Arbeitspapiere der GMD
GMD Technical Report
No. 763

Karl Aberer, Gisela Fischer

Object-Oriented Query Processing: The Impact of Methods on Language, Architecture and Optimization

July 1993

GERMAN NATIONAL RESEARCH CENTER FOR COMPUTER SCIENCE

Addresses of the authors:

Dr. Karl Aberer
GMD-IPSI
Dolivostraße 15
D-64293 Darmstadt
Email: aberer@darmstadt.gmd.de
Phone: ++49/6151/869-935
Fax: ++49/6151/869-966

Gisela Fischer
GMD-IPSI
Dolivostraße 15
D-64293 Darmstadt
Email: fischerg@darmstadt.gmd.de
Phone: ++49/6151/869-933
Fax: ++49/6151/869-966

Address all correspondence to:

Dr. Karl Aberer
GMD-IPSI
Dolivostraße 15
D-64293 Darmstadt
Email: aberer@darmstadt.gmd.de
Phone: ++49/6151/869-935
Fax: ++49/6151/869-966

Abstract

Although nearly all object-oriented data models proposed so far include behavioral aspects, most object-oriented query languages, algebras and query optimization strategies simply adapt relational concepts since they focus on the complex structures of objects and neglect the behavior. We claim that this approach is not sufficient since it does not reflect the much richer semantics methods can carry which have to be taken into account for really efficient query processing. The quite straightforward approach we consider is to integrate methods in an algebraic framework for query processing and to make there partial knowledge about methods available in the form of equivalences. We examine two important questions which emerge from taking this approach. First, how is it possible to integrate algebraic set operators with methods defined in database schemas within an object-oriented data model? Second, what is the impact on the architecture of the query processor when the algebra becomes an extendible component in query processing?

1 Introduction

Query processing in object-oriented database management systems is an active research area. The goal is to bring together the main features of object-oriented database management systems, namely a data model, which supports objects with complex structure and behavior, and extensibility, with a declarative and efficient query language. The concept of declarative query languages has proven to be successful in the framework of relational database management systems. As query processing in relational and nested relational database management systems is considered as starting point, currently most approaches for object-oriented query processing focus mainly on the structural part of object-oriented data models and do not take account of the much richer semantics methods can carry and which have to be taken into account for really efficient query processing in non-standard applications. Thus conclusions as in [32] are drawn: "In retrospect, the extent of similarities between relational and object-oriented query processing should not have come as a surprise, if we recognize that, regardless of data models, query processing is essentially a process of mapping a declarative query expression into a sequence of procedural executions."

Although the knowledge from relational query processing is an important starting point, we claim that there are many new issues to consider, which play only a minor role in relational query processing. Despite many efforts in the area of object-oriented query processing during the last years they are still quite far from being satisfactorily solved. Simply adapting relational query processing concepts by taking a structural view is not sufficient for data models providing structure <u>and</u> behavior. Any query processor that claims to support the features of an object-oriented database management system should satisfy the following objectives: support for complexly structured objects with *methods* defined on these objects, *extensibility* with respect to the processing resources and strategies, *modularity* of processing components, *declarativity* and *efficiency.* As an overall observation this means to step from hard-coded to flexible and adaptable query processors.

In this paper we want to shift the attention away from the structural view and focus on behavioral aspects. We will analyze the impact of behavioral extensions on the different components of the query processor. Simply allowing method calls in query statements as it is proposed in many query languages is by far not enough. Methods have an impact on the whole architecture of the query processor, starting from the language, over optimization up to execution. Therefore we will come up with a *modular reference architecture* for a query processor. Based on this architecture we will analyze some of the components in detail.

One of the main issues is the choice of a data model and a query language. There we rely on the framework of the object-oriented database management system VODAK which is currently developed and implemented at GMD-IPSI, its manipulating language VML and its query language VQL. VQL is an object-oriented query language, developed on the basis of the SQL paradigm. We show how VQL allows to fully exploit the features of the underlying object-oriented data model VML, in particular methods.

For query optimization we take a completely algebraic viewpoint. However, we want to get away from a built in algebra for two reasons. First, extensibility makes it necessary to give the user the means to extend the query algebra, and the means towards this end are methods. Second, the query optimizer should also be able to use knowledge about application specific methods. Thus we represent the algebra within the VML data model completely by methods in a *method algebra*. Based on this method algebra we discuss different possibilities how knowledge about methods, given in form of equivalences, can be used by the query optimizer. Thus the meaning of semantic query optimization as known from relational database systems radically gains in importance if we take methods into account.

Search strategy as well as cost and execution models should be exchangeable or modifiable as the applications we carry in mind will take place in different environments, e.g., on distributed platforms, which will of course lead to changing needs from applications. We will sketch shortly how to come to a completely self-contained description of the query processor within the object-oriented data model.

The remainder of the paper is organized as follows: In Section 2, we review the related work. In Section 3, we propose a modular reference architecture. In Section 4, we introduce the VML data model and in Section 5, the VQL query language, including some aspects of processing the query input. In Section 6, we discuss how to model a query algebra in our object-oriented data model leading to the notion of method algebra. In Section 7, we discuss equivalence-based query optimiziation based on the method algebra. In

Section 8, we give an outlook on a possible design of the other components of the query processing architecture and finally, in Section 9, we give some concluding remarks.

2 Related Work

The main body of work in object-oriented query processing focuses on structural aspects and is thus closely related to work on relational query processing (e.g. [29]). A large number of query languages and object-oriented algebras where proposed in this direction [13][31][33][35][37][39][41][45]. Closely related to this are works on different variants of indexing techniques for path expressions [7][8][30]. This work has already developed into a stable framework for query languages on the structural part of object-oriented databases, for overviews see [9][27]. In [43] an architecture closely related to relational architectures is proposed. Implementations that follow these proposals are found in Orion/Itasca [5][32], O2 [4][14][16], OSCAR [26] and Cocoon [40].

Work concentrating on the impact of methods in the framework of object-oriented query processing is somehow rare. Some theoretical aspects are covered in [6]. An approach suggesting precomputation of methods for indexing is described in [10]. A proposal to reveal knowledge about the execution of methods behavior to the query processor is described in the REVELATION project [23].

Apart from this general picture for standard object-oriented query processing we have to take account of different developments in related areas.

In particular relevant for us is work on extendible database systems. Here, we mention the extended relational system POSTGRES, with an extendible data model that is exploited in query processing [42]. In the EXODUS/VOLCANO projects generic support for query processing in extendible relational and object-oriented database systems was investigated and prototypes for query optimization and execution were developed [15][21][24]. In [12] a query optimizer that was built on this prototypes for the Open OODB system is described. An extendible database programming language is described in [44]. A proposal for modelling extendible search strategies of query optimizers is found in [36].

Rule based query optimization is fundamental in extendible database systems. Background on this is found in [19][22][24]. A particularly interesting implementation was given for structural object oriented data models in [30].

Semantic query optimization is another important issue for our work. For an example of semantic query optimization in relational systems see [11]. In [20] also semantic knowledge of operations on abstract data types is considered. In [43] semantic knowledge about classes is exploited for query optimization.

3 A Reference Architecture

In order to provide a clearly defined framework for the subsequent discussion and to clearly identify the different components of query processing that can be affected by methods we propose in this section a reference architecture for a query processing. It is designed according to our objectives and identifies the main orthogonal components needed for query processing. In the following sections we will discuss the impact of methods on different components of this architecture. An illustration of the architecture is found in Figure 1.

As a first component we consider the *query language* as it is offered to the user. This may be an SQL-like textual query language, a DML programming language extension, an application interface in a host language or a visual query interface. Important is support for set-oriented access and full exploitation of all structural and behavioral features of a database schema of the underlying *object-oriented data model*. The query expression is parsed and transformed into an internal representation. Then the *semantic correctness* of the query is checked. This step has to take account of the *database schema*. Next a *simplification step* takes place. This step simplifies the representation of the user input such that it can be translated to a *default algebraic expression* of the *query algebra*. The query algebra has to have at least the expressive power of the query language but it has also to be extendible by operations defined in the database schema.

The *optimization* process can be considered as a search problem: the *search space* for a specific expression is produced by a set of *equivalences* between algebraic expressions and consists of the set of expressions which are semantically equivalent under the given set of equivalences. The search is for the expression with minimal cost according to a given *cost model*. The *search strategy* is an algorithm which has to perform this task. We do not consider a specific strategy, as different strategies may be appropriate in different contexts. However, each strategy must be able to deal as well with the extendible set of algebraic operators, an extendible set of equivalences and an extendible cost model.

We separate the algebra from the *execution model*. The execution model describes the translation of the algebraic expression in a piece of executable code. The execution models may realize completely different paradigms of computations, e.g. object-at-a-time vs. set-at-a-time (data-flow vs. functional). It may happen that two algebraic operators are translated in the same way in a specific model. Closely related to the execution model is the cost model which, in dependence of a database state, quantifies the cost of each operator to be executed.

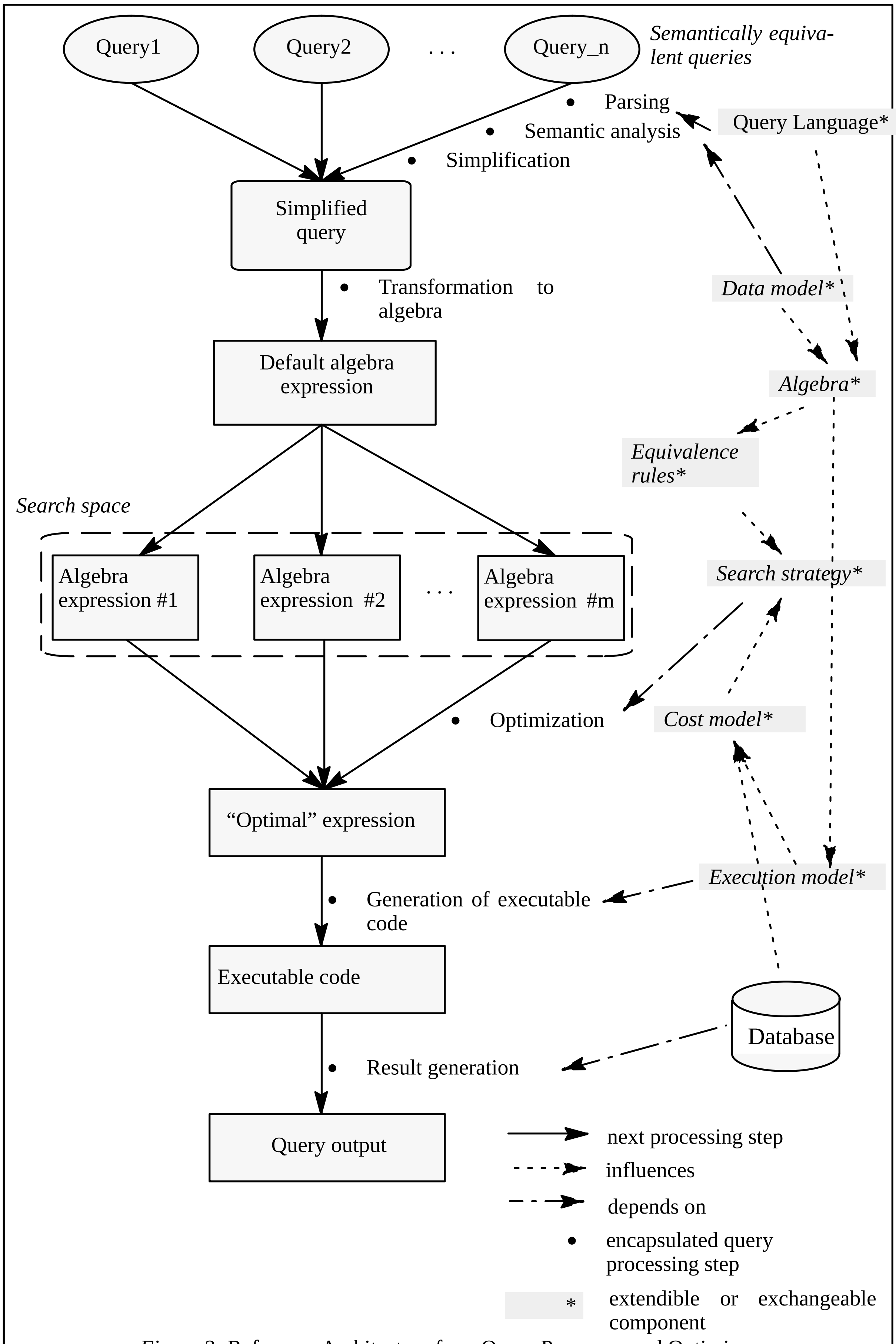

Figure 3: Reference Architecture for a Query Processor and Optimizer

In the final step the optimized algebra expression is mapped to an *executable expression* by referring to the execution model. The executable expression produces the desired *query result*. These two steps can be performed by the query processor either in an interpretative way, by looking up the appropriate execution for an algebraic operator and immediately executing it, or by compilation, where the executable is first produced and then executed afterwards.

4 The Data Model

The VODAK Modelling Language VML consists of a DDL part which is used to define database schemas and a computationally complete DML part for the implementation of methods and application programs. In the following we describe the data model which is underlying VML.

Objects represent material or immaterial real world entities or abstract concepts which are stored persistently. They possess *unique identifiers* and have *properties* and *methods* which describe their *structure* and *behavior.* Properties and methods may be *public* or *private*, while method implementations always remain hidden. Public properties are accessible through system-defined methods, hence the access to objects is provided through methods only. Method invocations in VML are denoted by *o→m(args)* where *o* is the object identifier of the receiver object, *m* is the name of the method and *args* are the method arguments.

Objects with common structure and behavior are instances of the same *class*. An object is an instance of only one class. The structure and behavior of the instances of classes is specified in *parametrized object types*. Thus object types and classes are different concepts, which is known as the *dual model*. This separation between types and classes can also be found in, e.g., COCOON [40] and OSCAR [26]. As a consequence for example different classes may use the same object type. Classes are objects themselves and belong to *metaclasses*. Thus they can also be receiver objects for messages, e.g., the system-defined method *instances()* can be sent to each class *C* with *C→instances()*. For details on the relation between object types and classes, and the metaclass concept see [3][34].

Domains are used for the declaration of variables, which keep transient values, of properties, which keep persistent values, and of methods in object interfaces. We distinguish *primitive domains*, like Integer and String, *class domains*, which consist of object identifiers, and *complex domains* built up from other domains with *domain constructors* like set, tuple, array or dictionary. To values of primitive domains built-in *primitive operations* can be applied, like addition or multiplication for Integer. To values of complex domains the *access operations* to the domain constituents can be applied, like access to components of tuples. To class domains, which consist of all (potential) object identifiers belonging to instances of a specific class, method calls can be applied. Any expression built up from primitive operations, access operations, method calls and paths is called a *VML expression*.

In VML usually part of the state of an object is made public in form of properties. As we have already mentioned, for public properties the DBMS generates corresponding methods for reading and writing automatically. In order to ease property access we allow access to the properties of an object by using the standard dot notation, i.e., instead of directly calling the system-generated read method we can read the property *p* of the object variable *x* of a class domain by *x.p*. Note that this does not violate object encapsulation.

Properties and method declarations in object types need the specification of domains. This is accomplished either by using primitive domains or through object-type parametrization. The object-type parameters represent domains which satisfy constraints given by object types. During class definition these parameters are substituted by class domains. This substitution is called *type-to-class mapping*.

Figure 2 gives an example of a simple VODAK schema. It shows the definition of the class *Person* and the object type *Person_Type* which is used as the instance type of *Person*. In the class definition of *Person* the object-type parameter *P* of *Person_InstType* is mapped to the class *Person*, thus leading to a recursive definition of this class. *Person_Type* defines the public properties *name*, *father*, *mother*, the private property *birthday*, and the public methods *setBirthday*, *age* and *children*. The properties *mother* and *father* reference other instances of the class *Person* using object identifiers. *setBirthday* assigns a value to *birthday* (we assume *date* is a user-defined data type), *age* computes the actual age of a person (since the meaning of this method is clear we only describe its implementation), and *children* computes all children of a person with a VQL-query expression. SELF denotes the actual object, i.e., the receiver object of the method.

The implementation of VODAK is based on C++. The VML compiler analyzes database schemas and applications written in VML syntactically and semantically, translates (DML-) VML code to executable C++ code, and generates a *data dictionary*, i.e., a persistent system catalog, for a database schema.

5 The VODAK Query Language VQL

The query language VQL supports declarative access to object-oriented databases in the VODAK database management system. For pragmatic reasons VQL is based on a SQL-like approach [18]; in fact, it is similar in appearance to POSTQUEL [42] and the query language of ORION as presented in [33]. VQL allows the exploitation of all features of the underlying data model described in the previous section, in particular method calls. Additionally, compared to standard SQL it provides features which are either convenient or necessary in order to efficiently access VML databases. Among these are path expressions of variable length, existential and universal quantifiers and arbitrary nesting of queries. Extensions for aggregations and recursions will be integrated in the future. For space limitations we also do not consider nested queries in this paper.

Since we allow arbitrary method invocation in queries we cannot determine in advance whether a query is a pure retrieval query or whether updates are performed due to the execution of the methods contained in the query [18]. In order to reflect this semantics in the syntax we replaced the SQL-keywords SELECT/ UPDATE by the more general keyword ACCESS.

A VQL query has the form

X := ACCESS – FROM – WHERE

and thus is an extension of VML. We now describe the different clauses of a query in more detail:

ACCESS

This clause contains an VML expression that computes the single values of the set which will be assigned to the query result *X*. If the expression returns a value of domain D the query returns a value of domain {D}. The expression contains variables bound in the FROM clause.

FROM

This clause contains the declaration of *range variables* which are bound to the *input sets* against which the query is posed. A query is posed against classes by taking the extensions of the classes as input. The extensions are given as sets over the class domains, i.e., as sets of object identifiers. These sets of object identifiers are obtained by sending the system defined method *instances()* to the class objects or by any valid VML expression returning sets of object identifiers.

```
OBJECTTYPE Person_Type [P: Person_Type]
INTERFACE
PROPERTIES
   name: STRING;
   father: P;
   mother: P;
METHODS
   setBirthday(d: date);
   age(): INT;
   children(): {P};
IMPLEMENTATION
PROPERTIES
   birthday: date;
METHODS
   setBirthday(d: date); { birthday := d;};
   age(): INT; {// compute the age in years using the actual date and birthday;};
   children(): {P};
      { RETURN
         (ACCESS c FROM c IN P WHERE c.father==SELF OR  c.mother==SELF;)
      }
END;

CLASS Person
INSTTYPE Person_Type [Person]
END;
```

Figure 4: Definition of the object type *Person_InstType* and the class *Person*

WHERE

specifies the conditions according to which elements should be selected from the input sets specified in the FROM clause. The WHERE clause consists of *atomic predicates* combined with the Boolean operators AND, OR and NOT. Atomic predicates are ==, !=, <, <=, >, >=, IS-IN, IS-SUBSET and any method call with Boolean result type. We assume for the operands and parameters strict domain compatibility. The WHERE clause may also contain existential and universal quantifiers over variables which are bound to finite sets.

In any place of a query where a value of a specific domain is required, e.g., a set in the FROM clause, a Boolean value in the WHERE clause, an operand of an atomic predicate or a parameter of a method call, also an VML expression may appear. This expression can involve method calls, path expressions or domain constructors and has to deliver values of appropriate domains. A formal specification of VQL is found in [1].

We allow arbitrary method calls to define the input sets in the FROM clause. However, we impose a dependency restriction for the FROM clause in order to exclude recursion. The variable bindings, i.e., the IN clauses, have to allow a reordering such that the following condition holds: a variable on the right-hand-side of an IN clause may only appear on the left-hand-sides of IN clauses, which follow the IN clause where the variable is bound. For example, the FROM clause *FROM c IN p→children(), p IN Person* can be reordered to *FROM p IN Person, c IN p→children()* which satisfies the condition. We will consider relaxations of this condition, which basically comes down to a topological ordering of the dependencies, when we introduce recursive VQL queries in the future.

The semantics of a VQL query is given by an *object calculus* expression. Object calculus is a specification mechanism derived from relational tuple calculus based on predicate calculus. Several variants of object calculus were investigated also taking account of method calls in the query, e.g., [9][37][43]. In this paper we do not specify the details of this calculus, e.g., with respect to atomic predicates, resolving path expressions, safety of calculus expressions and later translation to an algebra expression. We only sketch how a query is mapped to a calculus expression by an example

```
ACCESS expr(x,y,z)
FROM x IN A, y IN B, z IN C
WHERE pred(x,y,z)
```

is the set of data that satisfies

$$\{r \mid r = expr(x,y,z) \wedge x \in A \wedge y \in B \wedge z \in C \wedge pred(x,y,z)\}$$

Of course all transformations of VQL queries in the simplification process have to be consistent with this semantics.

After the query is parsed according to the above VQL syntax the semantic correctness is analyzed. In this step the validity of the input sets, the domain compatibility of operands and parameters, in particular also for nested method calls and path expressions, as well as the availability of methods for objects is checked according to the schema information stored in the data dictionary. Also the dependency restriction on input sets is checked here. After semantic analysis a simplification step takes place.

Simplification replaces shortcuts for the user, like using class names instead of method calls for producing class extensions, by the proper VQL expressions. It reorders the FROM clause according to the dependency restriction. The most extensive part of simplification applies to the WHERE clause, where atomic predicates are analyzed and simplified and the condition is brought into prenex disjunctive normal form.

6 The Query Algebra

A simplified VQL query is mapped to an internal algebraic representation for subsequent algebraic optimization. The *query algebra* we propose consists of the following constituents:

- *Kernel algebra:* It is provided by the system and has to be powerful enough to express any simplified VQL query, which is translated using these generic operations. Operators of this algebra have only values from set domains as arguments.
- *Application algebra:* These are the user-defined application specific methods. Operators of this algebra take values of arbitrary domains as parameters.
- *Execution algebra:* These are user-defined operators taking values from set domains as arguments. They do not introduce application specific semantics but express alternative execution strategies as those expressible in the kernel algebra.

The execution and the application algebra both are defined within the framework of VML schemas, whereas for the kernel algebra we have different options, which we will discuss later. It is important to observe that modelling the query algebra, including the kernel algebra, is crucial if we want to extend the query algebra by user defined methods and to exploit application specific knowledge in the optimization process.

6.1 Set Operators

To discuss modelling of the algebra in VML we first have to analyze the structure of algebraic operations as they typically appear in set–oriented query processing.

The query algebra is based upon *set operators* which take values from set domains as arguments. The signature of such an operator with arity *n* is in general of the form

$$OP(f_1,...,f_m; \{D_1\},...,\{D_n\}) : \{D\}.$$

where $\{D_1\},...,\{D_n\}$ are the set valued domains, $\{D\}$ is the result domain and $f_1,...,f_m$ are the *parameters* of the operator. The interpretation of such operators is given as mappings on the set-valued domains.

The parameters play an important role as they are used to define whole *families of operators*. A parameter is in general given in the form of a *pattern for a VML expression* with variables $x_1,...,x_n$ corresponding to the domains $D_1,...,D_n$

$f_i(x_1,...,x_n) : F_i$,

where F_i is the result domain of the expression. So, a parametrized operator OP corresponds in general to a family *OP*={OP_i}, where OP_i has the signature

$OP_i(\{D_1\},...,\{D_n\}) : \{D\}$.

More precisely the family *OP* is dependent on a particular schema as for each schema the expression pattern leads to a different set of matching expressions.

For illustration of the concepts we give a small example, namely different types of selection operations (with D an arbitrary domain)

(1) SELECT({D}, expr(x): BOOLEAN): {D}

(2) SELECT_EQ({D}, expr(x): INT, val): {D}

(3) SELECT_EQ_AGE({Person}, x–>m(): INT, value): {Person}

Many variations of relational and object oriented (set manipulation) algebras can be described within this framework. In this paper we do not discuss a specific algebra but concentrate on the means to integrate an (arbitrary) algebra into VML. What distinguishes the different variants is mainly the degree of parametrization of the operators. On the one end we have highly parametrized operators (e.g. [13][44]), together with set union and intersection. These operators typically have the following form

FILTER($pred(x_1,...,x_n)$: BOOLEAN , $expr(x_1,...,x_n)$: D , $\{D_1\},...,\{D_n\}$) : {D}

to which a simplified VQL query can be immediately mapped. The other extreme is, e.g., as proposed in the relational case, a pure algebra restricted to single-attribute, single-tuple constant relations, selection with single comparisons, natural join, projection, union, difference and renaming [38]. There the mapping from a VQL expression to the algebraic form is less straightforward.

Both extremes are not desirable for equivalence based optimization. A highly parametrized algebra will increase complexity in application of rules while a puristic algebra introduces many different operators which increases complexity of the algebraic expressions. A reasonable choice of operators for the kernel algebra is a question for further investigation for which we intend to provide a framework. Which particular choice is made is not important for the rest of the paper. For the sake of giving examples we assume the existence of a selection operation SELECT with different degrees of parametrization, as already introduced.

6.2 Modelling the Query Algebra in VML

In the following we discuss the representation of set operators within VML. Basically there are the following three possibilities available.

As Domain Operators:

In this case set operators are provided as built-in operators manipulating sets of data values directly. Some simple operators, like set union, are already provided in this form. This approach has the following drawback: there is a mismatch between the built-in set operators that are provided by the system and the user–defined set operators that can be supplied by the user only in the form of methods. The optimizer (as well as the execution model) has to deal with two different concepts for the same purpose. Additionally in VML the primitive domains and the operations are not extendible, thus any change of the query algebra leads to a change of the data model.

As Class Methods:

In this case set operators are modelled as class methods, i.e., the receiver object and the arguments of the operators are classes. The set operations are performed on the extensions of the classes. In order to reuse results of operations directly the result of those methods must again be a class. That means each query processing operation produces a new (intermediate) class holding the result. In VML the consequences are disastrous, as each object may belong only to a single class. Each result value has to be represented by a newly created object.

As Powerset Methods:

A compromise of the previous two solutions is the following: we introduce a powerset class P_C for each class C.

```
CLASS P_C
INTERFACE
PROPERTIES
    rep: { C }
METHODS
    Select(“parameter”) : P_C
    // other set operators
END;
```

That means, an instance p_S of the powerset class P_C represents an arbitrary subset S of the extension of C. The set operators are modelled as methods of the powerset classes. These methods for set processing operations are sent to the representative objects p_S. In each intermediate step a result set corresponds to one new instance of P_C, which is only a small overhead. The generated result objects again can be receiver objects for set processing operations.

From the above discussion it becomes clear that we favor the last approach. We introduce system classes for modelling arbitrary sets of objects with the corresponding set processing operations. Moreover, we have to consider more general classes which allow, e.g., the modelling of arbitrary sets of *tuples of objects* in order to represent the results of join operations.

This approach of course does not imply that the implementation of the set operators has to be realized in the same way as user-defined methods using the DML. This is similar to the situation for other system-defined classes, e.g., the class method *instances()* is also realized by a built-in implementation. The advantage of the approach is a *uniform model and interface* for the query optimizer as well as for the user and a homogeneous integration of user-defined methods with system-defined set operations.

After having defined a framework for representing set operators of the query algebra by methods we encounter the following problem: as already discussed parametrization plays an important role for the description of set operators. These parameters, when considered as method arguments, are not in domains of constant values but in domains of functions (see, e.g., the different SELECT operators given earlier). For built-in operators we can provide built-in mechanisms to deal with such parameters, but for method arguments we have to extend the VML data model. For example, we want to express the query

ACCESS p FROM p IN Person WHERE p→age()==30

by method calls corresponding to set operators. This can be done in the following ways depending on the degree of parametrization assuming that *P* is an instance of *P_Person*, which is the powerset class of *Person*:

(1) P→Select(“p→age()==30”)

(2) P→Select_Eq(“p→age()”,30)

(3) P→Select_Eq_Age(30)

Only (3) can actually be expressed with domains of constant values. For a conceptually clean representation of (1) and (2) *function domains* are needed. In the following we denote such a domain by giving the signature of the functions of this domain, e.g., the unary functions on *Person* returning Integer are denoted by *Person* ⇒ *INT*. Using such function domains we can write the signatures of the first two method calls as

(1) Select(cond: Person ⇒ BOOLEAN)

(2) Select_Eq(cond: Person ⇒ INT, val: INT).

Values of function domains are represented by any valid VML-expression containing placeholders for the function arguments (which is basically a lambda notation). The corresponding method calls then are

(1) P→Select(#→age()==30)

(2) P→Select_Eq(#→age(), 30)

Currently VML does not support function domains explicitly, but they will be provided in the future.

Finally a further issue has to be discussed. So far we have modelled the powerset classes individually for each application class. This may be appropriate for some application-specific set operators, but not for the general system-defined set operators, which are defined uniformly for all classes. At this point the metaclass mechanism of VML can be employed in order to define this system-wide uniform behavior of classes in metaclasses. The same mechanism is used, for example, to model system-inherent class methods like *instances()*. It is outside the scope of this paper to discuss this issue in detail.

6.3 Method Algebra

As a result of modelling set operators as methods our query algebra has now turned into a *method algebra*. Expressions of the method algebra are a subset of general VML expressions. Within this algebra we can now identify exactly the three parts mentioned at the beginning of the section. The kernel algebra consists of the system-defined methods for representing a basic set operators; the execution algebra consists of additional set operators which are user defined, and the application algebra consists of all user-defined methods. The latter appear inside parameters of set operators, but as we have parameters now available as ordinary method arguments, they become a fully integrated part of the method algebra. This will be one key for using application-specific knowledge in query optimization.

In this framework, also a distinction between a *logical and physical algebra* is no longer given. (Logical and physical algebra are taken from the terminology of [23]; expressions of the physical algebra often are referred to as *query evaluation plans.*) User-defined methods will always provide implementations and thus for those the two algebras collapse into one. Instead of a separate logical algebra we consider equivalence relations on the expressions of the method algebra which can be used to impose a structure on the search space.

7 Algebraic Optimization

In the method algebra all operators are equally righted. Furthermore, we no longer consider parametrization of set operators, but have integrated parameters as a part of the algebra. Therefore we can use the same mechanism to describe equivalences for built-in operations as for user-defined operations. For the *kernel algebra* we will assume a fixed *kernel equivalence set,* whereas for user-defined methods we consider *user-defined equivalences* for methods. This leads to the notion of *semantic query optimization*.

In relational systems semantic query optimization is proposed to exploit semantic properties defined in the schema, e.g., constraints on attributes. But this approach plays a minor role in relational optimization and usually is not considered in object-oriented optimization (for an exception see [20]). The crucial observation is that query optimization in behavioral object-oriented database systems has to exploit semantic knowledge about methods for query transformation. We provide semantic knowledge of methods in the form of user-defined equivalences.

In the following we want to analyze in which ways semantic knowledge about methods can contribute to query optimizations.

7.1 Complete Specification of Methods

In general the behavior of methods is defined operationally, i.e., by a piece of code. This is considered reason enough not to inspect this behavior as it appears to be intractable in a declarative framework. However, a close inspection shows, that in many cases methods by no means exploit the full complexity of operational specifications but can be specified fully by an equivalent expression in the method algebra. This results in equivalences of the form

$$\mathrm{o}{\rightarrow}\mathrm{m}(\mathrm{arg}) \equiv \mathrm{expr}$$

where the VML expression *expr* does not contain the method *m*, but may contain the receiver object *o* and the arguments *arg*. Such equivalences appear in two different contexts. In the first case, which we call *method expansion,* the equivalence is applied in forward direction, that means the intention is to replace a “black box” method call by a transparent expression in order to enable the optimizer to find a more efficient execution of the query. In the second case, which we call *expression contraction,* the rule is applied in backward direction, that means the intention is to replace a complex expression by a “black box” method call, which should perform better than the direct execution of the complex expression. The application direction of the equivalence as a rule is of course a conceptual distinction which is only of importance for the designer of a rule system, but not from an algebraic viewpoint.

7.1.1 Method Expansion

We illustrate this with the following three typical examples:

(1) *Path methods:* Often methods are used to compute reference chains to other objects. A simple example is the method *grandfather* which could be defined for the instances of the class *Person* as follows

grandfather(): Person; {RETURN (SELF.father.father)};

(2) *Queries:* Since the query language is a part of the DML, it is very natural that many methods will be expressed by queries, e.g., the method *children* in the example of Figure 2.

(3) *Message passing methods:* In VML there exists a mechanism which allows to automatically delegate a method call at run time to another object based on dynamically generated semantic relationships. These semantic relationships are realized through metaclasses [34]. For example, the class *Student* may be a role specialization of *Person*, and thus inherits all *Person* properties logically but not physically (as *Person* and *Students* are disjoint classes). The method call *name* to a student is then forwarded to the corresponding person.

The semantics of these methods can be described by the following equivalences, namely

(1) p IN Person: p→grandfather() $\equiv$ p.father.father

(2) p IN Person: p→children() $\equiv$ Person→Select(#.father==p OR #.mother==p)

(3) s IN Student: s→name() $\equiv$ (s→roleOf())→name()

Note that the rules do not necessarily only reflect the actual implementation of the method. Alternatives can be specified for the same methods, as is illustrated by the following example (under some mild assumptions on the marital status):

(4) p IN Person:
p→siblings() $\equiv$ p.father→children()
p→siblings() $\equiv$ p.mother→children()

Making knowledge about the semantics of such methods available in the schema, and thus to the optimizer, offers a great potential for the optimization of methods by revealing the semantics of their implementation without violating encapsulation. Note that such rules also can blur the clear distinction between a level of set operators and a level of parameters containing user-defined methods, as illustrated in example (2).

7.1.2 Expression Contraction

Since object-oriented database systems explicitly support extensibility it is natural that also the set operators which are usually considered as built-in constructs alternatively may be provided in form of methods. For example, consider a user-defined class method for exploiting indices with signature

Select_Ind_Age(val: INT): P_Person

which retrieves all instances of a the class *Person* for which the property *age* has value *val*. Then the semantics of this operation can be specified with the following equivalence, assuming a conventional *Select* operator on the powerset P_C of a class C is given.

(5) P_Person–>Select(#–>age()==val) $\equiv$ Person$\rightarrow$Select_Ind(val)

7.2 Partial Specification of Methods

In many cases it may not be possible to specify the behavior of methods completely. Still it might be possible to formulate equivalences involving the methods which are of the form

$\text{expr}_1(\text{m}) \equiv \text{expr}_2(\text{m})$.

Such equivalences formulated for set operators play an important role in query optimization, e.g., to express join associativity. So, as methods may exhibit similar properties, due to the fact that they can possess multiple parameters (whereas attributes have to be considered in this context as unary functions) and can realize set operators, it is natural to allow such equivalences also for (user-defined) methods.

We illustrate this point by the following examples. Assume a method *common_anc* is given that computes the (closest) common ancestor of two persons. This method is of course commutative but not associative:

(6) p IN Person, q IN Person: p$\rightarrow$common_anc(q) $\equiv$ q$\rightarrow$common_anc(p)

Commutativity can be exploited, e.g., when several appearances of this method call appear in a query, i.e. we can avoid repeated execution of the same method call.

We give another example which involves set operators. Assume a powerset class *P_Person* of the class Person is given. Let the method *children* be available in the interface of *P_Person.* This method, when invoked on an instance *P* of *P_Person*, computes the union of the children of all persons in *P* and returns the result as an instance of *P_Person*. Then we can define the following equivalence (using the Union method generically available for powerset classes),

(7) P IN P_Person, Q IN P_Person:
(P$\rightarrow$children())–>Union(Q$\rightarrow$children()) $\equiv$ (P–>Union(Q))$\rightarrow$children()

In this case the optimization can apply the rule to reduce the effort of duplicate eliminations, when applied on the parent level as opposed to the children level.

8 Optimization and Execution of Queries: An Outlook

The query optimizer is described by three different components. The algebra with its equivalences, the cost model, which can be considered as partial description of the execution of methods and operators (see also [23]), and the search strategy. We have already argued that for a uniform presentation of the algebra also the system-defined operators of the query algebra, i.e., at least their interfaces, should be provided as methods. Therefore it is quite natural to provide all equivalences, also those only affecting the kernel algebra, within database schemas.

As far the cost model is concerned, it is also natural to specify the cost of user-defined methods within the database schemas, as for those methods the execution model is clearly defined by the execution model inherent to VML. However, this model, which is straightforward evaluation of expressions, is not appropriate for processing large amounts of data. Therefore a different execution model, e.g., based on pipelining strategies, will be employed for the set-processing operators of the kernel algebra. So the cost model for these operations will be provided in the form of interfaces, whereas the implementation of the cost model, as well as the implementation of the operations themselves, are provided by the system components of the database system. However, in the long run, execution models for processing large data sets including their specification within database schemas will emerge from a very different area. VODAK is currently also extended to support multimedia datatypes [2], in particular continuous data, which requires new execution paradigms, e.g., dataflow oriented approaches. It will be an interesting direction of research to find a common basis for these approaches and approaches that are applied in processing of large data sets as they occur in query execution.

Up to this point we have discussed the integration of two of the three components of query optimizers, namely algebraic equivalences and cost models, at least based on their interfaces, into the VML data model. Of course also search strategies can be specified within object-oriented data models, which was already proposed in [36]. This last step would make the query optimizer a component described individually for each schema within the VML data model. Thus the VML description of the components of the query optimizer (and eventually of the query execution model) will become a self-descriptive component of a database similar to the data dictionary. For each schema the description of the query optimizer components is different. Therefore inspecting the corresponding information at run-time would be required if we provide the optimizer as a fixed component of the database system. This is not desirable as the optimizer itself has to perform as efficiently as possible. Thus we plan to generate for each schema an *individual optimizer,* and consider an optimizer generator (a kind of compiler) as a system component of the database management system.

9 Conclusion & Future Work

In this paper we have described the impact of methods on the different components of query processing in an object-oriented database management system. Using a modular reference architecture we have described the impact on different parts of query processing. For the query language VQL we have identified several relaxations from the strict SQL paradigm that become necessary when considering methods. Among these are dismissing a distinction between select and update queries, arbitrary method calls at any place in the query, and, resulting from this, dependency restrictions in the FROM clause. Then we have presented a completely new approach to implement query algebras within an object-oriented data model, which allows to represent set operators as methods. We have identified necessary features of the data model in order to support this approach, namely powerset classes, function domains and metaclass mechanisms. This approach allows homogeneous access to system-defined and user-defined operations. We have also clarified the role of parameters of set operators used in query algebras and described how they become part of the method algebra. Based on the method algebra we have identified three typical ways how semantic knowledge about schemas can be expressed in form of equivalences involving methods, which can subsequently be exploited in optimization. Thus we are giving the notion of semantic query optimization a new dimension in the context of methods. Finally, we have sketched a picture describing our overall goal: an architecture for query processing, which is as extendible and modular as object-oriented database management systems claim to be and which exploits the object-oriented data model for the specification of query optimizers individualized and tuned toward specific applications.

We are currently implementing a query processor based on VQL in the VODAK system. As a starting point for an implementation of the concepts presented in this paper we plan to use the Volcano Query Optimizer Generator [24] since it allows rapid development of a query optimizer. It also supports to a large degree extendibility with freely definable algebras and rule systems and interfaces to freely definable support functions.

Still many questions have to be solved towards reaching our goals. First, the necessary extensions of the data model have to be provided, e.g., for representing equivalences, cost models, search strategies and finally execution models. Then the appropriate search strategies have to be investigated. In this regard a central question will be an appropriate framework for the specification of equivalence patterns. Among the questions we have not considered in this paper are the treatment of nested queries, the management and reuse of query results, and the impact of updates on the semantics of queries. In the future also query processing with time dependent data, which will appear in multimedia applications, will have to be considered. Methods in queries are only a first step in this direction. Nevertheless, we think that we provide in this paper a framework which will enable to study such questions in a systematic manner.

References

[1] K. Aberer, G. Fischer: “VQL: VODAK Query Language, Version 1.0”, internal report GMD–IPSI, 1993.

[2] K. Aberer, W. Klas: “The Impact of Multimedia Data on Database Management Systems”, IEEE Workshop on Multimedia Computing, Pittsburgh, PA, 1993.

[3] K. Aberer, E.J. Neuhold, W. Klas: “Object–Oriented Modeling for Hypermedia Systems using the VODAK Modelling Language (VML)”, to appear in Object–Oriented Database Management Systems, NATO ASI Series, Springer Verlag Berlin Heidelberg, August 1993 .

[4] F. Bancilhon, S. Cluet, C. DelobelL: “A Query Language for the O_2 Object-Oriented Database System”, *Proceedings of the 2nd International Workshop on Database Programming Languages*, pp. 122–138, Salishan Lodge, Oregon, USA, 1989.

[5] J. Banerjee, W. Kim, K. Kyung-Chang: “Queries in Object-Oriented Databases”, *Proceedings of the IEEE 4th International Conference on Data Engineering*, pp. 31–18, Los Angeles, USA, February 1988.

[6] C. Beeri: “A formal approach to object-oriented databases”, *Data & Knowledge Engineering 5*, pp. 353–382, 1990.

[7] E. Bertino, C. Guglielmina: “Path–index: An Approach to the efficient execution of object-oriented queries”, *Data & Knowledge Engineering 10*, pp. 1–27, 1993.

[8] E. Bertino, W. Kim: “Indexing Techniques for Queries on Nested Objects”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 2, pp. 196–214, June 1989.

[9] E. Bertino, M. Negri, G. Pelagatti, L. Sbattella: “Object-Oriented Query Languages: The Notion and the Issues”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 3, pp. 223–237, June 1992.

[10] E. Bertino, A. Quarati: “An Approach to Support Method Invocations in Object-Oriented Queries”, *Proceedings of IEEE 2nd International Workshop on Research Issues on Data Engineering – Transaction and Query Processing (RIDE-TQP)*, pp. 163–168, Phoenix, Arizona, February 1992.

[11] E. Bertino, D. Musto: “Query optimization by using knowledge about data semantics”, *Data & Knowledge Engineering 9*, pp. 121–155, 1992/93.

[12] J.A. Blakeley, W.J. Kenna, G. Graefe: “ Experiences Building the Open OODB Query Optimizer”, preprint, Dec 1992.

[13] S. Cluet, C. Delobel: “A General Framework for the Optimization of Object-Oriented Queries”, *ACM SIGMOD 1992*, pp.383–392, 1992.

[14] S. Cluet, C. Delobel, C. Lécluse, P. Richard: “ RELOOP, an algebra based query language for an object-oriented database system”, *Data & Knowledge Engineering Journal 5 (1990)*, pp. 333–352.

[15] M. Carey, D. DeWitt. S. Vandenberg: “A Data Model and Query Language for EXODUS”, *Proceedings of the ACM SIGMOD Conference*, pp. 413–423, Chicago, USA, 1988.

[16] C. Delobel, C. Lécluse, P. Richard: “LOOQ: A Query Language for Object-Oriented Databases, Informal Presentation”, *Proc. of the ACFET Conference on Knowledge and Object-Oriented Database Systems*, pp. 333–352, Paris, France, December 1988.

[17] K.R. Dittrich (Ed.): *Advances in Object-Oriented Database Systems / Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, pp. 358–363, Bad Münster am Stein-Ebernburg, Germany, September 27–30, 1988 (LNCS 334)

[18] G. Fischer: “Updates in Object-Oriented Database Systems Caused by Method Calls in Queries”, *Proceedings of the 3rd ERCIM Database Researcg Group Workshop on Updates and Constraints Handling in Advanced Database Systems*, Pisa, Italy, September 28–30, 1992.

[19] J.C. Freytag: “A Rule-Based View of Query Optimization”, *Proceedings of the ACM SIGMOD Conference*, pp. 173–180, San Francisco, USA, May 27–29, 1987.

[20] G. Gardarin, R. Lanzelotte: “Optimizing Object-Oriented Database Queries using Cost-Controlled Rewriting”, *Proceedings of the 3rd International Conference on Extending Database Technology (EDBT ’92)*, pp. 534–549, Vienna, Austria, March 1992.

[21] G. Gräfe: “Volcano, an Extensible and Parallel Query Evaluation System”, *Technical report CU–CS–481–90*, University of Colorado at Boulder, 1990.

[22] G. Gräfe, D. DeWitt: “The EXODUS Query Optimizer”, *Proceedings of the ACM SIGMOD Conference*, pp. 160–172, San Francisco, USA, May 27–29, 1987.

[23] G. Gräfe, D. Maier: “Query Optimization in Object-Oriented Database Systems: A Prospectus”, pp. 358–363 in [17]

[24] G. Gräfe, W. J. McKenna: “The Volcano Optimizer Generator: Extensibility and Efficient Search”, *Proceedings of the 9th IEEE International Conference on Data Engineering*, pp. 209–218, Vienna, Austria, April 19–23, 1993.

[25] T. Härder, B. Mitschang, H. Schöning: “Query processing for complex objects”, *Data & Knowledge Engineering 7*, p. 181–200, 1992.

[26] A. Heuer, J. Fuchs, U. Wiebking: “OSCAR: An object-oriented database system with a nested relational kernel”, *Proceedings of the 9th International Conference on Entity-Relationship Approach*, pp. 95–110, Lausanne, Switzerland, October 1990.

[27] A.Heuer, M. Scholl: “Principles of Object-Oriented Query Languages”, *GI–Fachtagung, Datenbanksysteme für Büro, Technik und Wissenschaft*, pp. 178–197, Kaiserlautern, Germany, March 1991.

[28] International Standards Organization: “Database Language SQL2 and SQL3”, *international committee document*, ISO/IEC JTC1/SC21 WG3 DBL SEL–3b, April 1990.

[29] M. Jarke, J. Koch: “Query Optimization in Database Systems”, *ACM Computing Survey, vol. 16, no. 2*, pp. 111–152, June 1984.

[30] A. Kemper, G. Moerkotte: “Advanced Query Processing in Object Bases Using Access Support Relations”, *Proceedings of the 16th International Conference on Very Large Databases (VLDB ’90)*, pp. 290–301, Brisbane, Australia, 1990.

[31] M. Kifer, W. Kim, Y. Sagiv: “Querying Object-Oriented Databases”, Proc ACM SIGMOD, 1992.

[32] W. Kim, B.P. Jenq, D. Woelk, W.-L. Lee: “Query Processing in Distributed ORION”, *Proceedings of the International Conference on Extending Database Technology (EDBT ’90)*, Venice, Italy, 1990.

[33] W. Kim: “A Model of Queries for Object-Oriented Databases”, *Proceedings of the 15th International Conference on Very Large Databases (VLDB ’89)*, pp. 423–432, Amsterdam, The Netherlands, August 1989.

[34] W. Klas: “A Metaclass System for Open Object-Oriented Data Models”, *Dissertation*, Technical University of Vienna, January 1990.

[35] H. F. Korth: “Optimization of Object-Retrieval Queries”, pp. 352–357 in [17]

[36] R. Lanzelotte, P. Valduriez: “Extending the Search Strategy in a Query Optimizer”, *Proceedings of the 17th International Conference on Very Large Databases (VLDB ’91)*, Barcelona, Spain, 1991.

[37] L. Liu: “A formal approach to Structure, Algebra and Communication of Complex Objects”, PhD Tilburg University, 1993.

[38] D. Maier: “The Theory of Relational Databases”, Computer Science Press, 1983.

[39] S.L. Osborn: “Identity, Equality and Query Optimization”, pp. 346–351 in [17].

[40] M. Scholl, C. Laasch, M. Tresch: “Updatable views in Object Oriented Databases”, *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, (DOOD–2)*, pp. 189–207, Munich, Germany, December 1991.

[41] G. Shaw, S. Zdonik: “Object-Oriented Queries: Equivalence and Optimization”, *Proceedings of the 1st International Conference on Deductive and Object-Oriented Database Systems (DOOD ’89)*, pp. 264–278, 1989.

[42] M. Stonebraker, L. Rowe: “The POSTGRES Papers”, Electronics Research Laboratory, College of Engineering; Memorandum No. UCB/ERL M86/85, pp.115 , University of California, Berkeley, USA, June, 1987.

[43] D.D. Straube, M.T. Özsu: “Queries and Query Processing in Object Oriented Database Systems”, *ACM Transactions on Information Systems, vol. 8, no. 4*, pp. 387–430, October 1990.

[44] P. Valduriez, S. Danforth: “Query Optimization for Database Programming Languages”, *Proceedings of the 1st International Conference on Deductive and Object-Oriented Database Systems*, pp. 516–534, 1989.

[45] S.B. Zdonik: “Data Abstraction and Query Optimization”, pp. 368–373 in [17].