Querying Objects with Complex Static Structure

Iztok Savnik[1] and Zahir Tari[2]

[1] University of Ljubljana, Faculty of Computer and Information Science, Slovenia
[2] Royal Melbourne Institute of Technology, Department of Computer Science, Australia

Abstract. This paper describes the database algebra QAL which serves as the logical algebra in the query execution system of the Distributed Object Kernel [25]. QAL is based on the concepts of the early functional query languages. It provides a simple functional semantics of the language and the means to express complex queries in a step-by-step manner. The operations of QAL are designed to allow simple and efficient manipulation of objects having complex classification or composition structure. The expressive power of QAL is presented through a case-study consisting of a comprehensive set of examples of queries which manipulate complex objects.

1 Introduction

Database management systems are employed for storing data in many novel computer applications ranging from computer aided design, process control and computer integrated manufacturing to software engineering, multimedia and Internet applications. Novel demands for the representation and manipulation of the data in these applications include the needs to represent and query objects having complex composition and/or classification structure.

In this paper we present the database algebra QAL [20] designed for the manipulation of complex objects [14, 1]. QAL is used as the logical algebra [11] in the query execution system of the Distributed Object Kernel [25] (abbr. DOK) which is currently under development. DOK will serve as a platform for the development of applications which involve heterogeneous distributed databases storing business and multimedia data.

QAL evolved from the FQL family [6, 4, 17] of functional query languages by generalising their operations for the manipulation of objects. Let us briefly present the operations and the expressive power of QAL. Firstly, QAL includes a set of operations, called *model-based operations*, which are derived from the concepts used in the QAL's database model formalisation. They enable users to inquire about the basic properties of objects from the extensional and the intensional parts of the database including, however, the classification hierarchy of objects. Secondly, the *declarative* operations of QAL are the functional operations defined on sets of objects. The declarative operations of QAL include a group of higher-order functional operations which allow for a simple way of expressing queries that manipulate composite objects.

The rest of the paper is organised as follows. Section 2 presents the formal view of the object-oriented data model which is used for the definition of the basic operations of the algebra. The complete set of QAL's operations is presented in Section 3. In Section 4 the use of QAL operations for expressing queries that manipulate objects with complex static structure is demonstrated through a case-study. Further, the main ideas used for the integration of QAL into the DOK are presented in Section 5. In Section 6 we present the related work. Finally, Section 7 gives a short conclusion.

2 Data Model

This section presents the main features of the object-oriented data model serving as a framework for the definition of the QAL algebra. The data model provides, in addition to the basic constructs of the object-oriented data model [3], a uniform view of the database by treating classes as abstract objects. In this respect, it is based on ideas introduced in the logic-based declarative language F-Logic [13]. The extended presentation of the data model can be found in [20].

2.1 Object Identifiers

An *object identifier* (abbr. *oid*) is a unique symbol which represents an entity from the real world[1]. An object identifier is either an *individual identifier* which represents a concrete entity, or a *class identifier* which represents an abstract concept.

The class identifier stands for a set of individual identifiers which represent the *interpretation* of a class identifier. Formally, the interpretation of a class identifier c is a set of individual identifiers denoted as $I_c(c)$; the elements of $I_c(c)$ are called the *members* of the class identifier c. The interpretations of two class identifiers are non-intersecting sets of object identifiers. Therefore, an individual identifier has exactly one parent class identifier.

The set of class identifiers is organised with respect to the partial ordering relationship $ISA\_SUBCLASS$ – this relationship is denoted as "$\preceq_i$". The partially ordered set of class identifiers is extended to include individual identifiers. A member of a given class identifier is related to this class identifier by the relationship $\preceq_i$. Formally, $o \in I_c(c) \Longrightarrow o \preceq_i c$, where o represents an individual identifier and c is a class identifier.

The *inherited interpretation* $[2, 24]$ of a class identifier c, denoted as $I_c^*(c)$, includes all *instances* of c, that is, the members of c and the members of $c's$ subclass identifiers. Formally, $I_c^*(c) = \bigcup_{c_j \preceq_i c \wedge c_j \in \mathcal{V}_c} I_c(c_j)$, where $\mathcal{V}_c$ denotes the set of all class identifiers of a given database.

[1] The object identifiers of QAL correspond to the so-called *logical id-s*, as introduced by Kifer et al. in [13].

2.2 O-Values

An o-value [2] is either: an oid; a set $\{o_1, \ldots, o_n\}$, where o_i-s represent o-values; or a tuple defined as $[A_1{:}o_1, \ldots, A_n{:}o_n]$, where o_i-s represent o-values and A_i-s are attribute names. An o-value can be either a ground o-value or a type o-value. A *ground o-value* includes only the individual identifiers and a *type o-value* includes only the class identifiers.

An example of a ground o-value is $[name{:}\text{"Jim"},\ age{:}50, kids{:}\{ana, tom\}]$, representing the properties of a person. The string "Jim" and the integer number 50 are the simple identifiers which are usually called constants. Next, the component $\{ana, tom\}$ is the set of individual identifiers which represent kids, that is, the instances of the class identifier $person$. The type of the above o-value is, as it will be presented shortly, the type o-value $[name{:}string, age{:}int, kids{:}\{person\}]$, where the terms $string, int$ and $person$ represent class identifiers.

The interpretation of a type o-value t, denoted as $I(t)$, is a set of ground o-values which is defined as follows. Firstly, in the case that the type o-value t is a class identifier, its interpretation is $I(t) = I_c(t)$. Secondly, the interpretation of a tuple-structured type o-value is $I([a_1 : T_1, \ldots, a_n : T_n]) = \{[a_1 : o_1, \ldots, a_n : o_n]; o_i \in I(T_i), i \in [1..n]\}$. Finally, the interpretation of a set-structured type o-value is $I(\{S\}) = \{s; s \subseteq I(S)\}$. If a ground o-value v is an element of $I(t)$ than we say that t is the *type* of v.

The relationship $\preceq_i$ defined on identifiers is extended to relate o-values. The new relationship is denoted as $\preceq_o$; we refer to it as a *more_specific* relationship. Suppose we have two o-values v_1 and v_2. Firstly, if the o-values v_1 and v_2 are object identifiers then $v_1 \preceq_o v_2 \iff v_1 \preceq_i v_2$. Secondly, if $v_1 = \{s_1\}$ and $v_2 = \{s_2\}$ where s_1 and s_2 are o-values, then $v_1 \preceq_o v_2 \iff s_1 \preceq_o s_2$. Finally, if $v_1 = [a_1 : t_1 \ldots, a_k : t_k]$ and $v_2 = [b_1 : s_1 \ldots, b_n : s_n]$ where t_i-s and s_i-s are o-values and the attributes of v_1 and v_2 are ordered so that $a_i = b_j$ when $i = j$, then $v_1 \preceq_o v_2 \iff k \geq n \wedge \forall i (i \in [1..n] \wedge a_i = b_i \wedge t_i \preceq_o s_i)$. Note that the partial ordering relationship $\preceq_o$ relates type o-values as well as ground o-values.

In a similar way to the inherited interpretation of class identifiers, we introduce the *inherited interpretation* of type o-values. Given a type o-value t, its inherited interpretation includes the union of interpretations of the type o-value t and the interpretations of all type o-values more specific than t. Formally, $I^*(t) = \bigcup_{t_j \preceq_o t \wedge t_j \in \mathcal{V}_T} I(T_j)$, where $\mathcal{V}_T$ denotes the set of all type o-values of a given database. Finally, the *natural interpretation* of a type o-value t, denoted as $I^\diamond(t)$, includes all o-values more specific than t. Formally, $I^\diamond(t) = \{o; o \preceq_o t\}$.

2.3 Objects

An object is defined as a pair (i, v), where i is an object identifier and v its corresponding value. An object identifier is a reference to an object, and a *value* represents the state of object realised by *o-value.* Similarly as in the case of object identifiers, we differentiate between two types of objects: *class objects* and *individual objects.* A class object represents an abstract concept and acts as a representation of a set of individual objects which share similar static structure and behaviour.

2.4 Working Example

The examples in the following sections are based on a conceptual schema which describes a simplified University environment. We define the conceptual schema by listing the class objects and the relationships among class identifiers which define the inheritance hierarchy of classes.

```
(department,[dept_name:string,
             head:employee,
             staff:{employee}])
(course,[title:string,
         instructor:lecturer ])
(project,[pro_name:string,
          descr:string,
          participants:{employee},
          budget:integer ])
(person,[name:string,
         age:int,
         address:string,
         family:{person}])
(student,[courses:{course}])
```

```
(employee,[salary:int,
           manager:employee,
           projects:{project},
           dept:department])
(lecturer,[])
(assistant,[])
(student-assistant,[])
```

student $\preceq_i$ person
employee $\preceq_i$ person
lecturer $\preceq_i$ employee
assistant $\preceq_i$ lecturer
professor $\preceq_i$ lecturer
student-assistant $\preceq_i$ assistant
student-assistant $\preceq_i$ student

3 Operations of QAL Algebra

The QAL algebra is based on the previously presented object-oriented data model. It includes two types of operation: model-based operations and declarative operations. The *model-based operations* are used to access and manipulate the properties of individual and class objects represented by the constructs of the data model. The *declarative operations* of QAL are used for expressing declarative queries on databases. The model-based operations are in this context used to assist the declarative operations in expressing database queries.

3.1 Model-Based Operations

In this sub-section we define the operations that are derived from the constructs of the previously presented object-oriented database model. Some of the model-based operations, i.e. the valuation operation, the extension operations, the equality operation and the operation `class_of`, are defined similarly as in other query languages (see, e.g. [15, 13, 12, 7]). The model-based operations which are introduced by the algebra QAL are the poset comparison operations, the closure operations and the operations `lub-set` and `glb-set`.

Valuation operation: The value of the object referenced by an identifier o can be obtained by the valuation operation `val` i.e. the value of o is o.`val`. If the valuation operation is followed by attribute name, then it can be abbreviated by the operator "–>" as it is common in procedural programming languages.

Extension operations: On the basis of the ordinary interpretation I_c and the inherited interpretation I_c^* of class identifiers defined in Section 2, two types of extension operations are defined. Firstly, the operation `ext` maps a class identifier to the set of its members. Secondly, the operation `exts` maps a class identifier to the set of its instances.
Class_of: The operation `class_of` maps an individual identifier to its parent class identifier. It is defined as $x.\mathtt{class\_of} = c \iff x \in I(c)$.
Poset comparison operations: These operations allow for relating o-values with respect to the partial ordering of o-values $\preceq_o$ introduced in Sub-section 2.1. The poset comparison operations are: $\preceq_o$, $\prec_o$, $\succ_o$ and $\succeq_o$.
Closure operations: Given a class identifier c, the closure operations `subcl` and `supcl` are used to compute the set of more specific or more general class identifiers of the identifier c. While the comparison operations can serve merely for expressing relationships among object identifiers, the result of the closure operation is the set of class identifiers that can be further used in the query.
Nearest common more general and more specific objects: The operations `lub-set` and `glb-set` are defined to compute the nearest more general or more specific object identifiers of the argument set of object identifiers with respect to the relationship $\preceq_i$.
Equality: Two types of equality operations are provided. Firstly, the ordinary equality denoted as "==" is used to compare o-values. Secondly, the *deep equality* [23] denoted as "=" compares object identifiers on the basis of values of the corresponding objects.

3.2 Declarative operations

Every declarative operation of QAL is a function which manipulates a set of o-values called the *argument* of operation. The operations have a set (possible empty) of *parameters* which can be either the o-values or the queries. The result of the declarative operation is an o-value.

A QAL *query* is an expression $o.f_1 \ldots . f_n$, where o is an o-value represented as a constant or a variable, the dot operator "." represents the functional composition and $f_i's$ are QAL operations. The parameter queries of the QAL declarative operations can use the *identity* function `id` to refer to the elements of the set which is an argument of the declarative operation.

Apply: The operation `apply`(f) is used to evaluate a parameter function f on the elements of the argument set. The parameter function f can be an attribute, a method or another query.
Select: The operation `select`(P) is used to filter an argument set of o-values by using a parameter predicate P. The operations that can be used to in the predicate P are: the standard arithmetic relationships, the set membership and set inclusion operators, previously presented model-based operations and the boolean operations.
Set operations: QAL includes the standard set operations union, intersection and difference which are denoted as `union`, `intsc` and `differ`, respectively. The

ordinary value equality "==" is used for computing the set operations.
Tuple: The operation `tuple`$(a_1 : f_1, \ldots, a_n : f_n)$ is a generalisation of the relational operation `project`. Given a set of o-values as an argument of the operation, a tuple is generated for each o-value from the argument set. The value of the attribute a_i is an o-value which is the result of the evaluation of the parameter query f_i.
Explicit Join: The operation `ojoin`(f, a_1, a_2, P) [23] joins the argument set of o-values with the result of the query f using the predicate P. The parameters a_1 and a_2 are optional. In the case that any of the `ojoin` arguments is not a set of tuples, then the o-values from this argument set are treated as tuples including a single component labelled a_i. Except from this, `ojoin` is defined as the relational `join` operation.
Group: The operation `group`$(a : f, b : g)$ applies the functions f and g to each element of the argument set. The results of the function g are grouped into sets in accordance with the results of the function f. The result of the operation `group` is a set of pairs which include the value of f (labelled as a) and the set of results of g (labelled as b) that share the common value of f.
Unnest: Three operations are defined to be able to unnest arbitrarily structured o-values in the argument composite objects. In order to provide an unified view of the possible types of unnesting, we give the operations the same name. The QAL `unnest` operations are defined as follows. First, the operation `unnest` is used to flatten the set of sets [23]. Second, given a set of tuples, the operation `unnest`(A) unnests a set-valued component A [23]. Finally, given a set of tuples that include a tuple-valued component A, the operation `unnest`(A) flattens the nested tuple [1].
Apply_at: To be able to apply a query to any nested components of the o-values from the argument set, the functionality of the operation `apply` is extended by a new parameter that serves as a component selector. The operation `apply_at`(p, f) identifies the desired components by applying the *aggregation path* p to the o-values that are the elements of the argument set. The aggregation path is specified by a sequence of attributes separated by dot operators. The query f is evaluated on the identified components. The evaluation of the aggregation path serves solely for the identification of the component and does not restructure the argument complex object.

4 A Case-Study

So far, we have presented the operations of the QAL algebra. In this section we focus our attention on the use of operations for querying objects having complex static structure. Two types of queries are studied: (i) queries that use conceptual schemata, and (ii) queries that manipulate complex objects.

The natural interpretation $I^\diamond$ presented in Section 2 is used for the definition of the semantics of o-values that appear in QAL queries. Note that an instance of a type o-value `t` can be any o-value which is more specific than `t`.

4.1 Using Conceptual Schemata for Querying Databases

Let us first define two general types of query which will later serve for the classification of queries which use the conceptual schemata. Firstly, the *intensional queries* are the queries the answers to which consist of the elements of the conceptual schema [19, 22]. Secondly, the *extensional queries* are conventional queries the answers to which consist solely of the extensional portion of the database.

The following types of query that use the conceptual schemata are presented. Firstly, the intensional queries allow for inquiring and reasoning about the properties of the conceptual schemata. Secondly, we present the extensional queries which include references to the intensional part of the database. Finally, the queries which generalise the results of the extensional queries are demonstrated.

Intensional Queries. The following query selects all subclasses of `person` which are more general than at least one of the classes `student-assistant` or `head`. In general, such queries use the poset comparison operations and/or the operations `subcl` and `supcl` to select a subset of classes representing the conceptual schema. The result of the query is an o-value which is the instance of the type o-value {`person`}.

```
person.subcl.select( id ≻o student-assistant or
                     id ≻o head );
```

The query presented below retrieves all classes at the top level of the class hierarchy[2], that is, the only superclass they have is `object`. The result of the query is an instance of the type o-value {`object`}.

```
object.subcl.select( id.supcl = {object} );
```

Extensional queries. The following query uses poset comparison operations to relate object identifiers. It retrieves the instances of the class identifier `person` which are more specific than the class identifier `lecturer` and, at the same time, more general than or equal to the class identifier `student_assistant`. The query result is an instance of the type o-value {`person`}.

```
person.exts.select( id ≺o lecturer and
                    student_assistant ⪯o id.class_of );
```

The following query demonstrates the use of the poset comparison operations for relating tuple structured o-values. It returns a set of o-values describing persons; each o-value must be more specific than the o-value `[address:"Brisbane", family:{employee}, manager:instructor]`. The result of the query is of type {`[ name:string, age:int, address:string, family:{person} ]`}.

```
person.exts.apply( id.val ).
            select( id ≺o [ address: "Brisbane",
                            family: {employee},
                            manager: instructor ]);
```

[2] This query was originally presented in [9].

Generalisations of extensional queries. The query given below answers the following question: Which are the parent classes of employees who work in CSD and who are younger than 25? The query first filters the set of employees and then it applies the operation `class_of` to each element of the resulting set. The result of the query is an instance of the type o-value {`employee`}.

```
employee.exts.select( id->dept = csd and
                      id->age < 25 ).
              apply( id.class_of );
```

4.2 Querying complex objects

In this section, we present the use of QAL operations for the manipulation of complex objects. The presentation is based on the classification of operations on complex objects defined by Kim et al. in [14]. The following types of query which manipulate complex objects are presented.

Suppose that the argument of a query is a set of complex objects. Firstly, the set of complex objects can be filtered by specifying the selection condition on the values of simple and complex attributes of argument complex objects. The second type of query projects the selected attributes from the flat and/or nested levels of complex objects from the argument set. Next, we present queries which filter nested components of complex objects. Finally, the use of QAL operations for restructuring complex objects is demonstrated. All examples in this section are based on the class object `department` which is now redefined as follows.

```
(department,[dept_name: string,
             head: employee,
             staff: {[emp_name:string, address:string,
                     projects:{project}, family:{person}]}]);
```

Retrieval of complex objects. Let us first present a simple query that filters a set of complex objects. The following query selects all departments whose head is a member of the class `professor` or some more specific class. The result of the query is an instance of {`department`}.

```
department.ext.select( id->head.class_of ⪯o professor );
```

A more complex case is retrieving complex objects by expressing the predicates on the nested components of complex objects. The predicates are specified by the use of nested queries. The query given below retrieves departments which are involved in the project identified by object identifier $pr4$. The result of the query is an o-value of type {`department`}.

```
department.ext.select( pr4 in id->staff.
                              apply( id.projects ).
                              unnest ));
```

Projecting Components of Complex Objects. Given a set of complex objects describing departments, the following query projects the value of the attribute `head` and the values of attributes that describe personal information about employees. This type of query is similar to the nested-relational operation `project` as defined in [8, 16]. The result of a query is an o-value of type {[`head: employee, staff: {[name:string, address:string, family:{person}]}]`}.

```
department.ext.tuple( head: id->head,
                      staff: id->staff.
                             tuple( name: id.emp_name,
                                    address: id.address,
                                    family: id.family ));
```

Filtering Nested Components of Complex Objects. The following query filters the nested components of o-values which describe departments. The nested component is identified using the aggregation path `staff.family`. The original set is replaced by the set of family members who are younger than 30. The query result is a set of o-values having the same structure as the o-values describing departments.

```
department.ext.apply( id.val ).
               apply_at( staff.family,
                         id.select( id->age < 30 ));
```

Restructuring Complex Objects. The first example demonstrates the use of QAL restructuring operations to realise the typical restructuring operations on NF^2 relations. The query unnests the attribute `staff` and attribute `projects` and, after, it groups the names of employees and departments by the projects in which they participate. The result is an instance of the type o-value {[`pro_id: project, group: {[ emp: string, dept: string ]}]`}.

```
department.ext.apply( id.val ).
               unnest( staff ).
               unnest( projects ).
               group( pro_id: id.projects,
                      group: id.tuple( emp: id.emp_name,
                                       dept: id.dept_name ));
```

The following example presents the query which restructures the nested components of complex objects. Given a set of complex objects describing departments, the query identifies the components that include a set of identifiers referring to projects and then replaces every occurrence of an oid with a tuple that contains the name and the budget of each project. The result is an instance of type {`[dept_name: string, head: employee, staff: {[emp_name: string, address: string, projects: {[pro_name: string, budget: int]}, family: {person}]}]`}.

```
department.ext.apply( id.val ).
              apply_at( staff.projects,
                       id.tuple( pro_name: id->pro_name,
                                budget: id->budget ));
```

5 The integration of QAL into DOK

The algebra QAL will serve as the *logical algebra* [11] in the distributed query processing system of DOK [25]. The *physical algebra* of the query processing system is based on the subset of QAL which is equivalent to the complete QAL algebra. It includes the operations that can be efficiently evaluated on distributed databases, and, for which there exists a set of well-defined rules used for the low-level distributed query optimisation.

The DOM+ [25] (Distributed Object Model), the data model of DOK, allows for the representation of different aspects of distributed databases using the usual constructs of object-oriented database model, the meta-objects, and the virtual objects. Firstly, the constructs of object-oriented database model provide the means for the representation of the classification, composition, inheritance and encapsulation abstractions in a local database. Secondly, the meta-objects serve for the representation of the object properties which relate to their role in a distributed environment, for example, the physical location of objects. Finally, the virtual classes and objects allow for the global view of the database in a distributed environments.

The model-based operations allow QAL to inquire about the relationships between different types of DOM+ objects in a comprehensible manner. They can serve for querying the relationships among virtual objects as well as more specific properties of objects at the local level, i.e. its static properties, types, etc. In addition, QAL allows for the uniform treatment of the data and the meta-data levels of the database [22]. Hence, the same language can be used to explore the data, the schema and the meta-data levels of databases.

The general direction of our approach to the manipulation of DOM+ objects is to provide the tools for simple and efficient definition of queries for the manipulation of virtual objects at the global repository, which is the abstract level of the representation of DOK objects. Furthermore, a user have to be provided with the means to manipulate other more detailed levels of the representations of DOM+ virtual objects, such as the implementation specifics, locations of objects, etc. However, the manipulation of such data may require more deep understanding of the DOK architecture.

6 Related work

The work on the QAL algebra has been influenced by the functional query language FQL proposed by Buneman and Frankel [6], by some of its descendants, that is, GDL [4] and O^2FDL [17], and by the functional database programming language FAD [10]. QAL subsumes the operations of FQL, i.e. the operations `extension`, `restriction`, `selection` and `composition`, as well as the extensions of FQL provided by the query languages GDL [4] and O^2FDL [17]. In contrast to FQL, QAL does not support recursion.

Further, the algebra QAL is closely related to the Query Algebra originally proposed by Shaw and Zdonik in [23]. This algebra has been further used as the basis for the EXCESS/EQUAL algebra and its query optimiser [18]. QAL shares many similarities with these algebras in some of the basic operations (i.e. `select`, `tuple` and set manipulation operations), while it differs significantly in the operations for restructuring complex objects and for querying nested components of complex objects. In addition, QAL includes a rich set of primitive operations which serve for querying database schemata.

Let us now very briefly present the work closely related to the two main problems addressed in the paper: the use of conceptual schema for querying databases, and querying complex objects. The problem of querying conceptual schema of object-oriented databases has, to our knowledge, not been addressed by recent database algebras. The query languages which include the constructs that allow for querying conceptual schemata and/or relationships between the schema and the instance portion of a database are: ORION [15], F-Logic [13], OSCAR [12], ODMG OQL [7] and P/FDM query language [9]. The related studies of the operations for querying composite (complex) objects include: [14] by Kim, [8] by Colby, [8] by Liu and [1] by Abiteboul. The comparison of the operations of QAL with the operations of the related database algebras and query languages can be found in [20, 21].

7 Concluding remarks

In this paper we presented the logical algebra of complex objects QAL which allows for expressing the queries on objects with complex static structure. We proposed a set of simple operations, referred to as model-based operations, which provide a uniform access to the intensional and the extensional portions of the database. The model-based operations are used for inquiring and reasoning about the properties of individual and class objects. Further, an operation called `apply_at` which is used for querying nested components of complex objects is introduced. Its semantics is simple and comprehensible, and it can serve as the base for the efficient implementation of queries that manipulate composite objects.

References

1. S. Abiteboul, C. Beeri, *On the Power of the Languages For the Manipulation of Complex Objects*, Verso Report No.4, INRIA, France, Dec. 1993.
2. S. Abiteboul, P.C. Kanellakis, *Object Identity as Query Language Primitive*, Proc. of the ACM Conf. on Management of Data, 1989.
3. M. Atkinson, et al. *The Object-Oriented Database Systems Manifesto*, Proc. of First Int'l Conf. Deductive and OO Databases, North Holland, 1989, pp. 40-57.

4. D.S. Batory, T.Y. Leung, T.E. Wise, *Implementation Concepts for an Extensible Data Model and Data Language*, ACM Trans. on Database Systems, 13(3), Sep 1988, pp. 231-262.
5. E. Bertino et al, *Object-Oriented Query Languages: The Notion and Issues*, IEEE Trans. on Knowledge and Data Engineering, 4(3), June 1992.
6. P. Buneman, R.E. Frankel, *FQL- A Functional Query Language*, Proc. of the ACM Conf. on Management of Data, 1979.
7. R.G.G. Cattell (Editor), *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, 1993.
8. L.S. Colby, *A Recursive Algebra and Query Optimization for Nested Relations*, Proc. of the ACM Conf. on Management of Data, 1989.
9. S.M. Embury, Z. Jiao, P.M.D. Gray, *Using Prolog to Provide Access to Metadata in an Object-Oriented Database*, Practical Application of Prolog, 1992.
10. S. Danforth, P. Valduriez, *A FAD for Data Intensive Applications*, IEEE Trans. on Knowledge and Data Engineering, 4(1), Feb. 1992.
11. G. Graefe, *Query Evaluation Techniques for Large Databases*, ACM Comp. Surveys, Vol.25, No.2, June 1993, pp. 73-170.
12. J. Göers, A. Heuer, *Definition and Application of Metaclasses in an Object-Oriented Database Model*, Technical Report, Dept. of Computer Science, Technical University of Clausthal, 1994.
13. M. Kifer, G. Lausen, J. Wu, *Logical Foundations of Object-Oriented and Frame-Based Languages*, Technical Report 93/06, Dept. of Computer Science, SUNY at Stony Brook.
14. W. Kim, H.-T. Chou, J. Banerjee, *Operations and Implementation of Complex Objects*, IEEE Trans. on Software Engineering , 14(7), July 1988.
15. W. Kim, et al., *Features of the ORION Object-Oriented Database System*, 11th Chapter in *Object-Oriented Concepts, Databases and Applications*, W.Kim (ed.).
16. L. Liu, *A formal approach to Structure, Algebra & Communications of Complex Objects*, Ph.D. thesis, Tilburg University, 1992.
17. M. Mannino, I.J. Choi. D.S. Batory, *The Object-Oriented Data Language*, IEEE Trans. on Software Engineering, 16(11), Nov. 1990.
18. G.A. Mitchell, *Extensible Query Processing in an Object-Oriented Database*, Ph.D. thesis, Brown University, 1993.
19. M.P. Papazoglou, *Unravelling the Semantics of Conceptual Schemas*, Communications of the ACM, Sept. 1995.
20. I. Savnik, Z. Tari, T. Mohorič, *QAL: A Query Algebra of Complex Objects*, Under revision for Data & Knowledge Eng. Journal, 1998.
21. I. Savnik, *A Query Language for Complex Database Objects*, Ph.D. thesis, University of Ljubljana, CSD Tech. Report, J.Stefan Institute, CSD-TR-95-6, Jun 1995.
22. I. Savnik, Z. Tari, *Querying Conceptual Schemata of Object-Oriented Databases*, Proc. of DEXA'96 Workshop, IEEE Comp. Society, 1996.
23. G.M. Shaw, S.B. Zdonik, *A Query Algebra for Object-Oriented Databases*, Proc. of IEEE Conf. on Data Engineering, 1990.
24. S.L. Vandenberg, *Algebras for Object-Oriented Query Languages*, Ph.D. thesis, Technical Report No. 1161, University of Wisconsin, July 1993.
25. Z. Tari, A. Zaslavsky, I. Savnik, *Supporting Cooperative Databases with Distributed Objects*, In "Parallel and Distributed Systems: Theory and Applications", J.L. Aguilar Castro (Editor), Int. Inst. of Information and Systemics, 1998, To appear.