OID and applications are unaffected by the fact the value was deleted for a period of time (unless they tried to use it while it was absent). This may be fine for some domains, but would be dangerous if primary key values could be re-used for different real-world objects. Employee numbers may, in fact, be an example of this if they are re-used when employees quit.

4.4 Type membership tests

Query plans in OO databases often contain tests of the *instance-of relationship*, i.e., which objects are instances of which types. We will refer to these tests as *type membership tests*. They may, for example, be used to retrieve all instances of a type or to test whether an object is an instance of a particular type.

Consider, for example, query (Q1) from Sect. 3.4. The $Employee(m)$ predicate in the object calculus representation of this query constrains the type membership of the variable m. In the object algebra translation to the left in Fig. 12, the $Employee(m)$ predicate from the calculus has resulted in the Employee node at the bottom, generating the set of all employee objects.

In an OO database, the relationship between objects and types (which objects are instances of which types) can be specially represented directly in the database, for example, by encoding the type information directly in the OID. This makes it possible to implement type membership tests very efficiently.

In an object view of a relational database, the relationship between objects and types depends on the state of the relational database[20]. Definition:

A mapped object OBJ is an instance of a mapped type TP if:

(a) OBJ is mapped to a primary key value that occurs in the relation that TP is mapped to, and
(b) $mgmt(TP) = mgmt(direct\_instance\_of(OBJ))$.

(End of definition)

Condition (b) is necessary to handle cases where a primary key value occurs in two relations which are mapped to types with different mgmts. For example, an employee object which is mapped to the primary key value enr=314 is not an instance of the type department, even if there is a department with the primary key value dnr=314.

Type membership tests require access to the relational database. This makes these tests very expensive, and it is essential that they are handled in an efficient way by the optimizer. The relational database access required by type membership tests should be combined with the other relational database access in the query plan. For example, consider the following AMOSQL query:

```
select salary(s) for each           (Q3)
  salesman s
```

A normal OO query plan would start by retrieving all salesman objects and then applying the salary function to these objects. If this strategy was followed for an object view of a relational database, this would imply execution of the query

```
select enr from salesman
```

to retrieve the primary key values for all salesman objects and then for each of these values (say X) executing queries of the form

```
select salary from employee where enr=X
```

Clearly, this is a non-optimal execution strategy. An optimal execution strategy results in a single SQL query:

```
select salary
from employee, salesman
where employee.enr=salesman.enr
```

The problem is that type membership tests require access to the relational database, and that this access is embedded in the code implementing these tests.

To resolve this problem, observe that the definition of the instance-of relationship for mapped objects and types can be expressed as follows:

$$TP(OBJ) \Leftrightarrow$$
$$\quad oid\_translate_{mgmt(TP)\rightarrow LT}(OBJ) = VAL \ \wedge$$
$$\quad *REL(\ldots, VAL, \ldots)$$

For example:

$$Employee(e) \Leftrightarrow$$
$$\quad oid\_translate_{employee\rightarrow integer}(e) = enr \ \wedge$$
$$\quad *Employee(enr, \_, \_, \_)$$

$$Salesman(s) \Leftrightarrow$$
$$\quad oid\_translate_{employee\rightarrow integer}(s) = enr \ \wedge$$
$$\quad *Salesman(enr, \_, \_)$$

This assumes that condition (b) in the definition above is guaranteed by the $oid\_translate$ predicate. For example, assume that :d is a department object which is mapped to the primary key value 314. Then the function call $oid\_translate_{employee\rightarrow integer}(:d)$ should return an empty result, and the function call $oid\_translate^{-1}{}_{integer\rightarrow employee}(314)$ should return the employee object :e1 but not the department object :d.

[20] Unless the extent of mapped types is assumed to be stable, i.e., no tuples are added or removed from relations corresponding to mapped types. In that case, the extent of a mapped type can be materialized, once and for all, in the Translator. This assumption is made in Kemp et al. (1994).

Substitution of $TP(OBJ)$ predicates according to the above rule is a part of the object view resolution phase. Again, consider AMOSQL query (Q3). The final calculus representation of this query after object view resolution and calculus optimization is as follows[21]:

$$\{\ sal\ |\ *salesman(enr, \_, \_) \land *employee(enr, \_, sal, \_)\ \}$$

Which results in a single SQL query against the relational database:

```
select salary
from employee, salesman
where employee.enr=salesman.enr
```

4.4.1 Remaining problems with type membership tests

A more general way to describe type membership tests is with the $instance\_of_{type\rightarrow object}$ function. It takes a type object as argument and returns the objects that are instances of that type (direct or by generalization). Using this notation, a predicate of the form $TP(OBJ)$ would be written

$$OBJ = instance\_of_{type\rightarrow object}(TP)$$

For example:

$$e = instance\_of_{type\rightarrow object}(:typeEmployee)$$

When the type is known at compile-time, the instance_of function will be used in the forward direction. This was the case in all examples in the previous section, and can be seen as the normal case. However, some unusual queries require the instance_of function to be used in the backward direction (i.e., the inverse).

One example of this is a metadata query like the following:

```
select t                               (Q4)
for each type t
where :e1 = instance_of(t)
```

Other examples are queries that require late binding. For example, suppose the salary function is overridden for the salesman type. Then, a query like

```
select salary(e) for each              (Q5)
  employee e
```

requires late binding of the salary function. For each employee object (E), the inverse of the $instance\_of$ function must be used to retrieve the types that E is an instance of. This decides whether the $salary_{employee\rightarrow integer}$ or $salary_{salesman\rightarrow integer}$ resolvent should be applied to E.

Recall the substitution rule for TP(OBJ) predicates:

$$TP(OBJ) \Leftrightarrow oid\_translate_{mgmt(TP)\rightarrow LT}(OBJ) = VAL \land *REL(\ldots, VAL, \ldots)$$

[21] The query processing details for this query are given in appendix E.

Using the instance_of function, this can be written as:

$$OBJ = instance\_of_{type\rightarrow object}(TP) \Leftrightarrow oid\_translate_{mgmt(TP)\rightarrow lt}(OBJ) = VAL \land *REL(\ldots, VAL, \ldots)$$

This substitution rule can only be applied when TP is known at compile-time. For queries (Q4) and (Q5) above, TP is bound at run-time, which means that the execution plan contains calls to the inverse of instance_of.

The $instance\_of^{-1}{}_{object\rightarrow type}$ function takes an object as argument and returns the types that the object is an instance of. The semantics for an $instance\_of^{-1}{}_{object\rightarrow type}$ (OBJ) function call is as follows: Calculate the primary key value (v) that corresponds to OBJ. Let TREE be the subtree of the type tree that has $mgmt(direct\_instance\_of(OBJ))$ as its root. Traverse TREE top-down and for each type TP that is a node of TREE do the following: let rel be the relation that TP is mapped to. Check if v is a primary key value in rel. If it is, TP is one of the types that should be returned. If it is not, do not traverse the subtree having TP as its root any further.

For example, query (Q4) will result in the following SQL queries[22]:

```
select 'T' from employee where enr=314
select 'T' from secretary where enr=314
select 'T' from salesman where enr=314
```

As this example shows, queries that require the instance_of function to be used in the backward direction are very expensive. Furthermore, global optimization is impossible since relational database access is embedded within the code of the instance_of function.

Our approach is based on full query translation and no view materialization. Clearly, queries like the ones in this section would benefit greatly if the instance-of relationship was materialized. However, view materialization introduces new problems such as keeping the materialization up-to-date (Gupta and Mumick 1995). Which approach to take should be based on the nature of data (update frequency) and applications (query types).

5 Query optimization

Query optimization is performed in two steps (see Fig. 4): *calculus optimization* and *algebra optimization*. Calculus optimization (Sect. 5.1) concerns rewrite rules to simplify the declarative calculus representation of the query. Section 5.1.1 describes the technique of simplification using key information. Section 5.1.2 presents a rewrite rule which takes advantage of the semantics of the oid_translate function. Algebra optimization is the process of finding the cheapest algebraic representation of the query. This is discussed in Sect. 5.2.

[22] The queries are used to test whether a certain value is in the database or not. It does not matter *what* is returned, only that *something* is returned. Hence "select 'T'. . . "

5.1 Calculus optimization

5.1.1 Simplification using key information

Information about keys[23] can be used to simplify calculus expressions. This is the case when two predicates have the same predicate symbols, have the same constants/variables in the key attributes, and there are no conflicts between constants in the non-key attributes. For example, consider the following calculus expression:

$$\{ \, sal_1 \mid salary_{employee \to integer}(e) = sal_1 \wedge salary_{employee \to integer}(e) = sal_2 \wedge foo(sal_2) \, \}$$

The argument to the salary function is a key since employees can only have one salary (the function is single-valued). Since the two salary predicates have the same variable (e) as argument, they can be unified and replaced by a single predicate, provided that the resulting substitution (sal_2 should be replaced by sal_1) is applied to the rest of the predicates:

$$\{ \, sal_1 \mid salary_{employee \to integer}(e) = sal_1 \wedge foo(sal_1) \, \}$$

These simplifications are often needed during query processing due to the definition of the object view. Consider for example the following query:

```
select name(e), salary(e)          (Q6)
for each employee e
```

This is translated to the following calculus expression:

$$\{ \, n, sal \mid Employee(e) \wedge sal = salary_{employee \to integer}(e) \wedge n = name_{employee \to integer}(e) \, \}$$

Object view resolution gives the following, which is the input to the calculus optimizer:

$$\{ \, n, sal \mid oid\_translate_{employee \to integer}(e) = enr_1 \wedge *employee(enr_1, \_, \_, \_) \wedge oid\_translate_{employee \to integer}(e) = enr_2 \wedge *employee(enr_2, \_, sal, \_) \wedge oid\_translate_{employee \to integer}(e) = enr_3 \wedge *employee(enr_3, n, \_, \_) \, \}$$

Since all three oid_translate function calls take the same variable (e) as argument, and since the oid_translate function is single-valued, they can be unified. Each occurrence of variables enr_2 and enr_3 are replaced by enr_1:

$$\{ \, n, sal \mid oid\_translate_{employee \to integer}(e) = enr_1 \wedge *employee(enr_1, \_, \_, \_) \wedge *employee(enr_1, \_, sal, \_) \wedge *employee(enr_1, n, \_, \_) \, \}$$

Now, the $*employee$ predicates have the same variable (enr_1) in the key position[24]. Unification gives:

$$\{ \, n, sal \mid oid\_translate_{employee \to integer}(e) = enr_1 \wedge *employee(enr_1, n, sal, \_) \, \}$$

5.1.2 Removal of unnecessary OID translations

It is sometimes possible to remove predicates on the form $oid\_translate_{MGMT \to LT}(OBJ) = VAL$ from the query plan without affecting the semantics of the query. We define the rule for this as follows:

If: The argument (i.e., the OID) to the $oid\_translate$ function is a variable which does not occur in any other predicate of the calculus expression and is not one of the result variables of the query.

Then: The predicate containing the $oid\_translate$ function call can be removed from the calculus expression.

Before we motivate why the removal rule can be applied, observe that the only case we need to discuss is when the function is used in the backward direction (i.e., the inverse). If the function is used in the forward direction, the argument variable has been bound by some other predicate and the precondition of the rule is not satisfied.

A predicate of the form $oid\_translate_{MGMT \to LT}(OBJ) = VAL$ for which the precondition holds can be removed from the query plan, since $oid\_translate^{-1}$ is a single-valued function that never fails. Its only purpose is to compute the OID which corresponds to a certain primary key value. Thus, if the computed OID is never used, then the whole translation can be omitted.

We return to the example query from the previous section (5.1.1). Since the argument to the $oid\_translate$ function (e) does not occur elsewhere in the calculus expression, the predicate can be removed:

$$\{ \, n, sal \mid *employee(enr_1, n, sal, \_) \, \}$$

5.2 Algebra optimization

The object calculus that is the input to the algebra generator contains DRC predicates that represent access to the relational database. During algebra generation the query processor creates temporary functions which replace the DRC

[23] Or more generally, information about functional dependencies (Qian and Raschid 1995).

[24] Information about keys in the relational database is collected during schema translation.

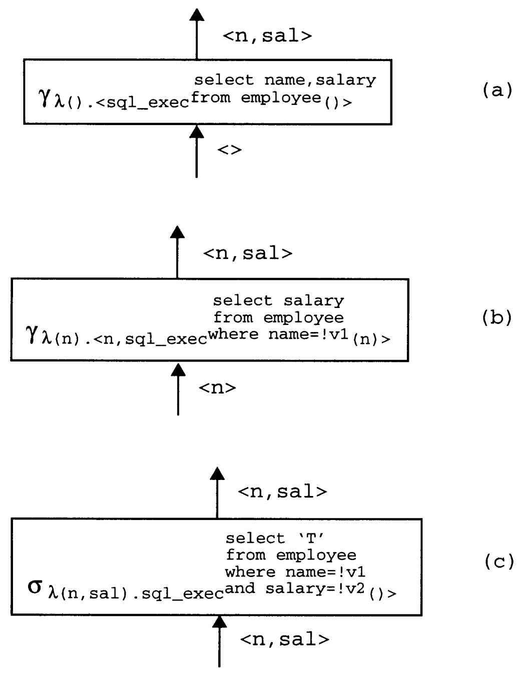


Fig. 16. Query tree nodes where zero, one, and two of the variables n and sal are bound

predicates. These temporary functions send SQL queries to the relational database.

The temporary functions are called sql_exec^q where q is an SQL query. The query q may be parametrized with variables on the form $!v_x$, where x=1, 2, 3 etc. A parametrized query is instantiated at execution time. The first argument to the sql_exec^q function replaces the variable $!v_1$, the second replaces $!v_2$ etc. The sql_exec^q function call returns the bag of tuples that is the result of the SQL query. An example will help to illustrate this. Consider the following calculus expression:

$$
\begin{aligned}
&\{\ n, sal\ |\\
&\quad foo(17) = n\ \wedge\\
&\quad fie(42) = sal\ \wedge\\
&\quad *employee(\_, n, sal, \_)\ \}
\end{aligned}
$$

The DRC predicate $*employee$ is replaced with different sql_exec^q function calls depending on which, if any, of the variables n and sal are bound, i.e., depending on the position of the sql_exec^q function call in the query tree. Figure 16 shows the resulting query tree nodes when zero, one, and two of the variables are bound. If both variables are bound, the SQL query works as a boolean test. In that case, the node is a select (σ) node and not a generate (γ) node.

When the calculus expression contains more than one DRC predicate, different combinations of these predicates may be grouped together to be replaced by a single sql_exec^q function call. The algebra optimizer must be extended with knowledge of how to find the optimal grouping of DRC predicates.

When two DRC predicates have a common variable, this represents a join between two relations. For example, consider the following query:

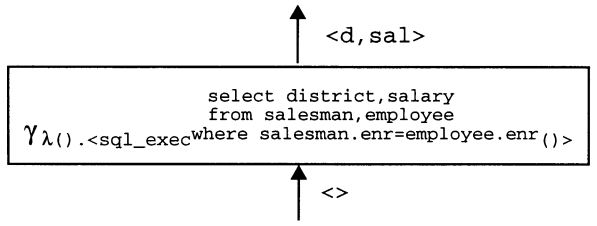


Fig. 17. Two DRC predicates are replaced by a single sql_exec^q function call

```
select district(s), salary(s)       (Q7)
for each salesman s
```

This is translated to the following optimized calculus expression:

$$
\begin{aligned}
&\{\ d, sal\ |\\
&\quad *salesman(enr, d, \_)\ \wedge\\
&\quad *employee(enr, \_, sal, \_)\ \}
\end{aligned}
$$

Figure 17 shows how the two DRC predicates can be replaced by a single sql_exec^q function call.

When two DRC predicates *do not* have any common variables, replacing them with a single sql_exec^q function call would result in a 'cartesian product query' to the relational database.

To illustrate this, we will assume that the Translator for the company example contains the following local function[25]:

$recreation_{charstring \rightarrow charstring}$

The function takes a district as argument and returns the recreational activities that are possible in that district. We will assume the following extension for the function:

```
recreation('charlotte')='squash'
recreation('charlotte')='fishing'
recreation('charlotte')='skiing'
recreation('raleigh')='golf'
```

Now suppose that the following query is given to the Translator ('what salesmen work in a district where the hobby of some employee earning 25000 can be practiced')[26]:

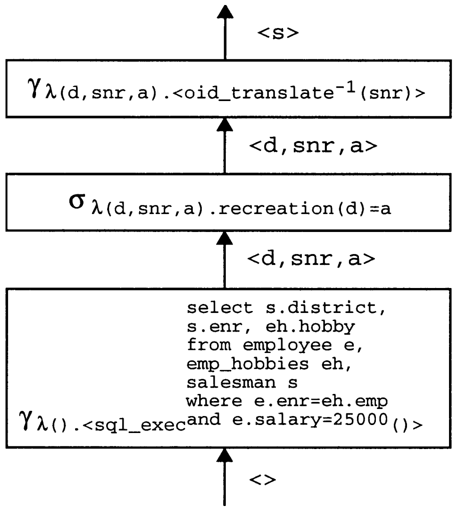
```
select s                              (Q8)
for each salesman s, employee e
where recreation(district(s))=hobby(e)
and salary(e)=25000
```

The input to the algebra generator will be the following calculus expression:

$$
\begin{aligned}
&\{\ s\ |\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = snr\ \wedge\\
&\quad recreation_{charstring \rightarrow charstring}(d) = a\ \wedge\\
&\quad *salesman(snr, d, \_)\ \wedge\\
&\quad *emp\_hobbies(enr, a)\ \wedge\\
&\quad *employee(enr, \_, 25000, \_)\ \}
\end{aligned}
$$

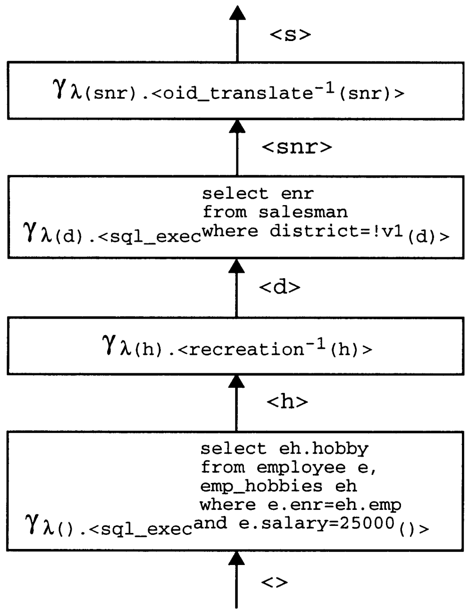
Figure 18 shows one of the possible execution plans where the three DRC predicates are replaced by a single sql_exec^q

[25] I.e., a function for which the extent is stored directly in the Translator.
[26] The query processing details for this query are given in appendix C.



<s>
γλ(d,snr,a).<oid_translate⁻¹(snr)>
<d,snr,a>
σλ(d,snr,a).recreation(d)=a
<d,snr,a>
γλ().<sql_exec select s.district, s.enr, eh.hobby from employee e, emp_hobbies eh, salesman s where e.enr=eh.emp and e.salary=25000 ()>
<>

Fig. 18. Execution plan for query (Q8) with *one* sql_exec^q function call



<s>
γλ(snr).<oid_translate⁻¹(snr)>
<snr>
γλ(d).<sql_exec select enr from salesman where district=!v1 (d)>
<d>
γλ(h).<recreation⁻¹(h)>
<h>
γλ().<sql_exec select eh.hobby from employee e, emp_hobbies eh where e.enr=eh.emp and e.salary=25000 ()>
<>

Fig. 19. Execution plan for query (Q8) with *two* sql_exec^q function calls

function call. A single query will be sent to the relational database. The query will join the employee and emp_hobbies relations and then create the cartesian product of this and the salesman relation.

Figure 19 shows one of the possible execution plans where the DRC predicates are replaced by two sql_exec^q function calls.

The first sql_exec^q function will be executed once and produce three output tuples: $<$'tennis'$>$, $<$'fishing'$>$, and $<$'golf'$>$. 'fishing' is a recreational activity in 'charlotte', and 'golf' is a recreational activity in 'raleigh', which means that the second sql_exec^q function will be executed two times:

```
select enr from salesman
where district='charlotte'

select enr from salesman
where district='raleigh'
```

The cost of an execution plan in the Translator depends on the relative costs of computations in the Translator, transmission costs, and the costs of processing the queries in the relational database. Ideally, the query optimizer of the Translator should estimate the cost of all possible execution plans and select the cheapest. This optimization problem is even more computationally intractable than regular query optimization, since DRC predicates may or may not be grouped together to be replaced by a single sql_exec^q function. To achieve an accurate estimation of the costs of the relational queries, the cost model of the relational database system will have to be simulated in the Translator. Note that the statistics and cost model parameters of the relational database system may not be available. In that case, they could be estimated by running a well-chosen set of test queries (Du et al. 1992).

In the current prototype implementation we use the heuristic rule to generate as few SQL queries as possible, but never cartesian product queries. This reduces the search space significantly and avoids the worst cases. The costs of the remaining query plans are then estimated and the cheapest is selected for execution. The estimated cost of an algebra operation involving an sql_exec^q function call is set to a fixed, very high value. The cardinality of the result of an algebra operation involving an sql_exec^q function call is estimated based on the number of bound and free variables. For example, algebra node (a) in Fig. 16 can be expected to greatly increase the cardinality since both variables (n and sal) are free. Algebra node (c) can be expected to decrease the cardinality, since both variables are bound.

6 Summary and future work

We have presented an approach to object view management for relational databases. The software component implementing the object view can store its own data and methods. This means that queries can combine local data residing in the object view with data retrieved from the relational database.

To simplify the mapping between the relational database and the object view, we assume that subtype/supertype relationships are represented in a particular way in the relational database. Relational databases represented in this way are said to be DM to a particular object view. When a relational database *is not* DM to a desired object view, the first step is to define an external relational schema that *is*.

Global optimization requires that all relational database access is represented explicitly in object view query plans. We use DRC predicates for this. The object calculus representation of a query may contain DRC predicates. All functions in the object view which require access to the relational database are defined as derived functions. During object view resolution, they are replaced by their definitions, which include one or more DRC predicates.

The mapping between OIDs and primary key values is modelled with system-generated *oid_map* functions. To avoid that relational database access is hidden within the code of oid_map functions, they are defined as derived in terms of DRC predicates and *oid_translate* functions. DRC predicates handle the relational database access and oid_translate functions handle the 'pure' OID translation functionality.

The relationship between objects and types is modelled by the *instance_of* function. When the argument to an instance_of function call is a mapped type that is known at compile-time, the function call can be replaced by a DRC predicate and an oid_translate function call. This avoids that relational database access is hidden within the code of the instance_of function.

The calculus optimization techniques of *simplification using key information* and *removal of unnecessary oid_translate predicates* are used to simplify query plans and avoid unnecessary OID translations. The query optimizer must then decide how to replace DRC predicates with actual calls to the relational database. DRC predicates may be combined in many different ways which increases the complexity of query optimization. Ideally, the optimizer should estimate the cost of all possible execution plans and select the cheapest. The cost depends on the relative costs of computations in the object view, transmission costs, and the costs of processing the queries in the relational database. We currently use the heuristic rule to generate as few SQL queries as possible, but never cartesian product queries. This reduces the search space and avoids the worst cases.

6.1 Future work

As discussed in Sect. 4.1, the first step of the mapping procedure is to define an external relational schema that is DM (DM) with respect to a particular object view. Unfortunately, current relational view definition languages are not general enough to allow all types of mappings (for an example, see Sect. 4.1). Two different solutions are possible. (a) Extend relational view definition languages, so that a DM external schema can be defined for all types of relational schemas, or (b) add constructs to the object view definition language, so that all relational schemas can be directly mapped to the desired object view, including those for which the relational view mechanism is not powerful enough to define the DM external schema.

The heuristic rules used for algebra optimization are not adequate in the general case. The selection of execution strategy should be based on real cost estimates. To estimate the costs of relational queries, the cost model of the relational database system has to be simulated in the object view.

The approach should be tested on existing relational databases. Scalability regarding database and schema size is not a problem, since our approach is based on non-materialized views and query translation. But realistically sized databases raise the demands on query optimization techniques, and our heuristics may prove inadequate. It is also essential to study the effects that bad or unusual data design has on the usability of our approach. For example, how frequent it is that DM views cannot be defined.

The approach should also be generalized for complex queries involving, for example, aggregation and grouping. Queries requiring late binding are currently very expensive, as discussed in Sect. 4.4.1. Materialization of the instance-of relationship may be necessary in order to achieve good performance for late binding queries. This introduces new problems such as view maintenance.

Acknowledgements The authors wish to thank the anonymous referees for their substantial comments on an earlier version of the paper. This research was supported by the Swedish Research Cousil for Engineering Sciences (NUTEK) and the Center for Industrial Information Technology (CENIIT).

Appendix A

The semantics of the γ operator is defined as follows. Let Y be the input bag of tuples of objects:

$$Y = \{| < y_{1_1}, \ldots, y_{1_n} >, \ldots, < y_{m_1}, \ldots, y_{m_n} > |\}$$

Then,

$$\gamma_{\lambda(x_1,\ldots,x_n).<x_1,\ldots,x_n,\Theta(x_1,\ldots,x_n)>}(Y) = \bigcup_{i=1\ldots m}^{\text{bag}} (\{| < y_{i_1}, \ldots, y_{i_n} > |\} \times \Theta(y_{i_1}, \ldots, y_{i_n}))$$

The semantics of the σ operator is defined as follows (where σ^{rel} is the relational algebra selection operator with bag semantics):

$$\sigma_{\lambda(x_1,\ldots,x_n).\Theta(x_1,\ldots,x_n)=z}(A) = \pi_{\lambda(x_1,\ldots,x_n,y).<x_1,\ldots,x_n>}(C)$$

where

$$C = \sigma^{\text{rel}}{}_{\lambda(x_1,\ldots,x_n,y).y=z}(B)$$

and

$$B = \gamma_{\lambda(x_1,\ldots,x_n).<x_1,\ldots,x_n,\Theta(x_1,\ldots,x_n)>}(A)$$

Appendix B

Object view resolution rules in the company example:

$$Employee(e) \Leftrightarrow oid\_translate_{employee\rightarrow integer}(e) = enr \wedge *employee(enr, \_, \_, \_)$$

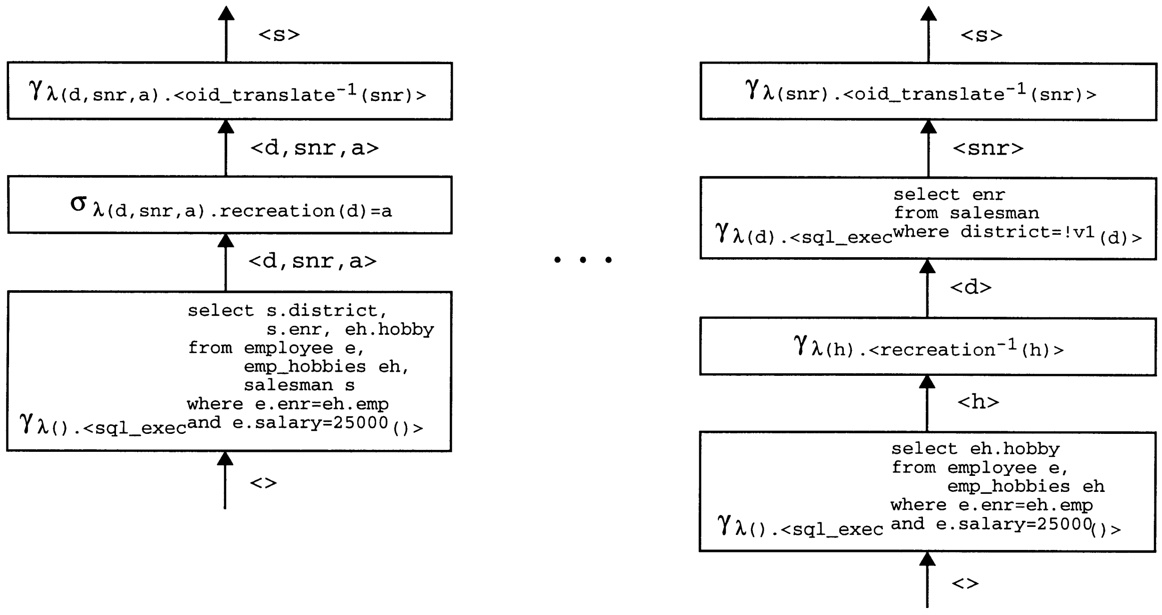
$$Salesman(s) \Leftrightarrow oid\_translate_{employee\rightarrow integer}(s) = enr \wedge *salesman(enr, \_, \_)$$

$$Secretary(s) \Leftrightarrow oid\_translate_{employee\rightarrow integer}(s) = enr \wedge *secretary(enr, \_)$$

$$enr_{employee\rightarrow integer}(e) = enr \Leftrightarrow oid\_translate_{employee\rightarrow integer}(e) = enr \wedge *employee(enr, \_, \_, \_)$$

$$name_{employee\rightarrow charstring}(e) = n \Leftrightarrow oid\_translate_{employee\rightarrow integer}(e) = enr \wedge *employee(enr, n, \_, \_)$$

$$salary_{employee\rightarrow integer}(e) = sal \Leftrightarrow oid\_translate_{employee\rightarrow integer}(e) = enr \wedge *employee(enr, \_, sal, \_)$$



```
<s>
γλ(d,snr,a).<oid_translate^-1(snr)>
<d,snr,a>
σλ(d,snr,a).recreation(d)=a
<d,snr,a>
γλ().<sql_exec select s.district,
                        s.enr, eh.hobby
                 from employee e,
                      emp_hobbies eh,
                      salesman s
                 where e.enr=eh.emp
                 and e.salary=25000 ()>
<>
```

• • •

```
<s>
γλ(snr).<oid_translate^-1(snr)>
<snr>
γλ(d).<sql_exec select enr
                from salesman
                where district=!v1 (d)>
<d>
γλ(h).<recreation^-1(h)>
<h>
γλ().<sql_exec select eh.hobby
               from employee e,
                    emp_hobbies eh
               where e.enr=eh.emp
               and e.salary=25000 ()>
<>
```

Fig. 20. Algebraic translations of the example query in Appendix C

$$\begin{aligned}
&manager_{employee \rightarrow employee}(e) = m \;\Leftrightarrow\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(m) = mnr \;\wedge\\
&\quad *employee(enr, \_, \_, mnr)
\end{aligned}$$

$$\begin{aligned}
&hobby_{employee \rightarrow charstring}(e) = h \;\Leftrightarrow\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr \;\wedge\\
&\quad *emp\_hobbies(enr, h)
\end{aligned}$$

$$\begin{aligned}
&typingspeed_{secretary \rightarrow integer}(s) = ts \;\Leftrightarrow\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = enr \;\wedge\\
&\quad *secretary(enr, ts)
\end{aligned}$$

$$\begin{aligned}
&district_{salesman \rightarrow charstring}(s) = d \;\Leftrightarrow\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = enr \;\wedge\\
&\quad *salesman(enr, d, \_)
\end{aligned}$$

$$\begin{aligned}
&sales_{salesman \rightarrow integer}(s) = sls \;\Leftrightarrow\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = enr \;\wedge\\
&\quad *salesman(enr, \_, sls)
\end{aligned}$$

Appendix C

AMOSQL query:

```
select s
for each salesman s, employee e
where recreation(district(s))=hobby(e)
and salary(e)=25000
```

Object calculus:

$$\begin{aligned}
&\{\, s \mid\\
&\quad Salesman(s) \;\wedge\\
&\quad Employee(e) \;\wedge\\
&\quad recreation_{charstring \rightarrow charstring}(d) = a \;\wedge\\
&\quad district_{salesman \rightarrow charstring}(s) = d \;\wedge\\
&\quad hobby_{employee \rightarrow charstring}(e) = a \;\wedge\\
&\quad salary_{employee \rightarrow integer}(e) = 25000 \,\}
\end{aligned}$$

Object calculus after object view resolution (see Appendix B):

$$\begin{aligned}
&\{\, s \mid\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = snr \;\wedge\\
&\quad *salesman(snr, \_, \_) \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr \;\wedge\\
&\quad *employee(enr, \_, \_, \_) \;\wedge\\
&\quad recreation_{charstring \rightarrow charstring}(d) = a \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = snr_2 \;\wedge\\
&\quad *salesman(snr_2, d, \_) \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr_2 \;\wedge\\
&\quad *emp\_hobbies(enr_2, a) \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr_3 \;\wedge\\
&\quad *employee(enr_3, \_, 25000, \_) \,\}
\end{aligned}$$

Object calculus after simplification using key information (calculus optimization phase 1):

$$\begin{aligned}
&\{\, s \mid\\
&\quad oid\_translate_{employee \rightarrow integer}(s) = snr \;\wedge\\
&\quad oid\_translate_{employee \rightarrow integer}(e) = enr \;\wedge\\
&\quad recreation_{charstring \rightarrow charstring}(d) = a \;\wedge\\
&\quad *salesman(snr, d, \_) \;\wedge\\
&\quad *emp\_hobbies(enr, a) \;\wedge\\
&\quad *employee(enr, \_, 25000, \_) \,\}
\end{aligned}$$

Object calculus after removal of unnecessary oid_translate predicates (calculus optimization phase 2):

$\{\ s\ |$
$\quad oid\_translate_{employee \to integer}(s) = snr\ \wedge$
$\quad recreation_{charstring \to charstring}(d) = a\ \wedge$
$\quad *salesman(snr, d, \_)\ \wedge$
$\quad *emp\_hobbies(enr, a)\ \wedge$
$\quad *employee(enr, \_, 25000, \_)\ \}$

Algebraic translations of the calculus expression are generated[27]. See Fig. 20.

The cheapest is selected for execution. See the rightmost query plan in Fig. 20.

Appendix D

AMOSQL query:

```
select name(e)
for each employee e
where salary(e)=15000
```

Object calculus:

$\{\ n\ |$
$\quad Employee(e)\ \wedge$
$\quad name_{employee \to charstring}(e) = n\ \wedge$
$\quad salary_{employee \to integer}(e) = 15000\ \}$

Object calculus after object view resolution (see Appendix B):

$\{\ n\ |$
$\quad oid\_translate_{employee \to integer}(e) = enr\ \wedge$
$\quad *employee(enr, \_, \_, \_)\ \wedge$
$\quad oid\_translate_{employee \to integer}(e) = enr_2\ \wedge$
$\quad *employee(enr_2, n, \_, \_)\ \wedge$
$\quad oid\_translate_{employee \to integer}(e) = enr_3\ \wedge$
$\quad *employee(enr_3, \_, 15000, \_)\ \}$

Object calculus after simplification using key information (calculus optimization phase 1):

$\{\ n\ |$
$\quad oid\_translate_{employee \to integer}(e) = enr\ \wedge$
$\quad *employee(enr, n, 15000, \_)\ \}$

Object calculus after removal of unnecessary oid_translate predicates (calculus optimization phase 2)

$\{\ n\ |$
$\quad *employee(\_, n, 15000, \_)\ \}$

The algebraic translation of the calculus expression is generated (there is only one translation). See Fig. 21.

[27] In the current prototype implementation, we use the heuristic rule to generate as few SQL queries as possible, but never cartesian product queries.

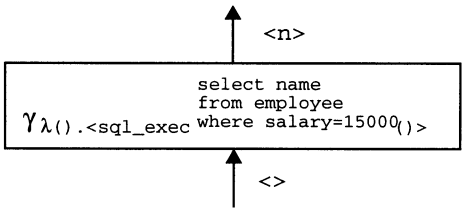


Fig. 21. Algebraic translation of the example query in Appendix D

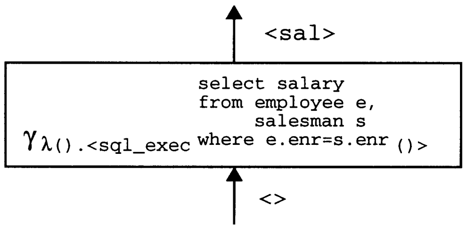


Fig. 22. Algebraic translation of the example query in Appendix E

Appendix E

AMOSQL query:

```
select salary(s) for each salesman s
```

Object calculus:

$\{\ sal\ |$
$\quad Salesman(s)\ \wedge$
$\quad salary_{employee \to integer}(s) = sal\ \}$

Object calculus after object view resolution (see Appendix B):

$\{\ sal\ |$
$\quad oid\_translate_{employee \to integer}(s) = enr\ \wedge$
$\quad *salesman(enr, \_, \_)\ \wedge$
$\quad oid\_translate_{employee \to integer}(s) = enr_2\ \wedge$
$\quad *employee(enr_2, \_, sal, \_)\ \}$

Object calculus after simplification using key information (calculus optimization phase 1):

$\{\ sal\ |$
$\quad oid\_translate_{employee \to integer}(s) = enr\ \wedge$
$\quad *salesman(enr, \_, \_)\ \wedge$
$\quad *employee(enr, \_, sal, \_)\ \}$

Object calculus after removal of unnecessary oid_translate predicates (calculus optimization phase 2):

$\{\ sal\ |$
$\quad *salesman(enr, \_, \_)\ \wedge$
$\quad *employee(enr, \_, sal, \_)\ \}$

Algebraic translations of the calculus expression are generated and the cheapest is selected for execution. See Fig. 22.

References

1. Abiteboul S, Bonner A (1991) Objects and views. In: Proc. ACM SIGMOD Conf. ’91, Denver, Colorado
2. Ahmed R, Albert J, Du W, Kent W, Litwin W, Shan M-C (1993) An overview of Pegasus. In: Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS ’93, Vienna, Austria
3. Albert J, Ahmed R, Ketabchi M, Kent W, Shan M-C (1993) Automatic importation of relational schemas in Pegasus. In: Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS ’93, Vienna, Austria
4. Barsalou T, Siambela N, Keller AM, Wiederhold G (1991) Updating relational databases thorough object-based views. In: Proc. ACM SIGMOD Conf. ’91, Denver, Colorado
5. Bertino E, Martino L (1991) Object-oriented database management systems: concepts and issues. IEEE Comput 24:4
6. Bright MW, Hurson AR, Pakzad SH (1992) A taxonomy and current issues in multidatabase systems. IEEE Computer 25:3
7. Chomicki J, Litwin W (1994) Declarative definition of object-oriented multidatabase mappings. In: Özsu MT, Dayal, Valduriez (eds): Distributed object management. Morgan Kaufmann, San Mateo, Calif., pp 375–392
8. Demuth B, Geppert A, Gorchs T (1994) Algebraic query optimization in the cooms structurally object-oriented database system. In: Freytag JC et al. (eds): Query processing for advanced database systems. Morgan Kaufmann, San Mateo, Calif., pp 121–142
9. Du W, Krishnamurthy R, Shan M-C (1992) Query optimization in heterogeneous DBMS. In: Proc. VLDB ’92, Vancouver, Canada
10. Elmasri R, Navathe SB (1989) Fundamentals of database systems. Benjamin/Cummings, Menlo Park, Calif.
11. Fahl G, Risch T, Sköld M (1993) AMOS – An architecture for active mediators. In: Proc. Int. Workshop on Next Generation Information Technologies and Systems, NGITS ’93, Haifa, Israel
12. Fahl G (1994) Object views of relational data in multidatabase systems. Licentiate Thesis, LiU-Tek-Lic 1994:32, Linköping University, Sweden
13. Fishman DH et al. (1989) Overview of the Iris DBMS. In: Kim W, Lochovsky FH (eds): Object-oriented concepts, databases and applications. ACM Press, Addison-Wesley, Reading, Mass., pp 219–250
14. Gray P (1984) Logic, algebra and databases. Ellis Horwood Ltd.
15. Gupta A, Mumick IS (1995) Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng Bull 18:2
16. Hanson EN, Widom J (1993) An overview of production rules in database systems. Knowl Eng Rev 8:2
17. Heiler S, Zdonik S (1990) Object views: extending the vision. In: Proc. of the 6th Int. Conf. on Data Engineering, L.A., California
18. Jarke M, Koch J (1984) Query optimization in database systems. Comput Surv 16:2
19. Johannesson P, Kalman K (1989) A method for translating relational schemas into conceptual schemas. In: Int. Conf. on Entity-Relationship Approach, Toronto, Canada
20. Kemp G, Iriarte J, Gray P (1994) Efficient access to fdm objects stored in a relational database. In: Proc. of the Twelfth British National Conference on Databases, BNCOD 12. New York: Springer, pp 170–186
21. Krishnamurthy R, Litwin W, Kent W (1991) Language features for interoperability of databases with schematic discrepancies. In: Proc. ACM SIGMOD Conf. ’91, Denver, Colorado
22. Landers T, Rosenberg R (1982) An overview of multibase. In: Schneider H-J (ed) Distributed Databases. North-Holland
23. Litwin W, Mark L, Roussopoulos N (1990) Interoperability of multiple autonomous databases. ACM Comput Surv 22:3
24. Litwin W, Ketabchi M, Krishnamurthy R (1991) First-order normal form for relational databases and multidatabases. SIGMOD RECORD 20:4
25. Litwin W, Risch T (1992) Main-memory-oriented optimization of oo queries using typed datalog with foreign predicates. IEEE Trans Knowl Data Eng 4:6
26. Lyngbaek P et al. (1991) OSQL: A Language for object databases. Technical Report, HP Labs., HPL-DTD-91-4
27. Markowitz VM, Markowsky JA (1990) Identifying extended entity-relationship object structures in relational schemas. IEEE Trans Software Eng 16:8
28. Navathe SB, Awong AM (1987) Abstracting relational and hierarchical data with a semantic data model. In: Proc. of the Sixth Int. Conf. on Entity-Relationship Approach, New York
29. Özsu MT, Blakeley JA (1994) Query processing in object-oriented database systems. In: Kim W (ed) Modern database management – object-oriented and multidatabase technologies. Addison-Wesley/ACM Press, Reading, Mass., pp 146–174
30. Qian X, Raschid L (1995) Query interoperation among object-oriented and relational databases. In: Proc. of the 11th Int. Conf. on Data Engineering, Taipei, Taiwan
31. Risch T (1989) Monitoring database objects. In: Proc. of the 15th International Conference on Very Large Data Bases, VLDB ’89, Amsterdam, The Netherlands
32. Saltor F, Castellanos M, Garcia-Solaco M (1991) Suitability of data models as canonical models for federated databases. SIGMOD RECORD 20:4
33. Shan M-C, Ahmed R, Davis J, Du W, Kent W (1995) Pegasus: a heterogeneous information management system. In: Kim W (ed) Modern Database Systems. ACM Press, Addison-Wesley, Reading, Mass., pp 664–682
34. Shaw GM, Zdonik SB (1990) A query algebra for object-oriented databases. In: Proc. of the 6th International Conference on Data Engineering, Los Angeles, California
35. Sheth AP, Larson JA (1990) Federated database systems for managing distributed, heterogeneous and autonomous databases. ACM Comput Surv 22:3
36. Shipman DW (1981) The functional data model and the data language DAPLEX. ACM Trans Database Sys 6:1
37. Sköld M, Risch T (1996) Using Partial Differencing for efficient monitoring of deferred complex rule conditions. In: Proceedings of the 12th Int’l Conf. on Data Engineering, New Orleans, Louisiana
38. Stonebraker M, Moore D (1996) Object-relational DBMSs: the next great wave. Morgan Kaufmann, San Mateo, Calif.
39. Straube DD, Özsu MT (1990) Queries and query processing in object-oriented database systems. ACM Trans Information Syst 8:4
40. Straube DD, Özsu MT (1995) Query optimization and execution plan generation in object-oriented database systems. IEEE Trans Knowl Data Eng 7:2, pp 210–227
41. Ullman JD (1988) Database and Knowledge-Base Systems, Volume I & II. Computer Science Press
42. Yan L-L, Ling T-W (1992) Translating relational schema with constraints into OODB schema. In: Proceedings of the Conference on Semantics of Interoperable Database Systems, Lorne, Victoria, Australia
43. Yu C, Zhang Y, Meng W, Kim W, Wang G, Pham T, Dao S (1995) Translation of object-oriented queries to relational queries. In: Proc. of the 11th Int. Conf. on Data Engineering, Taipei, Taiwan