

Query Processing in DOQL: A Deductive Database Language for the ODMG Model

Pedro R. Falcone Sampaio¹ Norman W. Paton

*Department of Computer Science, University of Manchester Oxford Road,
Manchester, M13 9PL, UK*

Abstract

This paper describes the architecture, algebraic query processing framework and query execution approach that comprise the implementation of the DOQL query processing system. To the best of our knowledge, it is the first deductive object query language to be designed and implemented as a complementary and non-intrusive query component within an ODMG OODBMS architecture. The query processing framework enables the combined use of logical rewriting and algebraic optimization, and features an object algebra, local and global query optimization, physical execution algorithms implemented as iterators, and a query execution engine implemented using the dataflow technique. Query execution is based on a non-first normal form tuple-based approach, where physical objects stored in the database are transformed into a system independent tuple that supports ODMG data and collection types as members. This approach simplifies the query engine implementation and enables its portability to different OODBMS. Example queries are provided that illustrate the flexibility and expressiveness of querying object databases with DOQL.

1 Introduction

Providing support for deductive rules in database systems has a long track record in database research. In particular, major contributions of deductive database research to the database community have been in the form of query processing and optimization techniques, sound formal foundations for query languages, and in novel ways of extending DBMS functionality.

One of the shortcomings of deductive database research has been the minimal commercial impact of deductive database systems and the limited number of

¹ Supported by CNPQ, Brazil

applications that have profited from deductive functionality. A possible reason for this is that most deductive database systems have been designed with deduction as both *the query model and the data model*, where deductive rules are the ubiquitous model for defining schema, data, I/O, and queries, and the deductive query processing data structures and algorithms are in the kernel of the DBMS. As a result, systems are often monolithic, lack certain services provided by commercial DBMS, and discourage users that could benefit from a deductive query language but are not inclined to migrate their data and programs to a non-mainstream DBMS.

An early and alternative research route for building deductive systems involved coupling the deductive database system to a host relational DBMS [16], using the host system to store data and provide the core DBMS functionality, and building the deductive system as a layer. This route tries to provide deduction as a more powerful *query and view model* that can be made available as a layer on top of a mainstream data model. However, these attempts were hindered by the fact that the deductive facilities were poorly integrated with other components of the DBMS architecture. In addition, the connection between the layers was often limited by the application programming interfaces (API) of the DBMSs, giving rise to evaluation mismatches between the coupled layers and thus performance problems.

Since then, there has been considerable progress on DBMS APIs, a substantial increase in processing power and communication bandwidth, a new branch of component-based database research [10,29,28], and new approaches to supporting rules outside the kernel of database systems [33], which provide a new context for investigating the possibility of supporting deductive rules functionality in a seamless, non-intrusive and efficient way, thereby complementing and enhancing the existing query and modelling power of database systems. The extra leverage obtained from support for deductive functionality is relevant for building database middleware for distributed information systems [60], managing semistructured data [42], and for building decision support and knowledge discovery systems [24].

To provide evidence of a “plug-in” approach to supporting deductive functionality, we have designed and implemented the deductive object query language (DOQL) as a complementary, non-intrusive deductive query component bundled in the setting of the ODMG standard. The query system is connected to the host OODBMS as a client-side component, allowing the use of deductive rules and queries that can range over objects stored in the object database without side-effects for existing data, schema and programs. The query and modelling power obtained through the addition of the deductive query component, as well as making available a new query paradigm that can be used alongside the other ODMG components, provides additional query capabilities that are not supported by OQL.

The query processing framework adopted in the implementation of DOQL employs an object algebra extended with fixpoint related capabilities, and enables the combination of deductive query rewriting techniques, with object-based algebraic optimization. The query execution engine supports a physical algebra implemented using the dataflow query processing technique. The physical algebra can accommodate different physical algorithms for implementing each logical algebra operator, and executes most of the operators on the client machine.

Query execution is based on a non-first normal form tuple-based approach, where physical objects stored in the database are transformed into a platform independent tuple structure that supports ODMG data and collection types as members. This approach simplifies the query engine implementation and enables its portability to different object-based DBMS.

The overall approach taken in the design and implementation of DOQL is based on the view that the deductive database paradigm can coexist harmoniously and orthogonally with mainstream data models and that mainstream query processing techniques can also be employed in the provision of the deductive functionality.

The remainder of this paper is structured as follows. Section 2 presents a range of DOQL queries over an object database, illustrating the expressive power and query features supported by DOQL. Section 3 describes the rationale and approach to providing the query component in the context of the ODMG architecture. Section 4 covers the algebraic query processing framework underlying DOQL, followed by a description of the query execution engine implementation in section 5. Related work is presented in section 6, and the paper concludes in section 7 with a summary of the work.

2 Querying Object Databases With DOQL

DOQL is a typed deductive language designed for querying ODMG databases. The query language supports features such as conjunction, navigation using path expressions, unnesting of nested collections, nested queries, disjunction, negation, aggregates, built-in arithmetic predicates and recursive view definition. Its design caters for interactive ad-hoc queries or for embedded use within ODMG imperative language bindings.

From the language design perspective, DOQL follows the language integration approach [7,52], where the deductive language is integrated with imperative programming languages in the context of an object model or type system. By providing deductive capabilities following the language integration approach

within the setting of an ODMG compliant OODBMS, the database programmer can select which portions of the application will use the deductive features, without the need to migrate or affect other parts of the application.

Parts of an application that are less well suited to the deductive approach can be implemented using other components of the ODMG architecture, and the interactive user also benefits from an additional query language interface that can be used for ad-hoc queries.

The DOQL language style supports constructs available in object query languages (path expressions and method calls) as well as constructs found in earlier deductive object-oriented languages such as molecules (F-Logic [36]) and set expressions (Chimera [17]). Some constructs like molecules and path expressions provide alternative ways of expressing similar queries, and the support for both in DOQL aims to enable a wide range of choices in formulating queries.

This section shows the use of DOQL to query a small database that keeps information regarding trains, stations and visits, followed by a brief description of the language structure and the main definitions used along the paper. For an extended overview of DOQL see [46,53]. The schema of the application is defined in figure 1 using the UML notation [11].

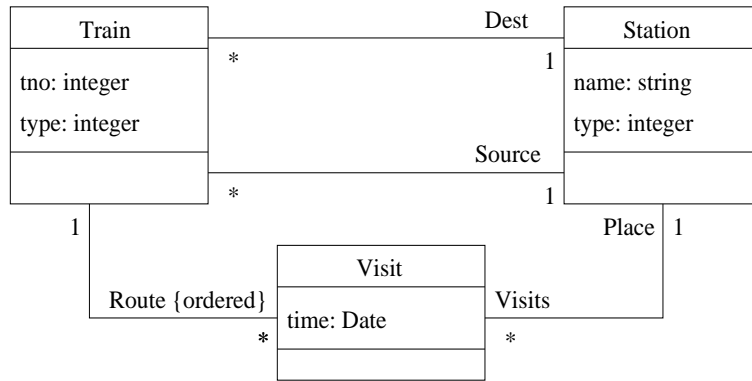


Fig. 1. UML description of the *Trains* application

The object database schema derived from the conceptual design of the application is described in figure 2, and the OODBMS used as underlying storage mechanism is Poet5.0 [48]. The version of the ODL compiler of the DBMS is compatible with the ODMG 1.2 standard [47]. In the ODL schema, attributes are used to represent relationships since ODMG relationship semantics is not yet fully supported by the ODL compiler. From the perspective of the query syntax of DOQL, the representation of ODMG relationships in the underlying DBMS as attributes or as proper relationships is irrelevant, although relationship support can enhance query optimization due to the presence of inverses.

<pre> interface Train (extent train key tno) : persistent { attribute long tno; attribute Station source; attribute Station dest; attribute list<Visit> route; attribute short type; }; </pre>	<pre> interface Station (extent station key name) : persistent { attribute string name; attribute set<Visit> visits; attribute short type; }; </pre>	<pre> interface Visit (extent visit) : persistent { attribute Station place; attribute d_Date time; attribute Train train; }; </pre>
--	--	--

Fig. 2. *Trains* database schema

The following queries illustrate the query expressiveness of DOQL. All queries described are operational in the alpha version of the DOQL query system.

Q1: *retrieve the name and the type of all stations.*

```
query(Name,T) :- station(X), X.name = Name, X.type = T.
```

Q1 uses the collection formula `station(X)` to bind `X` to the OID of each object of the extent of `station`, and the variables `Name` and `T` are bound to data fields of the object represented by `X`. The query is composed using conjunctive query syntax, which is straightforward to use and enables the incremental composition of a query.

■

Q2: *retrieve the train number of all visits to London.*

```
query(Train) :- visit[place.name="london", train.tno=Train].
```

Q2 uses the molecular construct to specify the information to be retrieved, starting from the `visit` type. The molecular construct is useful for grouping information related to objects of the same type.

■

Q3: *retrieve the time and place name of all visits whose train number is smaller than 3104104 and whose train's destination type is smaller than 2.*

```
query(P,T) :- visit(V), V.place.name = P, V.train.tno < 3104104,
              V.train.dest.type < 2, V.time = T.
```

Q3 employs path expressions that navigate through several class extents. The model of path expressions in DOQL is akin to OQL in the sense that only single valued relationships can be traversed. Path expressions can be used in comparisons, in arithmetic, or to unify a variable with a value from the database.

Q4: *retrieve the train number of all trains that include Aberdeen in their route.*

```
query(Train) :- T.tno = Train, visit(R) [place.name="aberdeen"],
               train(T) [route=>R].
```

Q4 employs unnesting of the nested collection `route` to obtain the desired result. Matching of the variable `R` is used to relate the objects obtained from the extent of `visit` to the objects unnested from the nested collection.



Q5: *retrieve the train number of all trains that include Aberdeen in their route.*

```
query(TNO) :- train(T), T[route=>R[place.name="aberdeen"]],
              T.tno = TNO.
```

Q5 has the same result as **Q4**, but it uses a nested query inside a molecule to specify the desired query. DOQL supports unlimited levels of nesting inside queries, which can simplify and decrease the query size when querying nested information structures which are frequently present in object databases.



Q6: *retrieve the train number of all trains that include Aberdeen or London in their route.*

```
q(Train)      :- visit(R) [place.name="aberdeen"],
                 train(T) [tno=Train,route=>R].
q(Train)      :- visit(R) [place.name="london"],
                 train(T) [tno=Train,route=>R].
query(Train) :- q(Train);
```

Q6 uses two identical rule heads as views defining portions of the results of a disjunctive query. The view mechanism supported by the deductive rule paradigm is useful for defining complex relationships among stored objects and also for the incremental composition of complex queries.



Q7: *retrieve the train number of all trains that include London but not Aberdeen in their route.*

```
t_place(Train,Place) :- visit(R) [place.name=Place],
                        train(T) [tno=Train,route=>R].
query(Train)          :- t_place(Train,"london"),
                        not t_place(Train,"aberdeen").
```

Q7 illustrates the use of negation in DOQL.



Q8: *where can you reach departing from Aberdeen? Retrieve the names of the destinations.*

```
reaches(S,P) :- train[source.name=S,dest.name=P].
reaches(S,P) :- reaches(S,L),
                  train[source.name=L,dest.name=P].
query(P)      :- reaches(S,P), S = "aberdeen".
```

Q8 illustrates the computation of a transitive closure operation over the connected (and possibly cyclic) object structures stored in the database. The support for fixpoint queries in DOQL allows to express recursive queries that are not supported in OQL, and that are also difficult to program using an imperative binding.



Q9: *retrieve the train number and the type of all trains where the sum between the number of visits in the route of a train and the number denoting the type of the train is greater than 110.*

```
query(T,TP) :- train(X1)[tno=T,type=TP],
               110 < count(X1.route) + TP.
```



Q9 illustrates the use of aggregate functions and arithmetic expressions. In the query, the `count` aggregate function is applied to compute the number of visits in the route of a train, and this value is added to the variable that is bound to the type of the train.

2.1 Terms

DOQL terms are based on data and operations that are definable using the ODMG object model [15]. The following terms are supported:

- *Variables:* represented by alphanumeric sequences of characters beginning with an uppercase letter (e.g. `X`, `Names`).
- *Atomic Constants:* representing ODMG atomic literals² (e.g., `2.32`, `"Alan`

² The word *literal* in the ODMG specification refers to a type without object identity. In the context of DOQL, a literal is a negated or non-negated atomic formula

Turing", 13, true, nil).

- *Compound Constants*: representing structured or collection literals defined in the ODMG model. (e.g., `struct("Eve", 21)`, `set(1,2,3)`, `bag(1,1)`, `list(1,1,2,4)`).
- *DB entry names*: representing the set of names associated with collections and extents in an ODMG schema.
- *Expressions*: evaluable functions that return an ODMG structural element (literal or object). Each expression has a type derived from the structure of the query expression, the schema type declarations or the types of named objects. Expressions can have other terms as parameters and are divided into the following categories:

path expressions: traversal paths starting with a variable denoting an object, or starting with an object-valued property that can span over attributes, operations and relationships defined in type signatures of an ODMG schema (e.g., `train.tno`, `source.name`). Path expressions traversing single valued features in type signatures can be composed thereby forming paths of arbitrary lengths (e.g., `V.train.dest.type`).

aggregate expressions: predefined functions applied to terms denoting collections. The following expression shows the use of an aggregate operator: `count(X.route)`.

arithmetic expressions: $(exp_1 + exp_2)$, $(exp_1 * exp_2)$, (exp_1/exp_2) , $(exp_1 - exp_2)$

set expressions: $(exp_1 \text{ union } exp_2)$, $(exp_1 \text{ intersect } exp_2)$, $(exp_1 \text{ except } exp_2)$

DOQL terms range over ODMG literal types, structured literal types, collection literals, NULL type, atomic objects, and collection objects. Although DOQL terms can span types defining operations, only operations without side-effects are allowed to be called in a DOQL formula. Release 2.0 of the ODMG standard introduces the notions of *Interfaces* as additional types participating in the ODMG type system. Given the fact that interfaces cannot have object instances, they also do not have extents, so DOQL terms do not range over interface types. The operations defined in an interface type can be accessed in DOQL through an object that is an instance of a concrete class that inherits from the interface type.

2.2 Literals

Literals denote atomic formulas or negated atomic formulas that define propositions about individuals in the database. The atomic formulas in DOQL are:

- *collection formulas*: enable ranging over elements of collections. A collection in the body of a deductive rule.

can be a *virtual view* [1] defined by a previous rule or a *stored collection* defined in the extensional database. Molecular groupings of collection properties can be done using the operator (`[]`). The operator (`=>`) can be used to finish path expressions that end in a collection, resulting in the unnesting of a nested collection into the scope of an outer collection. The following formulae show the diversity of possibilities in ranging over collections, depending on the needs of each rule. In the first formula, the identities of the collection instances are bound in turn to `X`. In the second formula, the items needed are the values for the properties `tno` and `type` of each collection instance. The third formula binds variables to identities and properties of the elements in the collection. The fourth formula denotes the unnesting of the visits of `Y`. The last formula denotes a virtual view.

```
train(X)
train[tno=TNO, type=T]
train(X)[tno=TNO, type=T]
Y.route=>X
reaches(X,Y)
```

- *operation formulas*: denote boolean formulas of type $\langle Term \rangle$ Operator $\langle Term \rangle$. Operation formulas encompass membership relationship between elements and collections (`in`), string related relationships (`substr`, `like`, `lower`), value comparisons (`<=`, `>=`, `<`, `>`, `!=`), and equality (`=`). The equality operator is overloaded and also performs a matching operation [38] when one of the terms is bounded and the other is free. Matching over object-valued properties (attributes, relationships) and methods in the extensional database are denoted by the single arrow operator (`->`). The following formulae are examples of operator formulas. In the first formula, `X` is equated the identity of `Y`'s source. In the second formula, `X` is equated to the identity of the first train of `Y`. The third formula denotes the truth of the membership of `X` in the collection of `Y` visits.

```
Y.source->X
Y.first_train()->X
X in Y.route
```

2.3 Rules, Rule Sets, Partitions and Programs

A DOQL *rule* is an expression of the form: $L_1 \leftarrow L_2, \dots, L_n$, where $n \geq 1$, and L_1, \dots, L_n are DOQL literals. The head of a rule is a literal denoting an intensional database predicate (virtual view) and the remaining literals form the body of a rule. Only intensional database predicates can appear as rule head literals, while any literal can appear in the body of a rule. A literal appearing in the body of a rule is positive or negative depending on the presence of negation.

A DOQL *rule set* is a set of DOQL rules. A *partition* of a rule set is a subset

of the rule set where all rules in the subset have the same predicate as rule head. A DOQL *program* is a rule set with one of the rules denoting the query predicate. This rule has the reserved word `query` as rule head, and only one rule with the `query` predicate in the rule head is allowed in a DOQL program.

2.4 Dependency Graphs, Stratification and Linearity

Derived information in DOQL is organized as virtual views which are analogous to intensional database predicates in Datalog-based languages. Given this similarity, techniques for obtaining dependency graphs for Datalog such as the ones described in [16] can be used to obtain dependency graphs for DOQL programs:

The *dependency graph* for a DOQL program P is the directed graph $DG_p(V, E)$ such that:

- (1) V is the set of virtual view predicates in P .
- (2) $E = \{ \langle v_i, v_j \rangle \iff v_j \text{ occurs in the body of a rule defining } v_i \}$.

The *extended dependency graph* for a DOQL program P is the directed graph $EDG_p(V, E)$ such that:

- (1) V is the set of virtual view predicates in P .
- (2) $E = [+,-]\{ \langle v_i, v_j \rangle \iff v_j \text{ occurs in the body of a rule defining } v_i \}$.
If v_j occurs negated then the label (-) is attached to the edge.

A DOQL program P is *recursive* if the dependency graph DG of P contains at least one cycle. Given a DOQL rule $R: p \leftarrow q_1, \dots, q_n$, R is *linear* in q_i ($1 \leq i \leq n$) if there is at most one occurrence of each virtual view predicate q_i in the rule body. A DOQL program P is *linear* if all rules in P are linear. A DOQL program P is *stratified* if the extended dependency graph EDG of P does not contain any cycle involving an edge labelled with (-).

2.5 Range Restriction and Safety

In deductive query languages, it is often the case that syntactically valid rules may be satisfied by an infinite set of instances. Since a query with infinite answers or a virtual view with an infinite extension has no practical semantic meaning, it is necessary to identify a subset of all possible rules which is guaranteed to be satisfied by finite sets of instances when the rule is applied over a finite database. Such subset of rules is considered range restricted. For a rule to be range restricted, all its variables need to be range restricted. The

following definitions address this issue in the DOQL context:

A variable X is *range restricted* in a DOQL rule F if:

- (1) X occurs as variable in a collection formula.
- (2) X occurs in an equality literal inside a collection formula.
- (3) X occurs in an equality $X = T$ where T is a path expression starting with a range restricted variable.
- (4) X occurs in an OID matching equation (e.g., $Y.father \rightarrow X$).

A DOQL formula F is *range restricted* if all variables appearing in F are range restricted.

Each DOQL rule defines a local scope from which the bindings for each variable are computed. To assure that every variable can have its bindings computed from the data sets appearing locally in a rule, safety conditions need to be defined. An example of an unsafe DOQL rule is:

```
unemployed(X) :- not employee(X).
```

The rule above implies that every object that does not belong to the extent of the class `employee` should belong to the extent of the virtual view, thus, generating an unintended result. The following definition specifies safety conditions that need to be satisfied by each DOQL rule individually:

A DOQL rule R is *safe* if the following items are satisfied:

- (1) R is range restricted.
- (2) All variables appearing in the rule body are range restricted.
- (3) Any variable that appears in the head of R occurs in at least one non-negated literal of the body of R .

A DOQL program P is *safe* if every rule in P is safe. A DOQL program P is *valid* if P is safe and stratified.

The safety definitions described above handle the problem of evaluable arithmetic predicates being interpreted as infinite tables, and also the possibility of a negation causing the same unintended interpretation for a rule. The safety notion in DOQL is similar to the notion of strongly safe rules for deductive databases as described in [5], which are guaranteed to have an evaluation strategy which will compute the answers and terminate (effectively computable). In DOQL, safe programs are effectively computable provided that all method calls in a DOQL rule are also effectively computable.

The safety notion also implies that a DOQL rule set is *bottom-up evaluable*, where the evaluation of the rule set can be carried out using a bottom-up

strategy, without materializing infinite intermediate results. The computation proceeds in a strictly bottom-up manner, using values for the body variables to produce values for the head variables, with the guarantee that the set of values for the body variables is finite at each step [5].

3 The DOQL System Architecture

Two alternative architectural strategies have been employed for implementing deductive database systems: *coupling* is the development of an interface between the deductive subsystem and the host DBMS providing a single-system image, but preserving the individuality of both subsystems; *integration* is the development of a single system where the data structures and algorithms related to the mass storage are specially tailored to the deductive language query processing model [16].

Recent research into component-based architectures for databases proposes *bundling* as a “construction theory” that allows the assembling of persistent systems out of reusable components in a cost-effective way [29]. This approach advocates a new way of providing functionality that differs from building from scratch or extending existing systems through layers, which have traditionally been employed in DBMS research, and has often resulted in inflexible monolithic systems and inefficient deeply layered systems, respectively.

While bundling tries to achieve the same goal as coupling does, bundling goes a step further than just providing an interface that allows one of the subsystems to be built on top of the other (layered systems). Bundling also tries to obtain components that are minimal, self contained, and usable both within their original context (e.g a deductive query component in a deductive database system), and also in other contexts (e.g. a deductive query component that can be easily ported to any ODMG compliant OODBMS). Furthermore, the rationale behind the design and connection of bundled components seeks to avoid evaluation and impedance mismatches that are normally associated with coupling.

3.1 Isolating Deductive Functionality: Unbundling

To isolate the core deductive functionality that can be bundled as a query component while leaving other elements of a typical DBMS architecture alone, existing deductive object-oriented database system (DOOD) architectures were analysed and placed along the main DBMS components: storage manager, transaction manager, and query processor [55], to assess the degree of inter-

vention in the main DBMS components that is necessary to build the deductive engine. Three classes of systems emerged:

- **Monolithic:** systems designed using the integration approach where the deductive database engine is responsible for storage management, transaction processing, and query processing. In this class of systems, the three major components of a DBMS architecture have to be implemented from scratch to provide deductive functionality. Examples of systems belonging to this class are ConceptBase [34], Florid [26], LAURE [14], Quixote [68], ROL [41] and Validity [25].
- **Deeply Layered:** systems designed using the coupling approach, where part of the functionality related to transaction processing and storage management is unbundled from the deductive engine. The presence of deduction affects mainly the query processor, but the presence of update capability in the declarative language results in replication of the transaction manager functionality, since the deductive system needs to define its own transaction model. Examples of systems in this class and their associated storage and transaction subsystems are Chimera [17] (Phoenix), CDOL [64] (Shore) and Logres [12] (Algres).
- **Layered:** systems designed using coupling and that also remove the functionality described above, but differ from the deeply layered ones by interfacing an imperative language to the declarative language, to unbundle from the deductive engine the update functionality and part of the I/O responsibility. Examples of systems in this class and their associated subsystems are Coral++ [56] (Exodus), Noodle [44] (Sword) and Rock & Roll [7] (Exodus).

Monolithic system architectures need to address implementation issues relating to the three main components of the DBMS architecture, replicating DBMS functionality. Layered architectures rely on existing subsystems for implementing parts of the architecture, but often need to replicate functionality associated with transaction and storage management. In layered systems, this replication is a main source of performance losses since the functionality replicated in distinct layers is part of the run-time image of the system.

As well as the need to replicate DBMS functionality, another drawback stems from the fact that in all systems previously described, the 3 levels of the ANSI/SPARC architecture are affected due to the deductive rules model being considered the data model of the system (monolithic systems), or because the systems are developed adopting non-standard data models (layered systems). As a consequence, migration of existing data and applications is often necessary to use the deductive database system.

The central question that is investigated in this paper and which represents the main contribution is:

How to provide deductive functionality as a complementary query and view service available within the context of a mainstream database model without replicating DBMS functionality, without intervention of deduction in the physical and conceptual levels of an OODBMS architecture, and without sacrificing performance or seamlessness of integration in the process.

The DOQL project tries to stretch the unbundling process, by designing the deductive engine bundled to an ODMG compliant OODBMS using the host's transaction and storage management mechanisms to build a client-side query component. The query component is implemented as a query processor that executes in application mode, subject to the host's transaction manager, storage manager, and access control mechanisms, without being intrusive to these components. By adopting the ODMG model and corresponding schema definition language (ODL) as the type system underlying the deductive language, DOQL can query existing application schemas and stored data in ODMG compliant databases, avoiding the need for migration.

3.2 Bundling the Deductive Component

The position of DOQL in the ODMG architecture is illustrated in figure 3. The architecture supports the synergy of different paradigms: deductive queries can be used to support complex ad-hoc and embedded queries, and to provide a straightforward view mechanism. The imperative language bindings are used to populate and update the database, build applications using the available query components (OQL, DOQL), provide implementations for class methods and implement user interfaces, while ODL is used to specify the database schema.

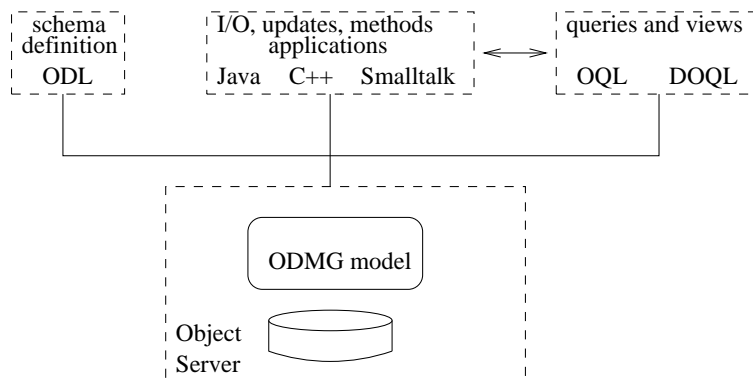


Fig. 3. Complementary and non-intrusive architecture

It is important to stress that the main distinction between layered and complementary architectures (DOQL) is the definition of a specific data model for the DOOD system in the layered architecture. In the layered architecture it is

not possible to use the deductive language to query data stored in the persistence service that does not conform to the DOOD data model, and there may be difficulties in using other services available within the persistence service such as programming language bindings.

The rationale behind paradigm integration in a complementary and non-intrusive system architecture is expressed by the following argument stated by Ullman and Zaniolo:

A declarative formulation cannot compete with the cogency and optimality of textbook algorithms for specific problems. These situations call for a mixed mode, and for the harmonious cooperation between the two modes at the language and system levels [61].

A major achievement in the bundling of DOQL in the ODMG framework is the harmonious co-existence of the imperative, deductive, functional (OQL) and schema definition components – there is no need to alter the defining principles of the component paradigms in order to achieve their integration, and the architecture was realized without ownership of the source code of the OODBMS that hosts DOQL.

3.2.1 Avoiding Mismatches

Developing deductive database systems by coupling subsystems has often resulted in less efficient systems than the monolithic ones obtained using the integration strategy. The inefficiency can generally be attributed to replication of functionality in distinct layers, and excessive levels of mismatch as shown in figure 4, which illustrates levels of mismatches in a typical coupled deductive database system architecture.

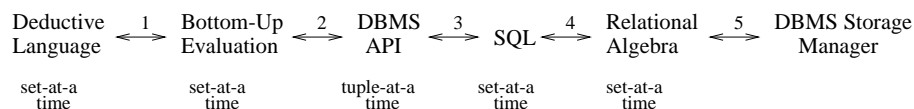


Fig. 4. Evaluation mismatch in coupled deductive databases

The key to overcoming this drawback is to eliminate extra levels of translation by using an API that provides direct access to the object manager. Core features needed from the API are:

- dynamic access of objects without the need of pre-compiled header files (class factories) for the class describing the object;
- sequential and indexed retrieval of objects in a class extent;
- navigation between objects using OIDs;
- retrieval of objects inside a collection;
- interrogation of the metatype and index hierarchy; and

- dynamic calling of a method of an object.

The last item is hard to achieve in the context of object databases that allow the methods to be written in different languages which can support different processing modes (compiled and interpreted). Some APIs that support method calls do not maintain type information about number and type of method arguments, increasing the chances of a runtime type error.

By accessing the object manager directly using an API that supports the core features described above, it is possible to build an object algebra on top of the object manager, eliminating levels 3 and 4 described in figure 4, and replacing them by the level that realizes the direct interface between the API and the object manager. Since the API provides an object-oriented interface to the object storage manager within an object-oriented language like C++, the impedance mismatch that exists when embedding SQL calls in C is eliminated.

The next short cut over the phases of figure 4 can be obtained by recasting the problem of bottom-up evaluation of a deductive language as an algebraic query processing problem, merging levels 2 and 5. This is done in DOQL by incorporating into the algebra, operators such as `fixpoint` of an intensional database (IDB) [66,1] predicate, and an operator that computes naive or semi-naive evaluation of the algebraic counterpart of a rule set, depending on the linearity property of the rule set.

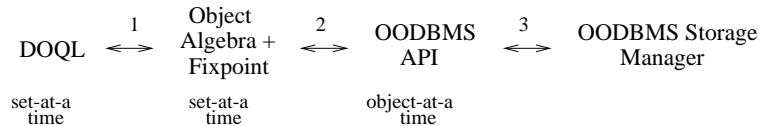


Fig. 5. DOQL/OODBMS interface

The strategy used to interface the DOQL component to a host OODBMS is depicted in figure 5. This strategy preserves all the non-intrusive and subsystem independence benefits of coupling, and from the seamlessness point of view, is comparable to that achieved in monolithic systems. This stems from the fact that the DOQL object algebra is a specially tailored algorithmic structure that enables deductive inferences over the stored objects without evaluation mismatches between the connected components. Details regarding the communication between the DOQL component and the host OODBMS are provided in section 5.

4 Query Processing Framework

This section provides a background on representations and transformations used in query processing, and describes the algebraic framework used in pro-

cesing DOQL queries.

4.1 Background

There is a considerable semantic gap between a declarative query language and the physical query execution routines supported by the DBMS storage manager. Query processing representations [35] are used to fill the gap between the query language and the target language defined by the DBMS execution routines. The choice of query processing representations and sequence of steps defines a framework or methodology for query processing [45].

The most popular representation employed in mainstream query processing methodologies is a database algebra, which has a natural representation as a tree or graph of operators, and acts as an intermediate language, where operations can be rearranged before they are translated into calls to DBMS execution routines. Benefits derived from an algebraic approach include the possibility of investigating the expressiveness of the query model based on the theoretical properties of the algebra, the provision of an operational semantic definition for a query language that is mapped into the algebra, and the availability of extensive results regarding algebraic query transformations in query optimization.

To be effective, an algebra must be able to range over all data structures supported by the underlying data model (coverage), to query different collection types in a uniform manner (uniformity), and to generate elements of the underlying data model as output results of the application of an operator (closure). Other representations that are also employed in query processing methodologies are:

- *Rule/Query language*: deductive database languages transform the rules (queries) during recursive query optimization, applying logical rewriting techniques such as magic sets to obtain more optimized versions of the same rule set. In SQL, a query is often expanded to incorporate information relating to integrity constraints and views that can be used in the optimization stage.
- *Calculus*: calculus can be used as the first internal representation in the query processing of SQL-based languages, where nested queries are simplified and redundancies are eliminated. In query processing methodologies for object query languages that support nested queries like OQL [15] or EXCESS [65], as well as the object algebra, it is also common to use a calculus-like representation as an intermediate language, performing simplifications and transforming the query into a canonical form that can more easily be translated into the object algebra.

- *Query graphs*: query processing methodologies for SQL often parse the queries into a graph representation to flatten queries and exploit properties of queries such as correlation [40]. Visual query languages also use query graphs in their query processing methodology. Deductive database languages can map the rule set to a rule-goal graph representation that is useful in the analysis of properties such as linearity, mutual recursion, and stratification. Some methodologies also employ the graph in the evaluation stage.

As well as the choice of the representations employed during query processing, other fundamental decisions in query optimization relate to deciding which transformations to use, and in which representation to apply the transformations. There are three types of transformation rules that can be applied to a query processing representation:

- Logical rewritings: the properties of the query language or calculus language are used to transform the query into an equivalent and potentially more efficient version of the query. Queries involving redundancies, nesting and complex constructs can also be brought to a canonical form through *standardization* and *simplification* rules [35], that lead to a more desirable initial query plan.
- Algebraic rewritings: the properties of the logical algebra are used to transform a logical algebra expression into one that is equivalent to the original expression, but has more desirable characteristics than the original.
- Reductions: the logical algebra operator is mapped into one or more physical operators that are responsible for computing the result of the query.

The simplification transformations performed over the calculus are typical examples of logical rewritings where no cost-based decisions are involved during the transformations. This approach is employed in the OQL optimizer described in [21]. There, OQL is translated to the monoid calculus [22], where the query is normalized and later mapped into an object algebra.

Given that languages like DOQL [46] and F-Logic [36] support nested queries, logical rewritings, as well as being useful in recursive query optimization, are also necessary for transforming the query into a canonical form. Due to the logic-based nature of deductive rule languages, no other intermediate representation is necessary for carrying out simplification and standardization, and the rewritings can be performed over the deductive language structure. After transformations are carried out at the conceptual query level, the canonical query representation is mapped into an initial algebraic expression that is the starting point of the algebraic optimization stage.

To process DOQL queries, logical rewriting and algebraic optimizations are employed. The transformations done during logical rewriting aim to perform some optimizations and reduce DOQL queries into a canonical form that can be translated into an object algebra for further optimization and for evaluation.

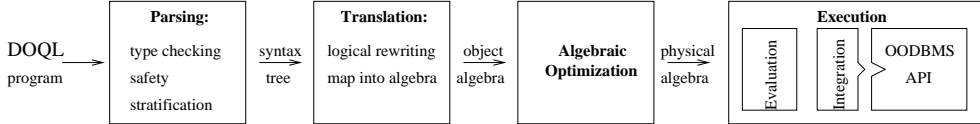


Fig. 6. DOQL query processing framework

The DOQL query processing framework shown in figure 6 provides a clear separation between logical rewriting and algebraic optimization, and is conceived to enable the combination of deductive optimization techniques, algebraic optimization, and different strategies for bottom-up evaluation (naive and semi-naive). As well as enabling complementary techniques to be combined, the separation provides a workbench for experimenting with different combinations of techniques, and reduces the complexity of implementing a solution to the optimization problem.

The layers, which have been designed as far as possible as replaceable components, communicate data to each other using a collection of abstract data types. For instance, DOQL syntax trees are the unit of exchange between the parser and the translator, the logical query plan tree represents the unit of exchange between the translator and the optimizer, the physical plan tree defines the unit of exchange between the optimizer and the evaluator, and a system-independent tuple-based data type defines the unit of exchange between the evaluator and the integrator component, the latter being responsible for the interface between the DOQL evaluator and the host OODBMS.

The reuse of DOQL as a component is only limited by the integrator layer and the type checker, which are host OODBMS dependent. The type checker is likely to become host OODBMS independent as soon as the ODMG standard specification for metatypes is implemented by the vendors. The integrator will remain dependent since the ODMG API is not powerful enough to provide the API functionality described in section 3, which is necessary for implementing DOQL. This is the only layer that has to be rewritten when bundling DOQL with a different OODBMS.

The object algebra underlying DOQL is based on the algebras presented in [19–21]. The semantics is close to the semantics of algebras that can underpin OQL like the algebras presented in [19,22], encompassing operators that deal with multiple collection types thereby avoiding the need to divide the algebraic operators into different sorts according to each collection type supported.

Object invention is not allowed, and all operators have an object preserving semantics, avoiding the problem of infinite chains of deduction due to object invention [59]. Fixpoint related operators are supported as bulk operators in the algebra to explore the results presented in [8,9], which have shown that a fixed point algebra has the same expressive power as stratified deduction. In DOQL, iterated fixpoint computation over the algebraic counterpart of the stratified rule set is applied to compute the fixpoint.

Each object manipulated by the algebra has an abstract representation as a non-first normal form tuple-based structure:

```
([oid], (value_assembly))
```

The tuple-based structure is composed of an optional object identity field *oid*, for tuples representing objects, and a *value_assembly* field which is also a tuple-based structure carrying values for attributes and relationships of the object. The set of ODMG types is the set of types allowed for values that can appear as members of a *value_assembly*. Fields containing values of type OID are also allowed in the *value_assembly* portion, but they represent relationships or object-valued attributes of the entity that is represented as a tuple-based structure. The presence of the optional *oid* field in the structure serves to identify when the tuple represents an object or not. As an example, a course object in the extent of a class *course* is represented within the algebra as follows:

```
(#c1, ('Database Systems', 'CS2311', #d1, {#c2, #c3}))
```

#c1 is the value for the object identity field *oid* of the object which belongs to class *course*. The object has its state represented as a value assembly of the values for the attributes *name*, *code*, *dept*, and *prerequisites*, which is a collection of courses.

Null value semantics is akin to the semantics adopted in ODMG, where there is a NULL type for every distinct literal type, and as a special value, a *nil*³

³ The *nil* is the value representation of a NULL type in the ODMG standard. [15], page 34.

is different from every other value, including another *nil*. If a bulk data type property of an object doesn't have an available value, an empty instance of the bulk data type defines the value.

As well as being able to manipulate objects as described above, which are scanned from the underlying object database class extent using a `get` operation of the algebra, other bulk operators are supported, which return bags or sets of tuple-based structures, following the approach adopted in industrial OQL implementations [18]. This approach provides a uniform representation for the output results of all bulk operations and helps to enforce the closure property of the algebra. The bulk operators can have one or more collections as input and can have boolean predicates composed using the primitive operators supported in the DOQL object algebra.

4.3.1 Primitive Operators

Primitive operators are built-in operators in arithmetic and comparison expressions appearing as arguments of bulk operators. Table 1 illustrates the primitive operators and their associated semantics:

Arguments	Operator	Semantics	Return type
a:T, $T \in \{\textit{object type}, \textit{literal type}\}$, b:T, $T \in \{\textit{object type}, \textit{literal type}\}$	<code>eq(a, b)</code>	$a = b$	bool
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>gt(a, b)</code>	$a > b$	bool
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>lt(a, b)</code>	$a < b$	bool
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>leq(a, b)</code>	$a \leq b$	bool
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>geq(a, b)</code>	$a \geq b$	bool
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>sum(a, b)</code>	$a + b$	<i>numeric type</i>
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>div(a, b)</code>	$a \div b$	<i>numeric type</i>
a: <i>numeric type</i> , b: <i>numeric type</i>	<code>mul(a, b)</code>	$a \times b$	<i>numeric type</i>
a:T, $T \in \{\textit{object type}, \textit{literal type}\}$, b: <i>set</i> < T >, <i>bag</i> < T >, <i>list</i> < T >	<code>in(a, b)</code>	$a \in b$	bool

Table 1
Built-in functions

4.3.2 Bulk Operators

The bulk operators of the algebra enable access to extents and named collections in an ODMG schema, and to tuples in the virtual views defined by the IDB predicates. The operators are governed by typing properties similar to those described in [19] so as to support algebraic equivalences. They can have boolean predicates as parameters, with matching between a variable and a stored value being possible when there is an equality predicate on an unbound variable as part of the parameter denoting a boolean predicate. The following bulk operators are supported:

join($A, B, pred$) the join operation joins two collections A, B using the predicate $pred$. If no predicate $pred$ is given, the cartesian product of the tuples of the two collections is computed, otherwise, the operation returns the bag of tuples that satisfies the condition $pred$. The output tuples are obtained by concatenating values from input tuples.

select($A, pred$) returns the bag of tuples from collection A for which $pred$ holds.

get($type, pred$) returns the bag of tuples from the extent of type $type$ for which $pred$ holds.

apply($A, f(a)$) applies the function f to input tuples a from collection A , returning the bag of modified tuples $f(a)$.

unnest($E, path, v, pred$) unnests the nested collection E into a bag of tuples by retrieving the $path$ component into variable v . The bag of flattened tuples that satisfy $pred$ is returned.

nest($E, group_var, t, v, pred$) creates a nested collection from E , by grouping each element in E according to the value of the variable appearing in $group_var$. The groups formed relative to each value of $group_var$ are accessible through v . The variable in $group_var$ must have been previously defined in E . The bag of nested tuples t for which $pred$ holds is returned.

fixpoint($idbPred, algExp$) computes the fixpoint of the algebraic expression $algExp$ that defines the predicate $idbPred$. The operation returns, at each stage of the iteration, the set of tuples that belong to the extent of the *virtual view* defined by the IDB predicate $idbPred$ that $algExp$ represents.

union(C_1, C_2) computes the set union of two union compatible collections C_1 and C_2 , returning a bag of tuples.

set-diff(C_1, C_2) computes the set difference $C_1 - C_2$ between two union compatible collections C_1 and C_2 , returning a set of tuples.

merge(C_1, C_2, \dots, C_n) merges the elements of n union compatible collections, returning a bag of tuples.

iterate(A_1, A_2, \dots, A_n) computes the naive or semi-naive evaluation⁴ of the list of algebraic expressions A_1, A_2, \dots, A_n . Each algebraic expression A_j represents the query plan corresponding to the translation of a subset of the rule set that has the same IDB predicate as rule head (partition of the rule set). The operation returns the bag of tuples that is part of the virtual view of the *query* IDB predicate, when it reaches a fixpoint.

feed(*idbPred*, *f*) the feed operator is associated with an occurrence of an IDB predicate in a rule body, and is responsible for retrieving the tuples that belong to the extent of the virtual view defined by the IDB predicate *idbPred*, and applying a variable renaming function *f* to the type of the input tuples. This operation returns the set of tuples retrieved from the virtual view extent.

```
print(
  iterate(
    fixpoint(
      merge(
        apply(
          get(train, source.name=S, dest.name=P),
          struct(S,P)
        ),
        apply(
          join(
            feed(reaches, S,L),
            get(train, source.name=L, dest.name=P)
          ),
          struct(S,P)
        )
      )
    ),
    apply(
      select(
        feed(reaches, S,P),
        S=aberdeen
      ),
      struct(P)
    )
  )
)
```

Fig. 7. Query plan for Q8

The plan depicted in figure 7 represents the algebraic counterpart of query Q8 in section 2. This algebraic expression is obtained after the mapping stage of the translation described in section 4.4, and serves as input to the optimizer. The algebraic counterparts of the primitive algebraic operators that serve as input to the bulk operators were not shown to simplify the illustration.

⁴ The choice of the evaluation method is based on the linearity property of a DOQL program. The property is computed using the dependency graph described in section 2.4.

In a DOQL query processing tree, `get` and `feed` represent the leaf nodes, where `get` retrieves its tuples from the stored database, and `feed` retrieves its tuples from the virtual view extent defined by a `fixpoint` node. The `fixpoint` operator serves as a control node that monitors the computation of the fixpoint and also controls the update to the virtual view extent that corresponds to the IDB predicate. If the predicate is not recursive, the `fixpoint` operator does not need to update the virtual view extent, and works as a control mechanism for the unfolding of the virtual view defined by the predicate, allowing its use in other queries.

The renaming function of the `feed` is responsible for renaming the variables that come from the corresponding `fixpoint`. For example, in the plan in figure 7, the tuples that are output from the `fixpoint` of the `reaches` predicate have (S,P) as the textual definition of variables corresponding to the first and second values in the tuple. After the renaming, the tuples will have (S,L) as the textual definitions of variables associated to the values. This transformation does not involve changes in variable values at the execution stage.

4.4 DOQL Translation

The translation of DOQL to the object algebra comprises the following steps:

- (1) *Logical Rewriting*: in this step, a normalization process transforms a rule into a canonical form (DOQLc), where some types of nested queries are unnested, DOQL formulas are rewritten as molecular expressions, and local optimizations are performed over each rule. The rewriting step is illustrated in section 4.4.1, and is described in detail in [53].
- (2) *Mapping*: in this step, the rule set composed of canonical rules is translated into a query plan with DOQL logical algebra operators as nodes. The object algebra is described in section 4.3, and the global mapping of the rule set is described in section 4.4.2.

During translation, a DOQL program is pre-processed to *rectify* [62] the rule set. Rectification is done by renaming multiple occurrences of the same variable in each rule head and adding equality conditions in the rule body; removing constants from rule heads and placing them as equality conditions in the rule body, and renaming all rules defining the same predicate so that the list of variables in each rule head has the same signature⁵. The rectification technique is described in [62].

Mapping into an object algebra provides an operational semantics to DOQL and enables the combination of deductive optimization techniques based on

⁵ Same name and order of appearance.

logical rewriting, with optimization and evaluation based on traditional algebraic query processing techniques.

4.4.1 Logical Rewriting: Normalization and Local Optimizations

In deductive object-oriented databases (DOODs), objects, classes and class members (structure and behaviour) play a central role in the organization of information, contrasting with deductive relational databases where tuples, relations and attributes define the main abstractions.

Query processing technology for deductive databases is mainly targeted at the relational data model, which leaves to DOOD languages the options of conceiving new query processing techniques or of rewriting the queries into a form that adapts to an existing technology. The first approach is still an open research problem, while the latter approach has been employed in several systems and is also adopted by DOQL.

The rewriting process in DOQL is based on grouping the information directly related to collections as molecules. Molecules are used to group formulas regarding properties and operations with the objects that belong to collections. The molecular form can exploit classical query evaluation methods which are centered on the evaluation of scans over the collections represented by each molecule. The conditions grouped inside the molecules can be used to attach filtering predicates to the scans. The following example shows a DOQL rule containing two molecules on the right hand side of the rule:

```
query(TNO) :- train[tno=TNO,route=>R], R[place.name="aberdeen"].
```

In the rule above, two molecules are used to group information related to the objects in the extent of class `train` and to the collection of objects that are part of the nested collection `R`.

Although it is possible to encode certain deductive object-oriented database languages as Datalog rules, as a preliminary step of rule evaluation, this process can generate complex recursive patterns and drastically increase the number of predicates [67]. In DOQL, instead of completely breaking the object information structure as is done when translating DOOD languages to Datalog, the notion of collection expressions organized as DOQL molecules is retained during query evaluation.

The rewriting process proceeds by applying the normalization rules *Associate* (\odot), *Compose* (\otimes), and *PullOut* (\propto) introduced in [67] to obtain a canonical form for DOQL. The symbol (\rightsquigarrow) is used to denote a formula operator in DOQL (i.e. \rightarrow or \Rightarrow).

The *Associate* operator binds the collection membership information between objects and collections:

$$p(X), X[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n] \xrightarrow{\odot} p(X)[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n]$$

The following example illustrates the application of the *Associate* rewrite to the DOQL rule Q5 described in section 2:

(Q5)

```
query(TNO) :- train(T), T[route=>R[place.name="aberdeen"]], T.tno=TNO.
```

$$\xrightarrow{\odot}$$

(Q5-A)

```
query(TNO) :- train(T)[route=>R[place.name="aberdeen"]], T.tno=TNO.
```

The *Compose* operator groups all formulas referring to the properties and operations of an object into the same molecule:

$$p[l_1 \rightsquigarrow p_1], \dots, p[l_n \rightsquigarrow p_n] \xrightarrow{\otimes} p[l_1 \rightsquigarrow p_1, \dots, l_n \rightsquigarrow p_n]$$

The following example illustrates the application of the *Compose* rewrite to Q5-A:

(Q5-A)

```
query(TNO) :- train(T)[route=>R[place.name="aberdeen"]], T.tno=TNO.
```

$$\xrightarrow{\otimes}$$

(Q5-B)

```
query(TNO) :- train(T)[route=>R[place.name="aberdeen"], T.tno=TNO].
```

In case of a formula describing relational properties such as equalities, inequalities, greater than, etc., between more than one object (variables of two different molecules appearing in the left and right hand sides of the relational operator), the *Compose* is not applied, as in the formula:

```
query(TNO) :- train(T)[tno=TNO], station(S), T.type = S.type.
```

The formula `T.type = S.type` relates to two objects appearing in different molecules. The *Compose* rewrite is not applied in this case. The *Compose* operator need not be applied because in the mapping stage, formulas involving relational properties between objects of different molecules will appear as predicates of joins between the scans of the two extents corresponding to the molecules, while formulas that involve an object of a single molecule will be translated as filters of scan operations over a single extent. In the latter case,

the *Compose* helps to group the predicates inside the molecule, simplifying the translation to the object algebra.

The *PullOut* operation flattens a DOQL rule containing a nested collection expression.

$$\begin{array}{c}
 p[l_1 \rightsquigarrow p_1, \dots, l_i \rightsquigarrow p_i[l \Rightarrow v], \dots, l_n \rightsquigarrow p_n] \\
 \xrightarrow{\alpha} \\
 p[l_1 \rightsquigarrow p_1, \dots, l_i \rightsquigarrow p_i, \dots, l_n \rightsquigarrow p_n], p_i[l \Rightarrow v]
 \end{array}$$

The following example illustrates the application of the *PullOut* rewrite to the DOQL expression Q5-B:

(Q5-B)
`query(TN0) :- train(T) [route=>R [place.name="aberdeen"], T.tno=TN0].`

$$\xrightarrow{\alpha}$$

(Q5-C)
`query(TN0) :- train(T) [route=>R, T.tno=TN0], R [place.name="aberdeen"].`

The overall effect of the normalization of the nested rule is to obtain a canonical DOQL program which can be mapped into the DOQL object algebra.

A *canonical* DOQL program, denoted by $DOQL_c$ is a DOQL program that has been normalized by the rewriting process.

DOQL molecular expressions have a direct translation counterpart into the `get` operation of the DOQL object algebra, whose parameters include the comparison predicates that were grouped inside each molecule. This corresponds to the local optimization heuristics of *pushing selections before joins*. It is local in the sense that it is done in the context of each rule body individually.

Global optimizations consider the rule set as a whole. For instance, a comparison predicate given in the query may be a potential source of optimization that restricts the number of tuples generated by a `get` generated from the translation of another rule in the rule set. Pushing this condition downwards in the algebraic counterpart of the rule set may involve not only optimizations that push selections before joins, but also optimizations that push selections into fixpoint operators. This will be discussed later in this paper.

4.4.2 Mapping DOQL to the Object Algebra

The mapping step in the DOQL query processing framework is responsible for transforming the canonical rule set into an algebraic query plan. By mapping into an object algebra, the handling of many DOQL queries which involve nested collections and path expressions can benefit from research on algebraic approaches to OO query optimization and evaluation. Figure 8 presents the algorithm **MapRuleSet** that maps a DOQL canonical rule set into the logical algebra⁶.

```
INPUT: DOQL rule set   OUTPUT: tree of algebraic operators (PlanTree)

Pre-Processing Stage:

1. Build partitions // Each partition contains a subset of the rule set
   // with the same IDB predicate as rule head in each rule.
   // One of the partitions denotes the query special predicate.

2. Stratify the rule set and compute the strata level of each partition

3. Increment by 1 the stratum of the partition relative to the query predicate
   // forces the partition containing the special query predicate to
   // be the one with highest strata, and consequently,
   // the last partition translated in the Translation Stage

Translation Stage:

4. PlanTree query_plan ← nil
   List <PlanTree> subplans ← ∅

5. for each partition Pi defining an IDB predicate, starting from the lowest strata
   5.1. List <PlanTree> partial_plans ← ∅
       PlanTree partition_plan ← nil
   5.2. for each rule Rj in partition Pi
       5.2.1. Translate rule Rj into algebraic expression Aj
              PlanTree Aj = RuleToAlgebra(Rj)
       5.2.2. partial_plans ← partial_plans.push_back(Aj)
   5.3. partition_plan ← merge(partial_plans)
   5.4. if (Pi != query predicate)
       partition_plan ← fixpoint(partition_plan)
   5.5. subplans ← subplans.append(partition_plan)

6. query_plan ← print(iterate(subplans))

7. return(query_plan)
```

Fig. 8. **MapRuleSet** algorithm

The **MapRuleSet** algorithm works by building partitions containing rules with the same intensional database predicate, stratifying the rule set, and mapping each partition into an algebraic subplan that will be iterated until a fixpoint is reached. To map each rule of a partition, the algorithm **RuleToAlgebra** presented in figure 9 is called. Both algorithms use algebraic construction functions that have the same names as logical algebra operators to build the global query plan. Each construction function receives one, two or a collection of plan trees as input, and returns a plan tree with the logical

⁶ List container, and operations such as **push_back**, for insertions at the end of a list, and **append** for appending lists are used in the pseudocode.

algebra operator corresponding to the function as the root of the output plan. The construction operators work by building larger plan trees from smaller plan trees representing partial plans, until the plan tree corresponding to the rule set query plan is obtained.

The resulting plan tree is the input to the algebraic query optimizer which will perform algebraic manipulations based on the equivalence properties of the algebra, and choose the physical operators that implement each operator in the logical algebra. There is one physical operator for each logical operator in the algebra, except for the `join` and `get` operators, for which there is more than one physical operator.

```

INPUT: rule H :- L1,...,Ln   OUTPUT: tree of algebraic operators (PlanTree)

1. List <PlanTree> join_list ← ∅
   List <PlanTree> select_list ← ∅
   PlanTree partial_plan ← nil
   PlanTree rule_plan ← nil

2. for each literal Li in the rule body
  2.1 if (Li is a molecule)
    2.1.1 partial_plan ← get(get_type(Li),get_preds(Li))
    2.1.2 for each nested collection (attr_i => Var_i) in get_nested_cols(Li)
      2.1.2.1 partial_plan ← unnest(partial_plan,attr_i,Var_i)
      2.1.2.2 join_list ← join_list.push_back(partial_plan)
  2.2 if (Li is a virtual view)
    2.2.1 partial_plan ← feed(get_IDB_name(Li),get_vars(Li))
    2.2.2 join_list ← join_list.push_back(partial_plan)
  2.3 if (Li is a predicate: =, !=, <=, >=, <, >, in)
    2.3.1 partial_plan ← select(Li)
    2.3.2 select_list ← select_list.push_back(partial_plan)

3. if (join_list.size() > 1)
  3.1 partial_plan ← join(join_list[0],join_list[1])
  3.2 int current = 2
  3.3 for ( int iter = join_list.size() - 1 ; iter > 1; iter--)
    3.3.1 partial_plan ← join(partial_plan,join_list[current])
    3.3.2 current++

4. for ( int iter = 0; iter < select_list.size(); iter++)
  4.1 partial_plan ← select(partial_plan,select_list[iter])

5. rule_plan ← apply(partial_plan,get_vars(H))

6. return(rule_plan)

```

Fig. 9. **RuleToAlgebra** algorithm

The **RuleToAlgebra** algorithm described in figure 9 performs a mapping of the rule body into a logical algebra expression composed of cartesian products of flattened collections, and chains of selects. The rule head is translated into an `apply` operator that denotes the projection of the rule head variables. Nested collections are flattened using the `unnest` operator and cartesian products are represented by sequences of joins that work as cartesian products when no join condition is given. The sequence of joins generated by the mapping algorithm gives rise to *left-deep* join trees.

4.5 Global Optimizations in DOQL: When Is Magic Relevant?

Given that the DOQL query processing framework supports deductive optimization techniques based on logical rewriting and algebraic optimizations, it is important to identify which combinations of logical rewriting and algebraic optimization techniques have the greatest potential to reduce query response times.

Logical rewriting is based on the blind application of heuristic rules and proceeds by applying transformations to the initial query supposing that they do not incur any loss of optimality. No selection among alternatives is involved because the heuristics are always considered to be worth applying [39]. The *magic sets* family of rewritings [4,63,43] are often considered to represent among the most effective techniques for deductive query optimization. Algebraic optimization, in contrast to logical rewriting, can often rely on a cost model to decide when a transformation is worth applying.

The decision regarding which transformations to use, and whether an optimization should be done at the logical rewriting level or at the algebraic level, depends mainly on what language constructs are used in the query (path expressions, comparison operators, equality, method calls, etc.), the availability of access support structures like indexes, and on the optimization scope (local or global optimization). To illustrate the problem of local and global query optimization in DOQL, we provide two alternative ways of implementing query Q8 described in section 2:

Q8: *where can you reach from Aberdeen? Retrieve the names of the destinations.*

(Q8-A)

```
reaches(S,P) :- train(Tr) [source.name=S,dest.name=P],
               Tr.source.name = "aberdeen".
reaches(S,P) :- reaches(S,L), train[source.name=L,dest.name=P].
query(P)      :- reaches(S,P).
```

(Q8-B)

```
reaches(S,P) :- train[source.name=S,dest.name=P].
reaches(S,P) :- reaches(S,L), train[source.name=L,dest.name=P].
query(P)      :- reaches(S,P), S = "aberdeen".
```

■

In the first implementation, the condition that expresses the requirement that the train departs from Aberdeen is placed in the same rule as the molecular

expression describing the result set obtained from the extent of train. In the second, it is placed in the rule containing the query predicate, far from the extent that needs to be filtered. Pushing the selection inside the molecule that will map into a `get` in the first implementation is an example of a local optimization that can be performed in the context of a single rule, while in the second example, pushing the selection from the query rule into the reaches rule is an instance of the global optimization problem.

Deductive query optimization based on the magic sets method typically proceeds by rewriting using a simple heuristic: restricting the computation in recursive predicates to relevant facts. This is achieved by pushing conditions through recursion using adornments [43]. In OODBs, however, selections involving method calls or path traversals on complex objects may be expensive to compute. Therefore, they need to be assessed in the presence of a cost model [39].

From the above, it seems safe to apply magic sets whenever a condition does not involve a path expression or a method call. The optimization of path expressions and method calls can be delayed until the algebraic optimization stage, where path expressions can be rewritten as joins and physical properties can be used to identify the best choice of physical operators that will be used to implement the algebraic plan.

Global Algebraic Optimizations Involving Fixpoint

In DOQL, queries involving recursion are optimized by algebraic transformations in the global query plan. Algebraic rewriting rules that enable the moving of selections into fixpoint operators were originally proposed by Aho and Ullman in [3]. These ideas were later generalized by Agrawal and Devanbu in [2] to extend the range of selection conditions that can be moved inside the fixpoint operator.

In DOQL, the rules for moving selections proposed by Aho and Ullman are implemented in the algebraic optimization of the global query plan to move a `select` operator inside a `fixpoint`. This transformation can reduce a general transitive closure computation into a less complex computation using the information in the predicates that are part of the `select` operator.

The rules for moving the selections check if a selection can be distributed through the subexpressions of the `fixpoint` node. Other equivalence rules for moving selection conditions before other operators in the object algebra are also used in the process.

Table 2 describes a subset of the valid equivalence rules between bulk operators of the DOQL object algebra that enable moving selections into the scope of

a fixpoint operator. In the rules, C, C_1, C_2, \dots, C_n denote collections of tuples returned by the operator, and $pred, pred_1, \dots, pred_n, p_1, \dots, p_n$ denote predicates formed using object, tuple and primitive operators supported in the algebra.

Rule	Algebraic Property
commutativity	$\text{select}(\text{select}(C, pred_1), pred_2) \equiv \text{select}(\text{select}(C, pred_2), pred_1)$
cascading	$\text{select}(C, pred = p_1 \wedge p_2 \dots \wedge p_n) \equiv \text{select}(\text{select}(\dots \text{select}(C, p_1), \dots), p_n)$
distributivity	$\text{select}(\text{merge}(C_1, \dots, C_n), pred) \equiv \text{merge}(\text{select}(C_1, pred), \dots, \text{select}(C_n, pred))$
distributivity	$\text{select}(\text{union}(C_1, C_2), pred) \equiv \text{union}(\text{select}(C_1, pred), \text{select}(C_2, pred))$
incorporation	$\text{select}(\text{get}(C, pred_1), pred_2) \equiv \text{get}(C, pred = pred_1 \wedge pred_2)$
incorporation	$\text{select}(\text{unnest}(C, path, v, pred_1), pred_2) \equiv \text{unnest}(C, path, v, pred = pred_1 \wedge pred_2)$ if $pred_2$ involves the property $path$.
distributivity	$\text{select}(\text{unnest}(C, path, v, pred_1), pred_2) \equiv \text{unnest}(\text{select}(C, pred_2), path, pred_1)$ if $pred_2$ does not involve the property $path$.
distributivity	$\text{select}(\text{join}(C_1, C_2, pred_1), pred_2) \equiv \text{join}(\text{select}(C_1, pred_2), C_2, pred_1)$ if $pred_2$ involves only properties in C_1 .
distributivity	$\text{select}(\text{apply}(C, f(t)), pred) \equiv \text{apply}(\text{select}(C, pred), f(t))$

Table 2

Equivalence rules involving the `select` operator

Figure 10 illustrates the pushdown of a `select` for the logical query plan of query Q8-B. The `select` is distributed across the `merge` and placed close to the `get`, limiting the flow of tuples and reducing the complexity of the computation. The pushdown step is controlled by the variables involved in the arguments of the condition that is part of the `select` operator.

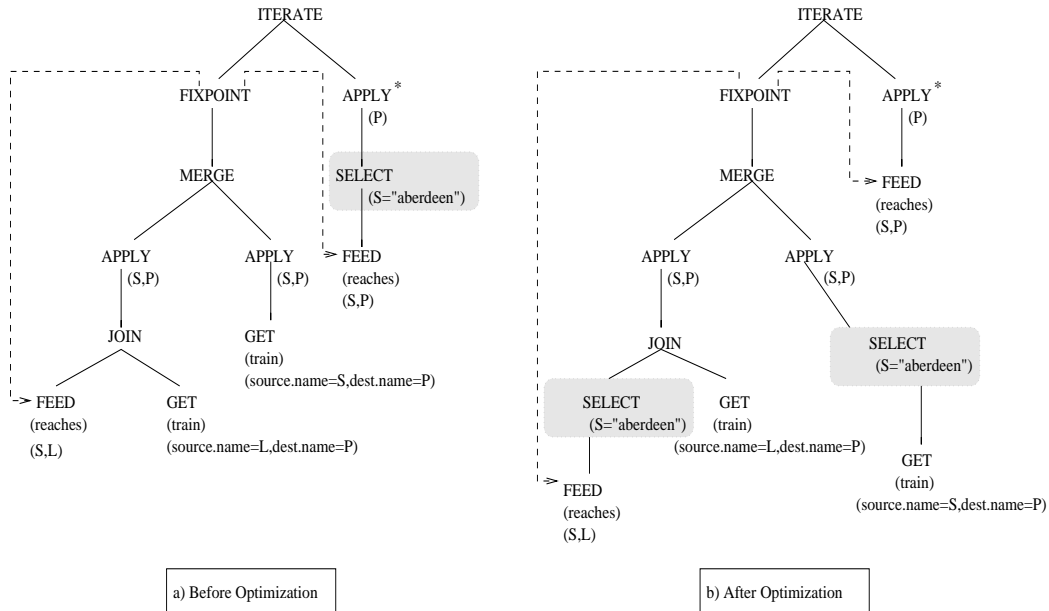


Fig. 10. Algebraic recursive query optimization

5 Query Execution

The DOQL execution engine is responsible for evaluating the physical algebra⁷ expression that represents the execution plan for the query. The physical operators represent the interface provided by the execution engine to the optimizer. The order of execution for an operator tree is determined by a tree traversal algorithm that is initiated by an activation of the root operator of the tree.

The main components of the execution engine are the *integrator* component that is responsible for interfacing with the underlying storage system and the *evaluator* component, which implements the physical algebra that underpins the DOQL query processing framework. The execution engine in the context of the DOQL query processing framework is shown in figure 6.

5.1 The Integrator Component

To compute the answer to a query, a DOQL program is parsed, type checked, optimized and control is immediately transferred to the evaluation of the physical algebra plan over the stored data. Most of the query processing framework and the corresponding software layers that implement it are independent of the underlying storage mechanism used, with the exception of the integrator component, which is responsible for interfacing with the storage mechanism.

Distinct object database systems tend to support different storage data types with different APIs, and it is likely that this will continue, since the ODMG standard only covers compile-time portability for applications, and does not address issues such as standard formats for objects in the storage manager, or standards for the low level API that is supported by each individual DBMS. The latter issues are very important for the designer of a query system, since lower level access to the storage system is often required.

The task of providing a host DBMS independent view of the underlying storage system to the upper layers is performed by the integrator layer using a tuple-based approach. This converts stored objects into the tuple-based structures of the DOQL object algebra, which have ODMG defined data types as members. The integrator layer also deals with system-specific notions such as physical OIDs, storage data types, conversion between system independent OIDs into physical OIDs, and provides an abstract view of the store to the upper layers.

⁷ Algebra formed by the query processing algorithms supported in a query execution engine [31].

DOQL is currently interfaced with the Poet 5.0 C++ API. The Poet API provides functions that allow the retrieval and interpretation of the stored data during the course of query execution. The API functions are provided as a call back handler class `PtObjectWalker` that walks through an object description calling handlers for each data member of the object in turn.

The handlers enable the offsets and lengths of the component data fields to be computed, providing access to their values at runtime. To interface DOQL to Poet, it was necessary to implement a class `MyWalker` derived from `PtObjectWalker` which redefines the default behavior of the handler functions in order to build tuples that represent each object within the DOQL query processing framework. The tuples have ODMG data types as members (built by the redefined handlers) and are opaque with regard to the underlying OODBMS. This interface between DOQL and Poet is encapsulated in the implementation of the scan iterators that are in the leafs of an operator tree. This is described in detail in the following section.

5.2 *The Evaluator Component*

The DOQL query execution engine comprises a set of algorithms that are responsible for identifying, retrieving, assembling and iterating over stored objects to compute the result of query expressions. The algorithms are organized as physical algebra operators that iterate over members of input streams. The input streams of tuples can originate from stored collections or intermediate collections resulting from the execution of a previous physical algebra operator.

The physical algebra operators are executed in a dataflow model, where each operator consumes zero or more inputs from previous operators and produces zero or more outputs. A query plan is a tree of physical algebra operators where operators situated at a higher position in a tree hierarchy activate operators situated at lower levels. The leaves of a tree are scan operations over the objects stored in the underlying storage mechanism or feed operations that request data from virtual views.

The main goals in the design of a physical algebra and a query execution engine are to achieve a balance between the issues of ease of implementation, extensibility, portability, and performance [32].

The **ease of implementation** goal is achieved by adopting the iterator model [50] as the main control abstraction which guides the implementation of the query execution engine. The iterator is a common design pattern [27] in query processing software construction and can be used for implementing physical algebra operators that produce output streams while hiding the details of how

the items are obtained.

An iterator saves its state between calls, allowing each successive call to resume execution where the previous call left off. The iterator abstraction hides the complex control connections (synchronization and scheduling), performs bookkeeping of states, and simplifies data transfer between operators during query processing.

In the iterator model, query execution plans are activated at the root of the plan tree, progressing towards the leaves. Invoking the root operator of a subtree automatically activates the child operators, starting a chain of operator activations. Since each iterator schedules its inputs according to its needs, all iterators in a complex plan are activated and driven at the appropriate pace for their consumers [32,31].

In the DOQL query execution engine, every physical operator (`seq_scan`, `index_scan`, `nestedloop`, `unnest_iter`, `print_iter`, `nest_iter`, `union_iter`, `hash_join`, `fixpoint_iter`, `iterate_iter`) is implemented as a subclass of the abstract class `phys_algebra_op`, shown in figure 11.

```
class phys_algebra_op
{
    public:
    virtual void open() = 0;
    virtual void close() = 0;
    virtual Tuple_Object next() = 0;
    virtual ~phys_algebra_op() {};
};

typedef phys_algebra_op* Physop; // General type alias for physical operators
```

Fig. 11. Abstract class defining the iterator interface

The class `phys_algebra_op` defines the iterator protocols that are implemented by each physical algebra operator. As well as the iterator protocols, the physical algebra operators also have attributes defining state information that needs to be kept between successive activations of the operator, and functionality relating to garbage collection of the items that are dynamically created by the operator.

Data exchange between the consumer and producer iterators happens through a call to a `next()` method. For example, the nested loop iterator can call `left->next()` to activate the producer and receive a tuple object from the left input stream. The data items exchanged between iterators are allocated in temporary space (the heap), and are kept track of by the producer of the item when garbage collection is performed.

This design also addresses the **extensibility** goal, as new physical algebra algorithms can be added to the system as subclasses of the `phys_algebra_op` class, without affecting other components of the query execution engine.

In figure 12, the declaration part of the nested loop iterator is given. A tuple is created by the nested loop when the combined tuple resulting from the cartesian product of the two input streams (represented by the `left` and `right` members) has its predicate evaluated (represented by the `l_pred` member) to true. This concatenated tuple is returned by the nested loop to the iterator that activated the nested loop, as a result of calls to the `next()` function. The `collector` member is responsible for the run-time control of all tuples that were created by the nested loop, performing their garbage collection when they are no longer needed.

Other members of the iterator are responsible for keeping in memory the smallest collection that is being joined and for keeping track of each tuple from the input collections that are being joined in each iteration.

```
class nestedloop : public phys_algebra_op
{
    list<genform*>* l_pred;
    Physop left;
    Physop right;
    collection_object* cleft;
    collection_object* cright;
    Tuple_Object tobj1;
    Tuple_Object tobj2;
    collection_object* collector;
public:
    nestedloop(Physop leftop, Physop rightop, list<genform*>* l_preds);
    ~nestedloop();
    void open();
    void close();
    Tuple_Object next();
};
```

Fig. 12. Nested loop iterator interface

All physical algebra operators have a uniform return type (`Tuple_Object`) that is returned when a call to a `next()` function is performed. The `Tuple_Object` type defines the non-first normal form tuple-based data structure that represents each element of the data flow that is communicated between iterators. An element of the data flow can be:

- (1) A stored object retrieved from an extent, which is converted into a `Tuple_Object` type.
- (2) A `Tuple_Object` type resulting from the concatenation of two other `Tuple_Object` types obtained as output of a physical operator that implements a join (e.g., `nestedloop`).
- (3) A `Tuple_Object` type resulting from the nesting or unnesting of a field of a `Tuple_Object` type.

To support tuple-based processing over an object store, the scan operators need to be able to access the object store, assemble an object as a tuple, convert the OIDs into a system independent OID data type, and have the corresponding object materialized in main memory when necessary.

This tuple-based structure defined by the type `Tuple_Object` is obtained by assembling the simple attributes (atomic data types) of the object, together with a collection reference type for each nested collection that is a member of the object, and an OID data type for each object that is a member of the object that is being converted into a tuple. In other words, the data members that are usually investigated in a shallow equality are made part of the tuple structure.

Figure 13 illustrates the conversion of a stored object of type *employee* into a `Tuple_Object` instance. The atomic data values for the members *name* and *telno* are placed in the tuple, together with a logical OID data value for the *department* member, and a collection reference value for the *dependents* member. The object identity of the object being converted is also placed in the first position of the tuple as a logical OID data value. The behavioural part of the object can be accessed at any time using the typing information associated with the tuple.

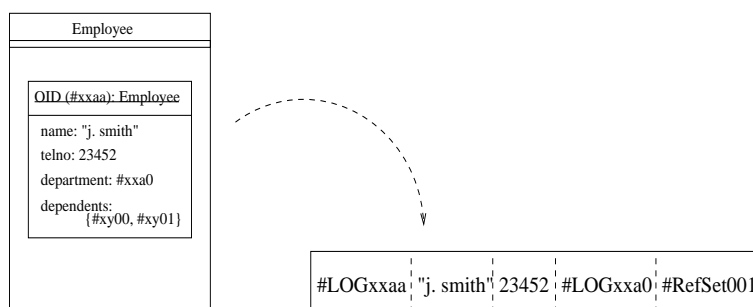


Fig. 13. Conversion into a tuple object

The scan operators are responsible for interfacing with the underlying object storage and assembling the `Tuple_Object` types that will feed the data flow execution. This interaction with the underlying storage mechanism is completely encapsulated inside the scan operations, and once the `Tuple_Object` type is assembled, the other iterators deal with the abstract tuple.

The code in figure 14 describes the declaration part of a scan operation with the underlying object storage. The `MyWalker` class that is a member of the scan encapsulates the code that interacts with the underlying object storage. It manipulates the stored extents, retrieving objects from the extent, and assembling the object as a `Tuple_Object` type. If the underlying storage mechanism is replaced, the `MyWalker` class is the portion that needs to be adapted to the new setting.

Further interaction with the underlying storage mechanism is necessary when a logical OID needs to be converted into a physical identity and its corresponding object loaded in memory, or when a reference to a collection needs to be processed. Again, these interactions are abstracted as functions that support the physical algebra and can be easily adapted for different storage

```

class seq_scan : public phys_algebra_op
{
    MyWalker pw;
    string cl_name;
    int iter;
    list<genform*>* l_pr;
    collection_object* collector;
public:
    seq_scan(string class_name, list<genform*>* l_preds);
    ~seq_scan();
    void open();
    void close();
    Tuple_Object next();
};

```

Fig. 14. Sequential scan iterator interface

mechanisms.

Although the scan family of physical operators is system-specific, the design of the query execution engine and its corresponding physical algebra is conceived in a way that eases its porting to other storage systems. The encapsulation of the interface between the physical algebra and the underlying storage mechanism using the `MyWalker` class, and the use of ODMG data types as members of the `Tuple_Object` type is the key to achieving the **portability** goal. The `MyWalker` class works as a plug-in component that will connect the physical algebra layer to the underlying storage system, providing a system independent tuple-based view of the storage system to the upper layers.

The **performance** goal is addressed by adopting a data-shipping query execution policy [23], where operators are organized for execution by placing data fetching operators (scans) on the server side while the remaining operators of the plan execute remotely on the client. This approach exploits the resources available on the client and saves communication and server resources when the client data is cached. Further implementation details can be found in [51].

The use of the iterator model also enables exploiting pipelined and partitioned parallelism that can be obtained through the addition of an **exchange** operator, as described in [30] and a parallel version of the DOQL evaluator is under development [54]. There are also benefits related to avoiding overheads associated with the materialization of intermediate results in the query execution plan, especially when there are few operators that work in a stop-and-go fashion in the plan.

The loss of performance that is incurred in coupled systems is diminished by accessing the stored data through an API, instead of using the query language provided by the underlying storage mechanism. The use of the API combined with the `MyWalker` class provides a tight approach to coupling, without sacrificing the portability goal.

Fixpoint computation in DOQL is also implemented using the dataflow concept. To compute the fixpoint of a set of algebraic expressions, three iterators are used:

- `fixpoint_iter` which keeps track of the fixpoint computation of each partition of the rule set, keeping the set of tuples that will feed each occurrence of the head predicate in a rule body within each iteration. A partition is a subset of the rule set where each rule in the subset has the same head predicate.
- `feed_iter` which fetches from `fixpoint_iter` all tuples that will be used during an iteration. One `feed_iter` is placed in a plan for each occurrence of an IDB predicate in a rule body, and together with the scan iterators, are the only iterators appearing as leaves in a query processing tree.
- `iterate_iter` which computes a naive or semi-naive (depending on the linearity property of the rule set) evaluation of the operator tree that represents the translation of the rule set into physical operators. The fixpoint iterators that represent each partition of the entire rule set are the inputs to the `iterate_iter`, together with the physical operators that denote the translation of the special *query* predicate. The `iterate_iter` operator iterates until the fixpoint of each input partition is reached. When this happens, the expression denoting the special *query* predicate is evaluated, providing the result to the query.

```

print_iter(
  iterate_iter(
    fixpoint_iter(
      merge_iter(
        apply_iter(
          select_iter(
            seq_scan(train,source.name=S,dest.name=P),
            S=aberdeen
          ),
          struct(S,P)
        ),
        apply_iter(
          select_iter(
            nestedloop(
              feed_iter(reaches,S,L),
              seq_scan(train,source.name=L,dest.name=P)
            ),
            S=aberdeen
          ),
          struct(S,P)
        )
      )
    ),
    apply_iter(
      feed_iter(reaches,S,P),
      struct(P)
    )
  )
)

```

Fig. 15. Physical query plan for Q8

Figure 15 presents the physical plan generated after the optimization of the logical plan described in figure 7. The physical plan describes the choice of physical operators for implementing each logical algebra operator. As shown in the plan, since the boolean predicates that serve as a parameter of the logical `get` operator do not range over any available index, the `get` is translated into a `seq_scan` of the physical algebra.

5.3 Classification of the DOQL Evaluation Strategy

In [5], several criteria are described for distinguishing between different proposals for deductive query processing. The first criterion relates to providing a clear distinction between optimization methods, evaluation methods and methods that combine optimization and evaluation in a single algorithm. The DOQL processing framework provides a clear distinction between optimization and evaluation, allowing the different combination of optimization methods like magic sets [4], normalization [67], and algebraic optimization [63], with evaluation methods like naive and semi-naive evaluation [16].

Other criteria that can be used to classify the evaluation strategy are [4]:

- the DOQL evaluation strategy combines a compiled phase with an interpreted phase, since it is divided in two steps: a compilation phase which accesses only the intensional database and generates an object program (physical algebra), and an execution phase that interprets the object program (physical algebra) against the stored data.
- the DOQL evaluation strategy is an *iterative* strategy, since the target program (physical algebra) has an iterate operator that evaluates the physical operators in scope until the fixpoint is reached, without using a stack as a control mechanism.
- the DOQL evaluation strategy is a *bottom-up* strategy.

Finally, we also classify the evaluation strategy of DOQL as a *dataflow-based* strategy, since the query execution engine is implemented according to the dataflow query engine design described in [31].

6 Related Work

Although there are more than a dozen prototypes that combine deductive and object-oriented functionality in databases (for an overview and comparison of the prototypes, see [52]), Coral++ [56] and Rock & Roll [7,6] are the closest to DOQL. While Coral++ and Rock & Roll have been coupled to the Exodus

storage manager [13], DOQL is used in conjunction with Poet 5.0 OODBMS [48].

Unlike Rock & Roll, which uses a graph-based optimization and evaluation algorithm based on static filtering [37], with limited local optimizations, DOQL supports local optimizations done as logical rewritings, and uses selection pushdown to perform global recursive optimization. DOQL can also perform conventional object-oriented query optimizations due to its support of an object algebra.

Coral++ only supports primitive data types as persistent data types [49]. Query processing in Coral++ does not employ an object algebra, and provides non-ground tuples in the evaluation, thus limiting the possibility of employing more powerful general versions of the magic sets rewriting that can push down conditions like the ground magic sets transformation [43]. In contrast, DOQL provides full support to all user defined data types (classes) in the persistent store, only allows ground tuples in the evaluation, and can combine optimization techniques based on logical rewriting or algebraic optimization.

The query language of DOQL is more powerful than the deductive language of Rock & Roll, and provides language features similar to the ones supported by Coral++. The query model of Coral++ often relies on the programmer to provide control features that can help the optimizer, while DOQL queries are optimized without any control features provided in the query.

Other recent DOOD systems like ROL [41], and Florid [26] lack some features needed in a DBMS. Florid does not support concurrency, persistency or transactions, and ROL does not support concurrency. These systems fall in a class of object-based logic programming systems with limited database features. The query and modelling languages provided by these systems are the most powerful available, with features such as higher-order queries, set unification, incomplete information support, and collections as first class citizens. However, their applicability to mainstream database applications is likely to be limited by the lack of conformance with object database standards and the non-conventional query processing technology that is required in their implementation.

Unlike all other DOOD systems, DOQL uses the ODMG data model and architecture as a reference in its design and implementation, providing a deductive query language implementation for a standard object data model, that can be made available in a mainstream OODBMS.

7 Summary

Implementation: DOQL is implemented in Microsoft Visual C++ 5.0 using standard C++ [58] and STL containers and algorithms [57] in the implementation. Flex and Bison are used in the implementation of the parser, and the source consists of around 19000 lines of code, with a command line interface available. The query component is used in conjunction with the Poet 5.0 OODBMS. This version of the system does not support method calls within the deductive language due to the lack of metatype definitions relating to methods in Poet. OQL in Poet also does not support method calls due to the same limitation.

Contributions: DOQL is a complementary query language to be used alongside the query and programming components of an ODMG compliant object database system. The architecture of the connection between the deductive query component and the underlying database is designed to allow a seamless and portable connection between the query language and the persistent store. It is the first deductive language available for a mainstream OODBMS. The query processing framework is based on an object algebra and combines logical rewriting and algebraic optimization, within clearly defined boundaries. DOQL currently supports local optimizations as logical rewritings, and algebraic global optimizations over global query plans in the presence of recursion. Query evaluation is performed using a dataflow query processing technique.

Despite the decreasing interest in deductive database research in the last 5 years, demonstrating that a deductive query language can be made available as a complementary query component in the context of the ODMG architecture without being intrusive to other components is a positive result for the deductive database community.

It demonstrates that the deductive database paradigm can be used alongside other data modelling and query language paradigms without having to abandon the existing infrastructure or incurring performance penalties for existing applications. The results obtained also reinforce the idea that the existing infrastructure for data storage and query processing in object databases can coexist harmoniously with deductive language facilities.

It is hoped that the design and implementation of DOQL will serve as a foundation for ongoing and future research, leading to the development of other services that can be provided alongside mainstream database systems following a non-intrusive and complementary engineering approach and making use of mainstream query processing technology.

References

- [1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases* (Addison-Wesley, 1995).
- [2] R. Agrawal and P. Devanbu, Moving selections into linear least fixpoint queries, *IEEE Transactions on Knowledge and Data Engineering* **1**(4) (1989) 424–432.
- [3] A. Aho and J. Ullman, Universality of data retrieval languages, in: *Proc. of the 6th ACM Symposium on Principles of Programming Languages* (1979) 110–120.
- [4] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc. of the 5th Symposium on Principles of Database Systems* (1986) 1–15.
- [5] F. Bancilhon and R. Ramakrishnan, An amateur’s introduction to recursive query processing strategies, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1986) 16–52.
- [6] M. Barja, A. Fernandes, N. Paton, M. H. Williams, A. Dinn, and A. Abdelmoty, Design and implementation of Rock & Roll: A deductive object-oriented database system, *Information Systems* **20**(3) (1995) 185–211.
- [7] M. L. Barja, N. W. Paton, A. A. A. Fernandes, M. H. Williams, and A. Dinn, An effective deductive object-oriented database through language integration, in: *Proc. of the 20th VLDB Conference* (1994) 463–474.
- [8] C. Beeri and T. Milo, Functional and predicative programming in OODBs, in: *Proc. of the 11th Symposium on Principles of Database Systems* (1992) 176–190.
- [9] C. Beeri and T. Milo, On the power of algebras with recursion, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1993) 377–386.
- [10] J. Blakeley, Data access for the masses through OLE DB, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1996) 161–172.
- [11] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* (Addison-Wesley, 1999).
- [12] F. Cacace, S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, An overview of the Logres system, in: *Proc. of the Workshop on Combining Declarative and Object-Oriented Databases* (1993) 31–43.
- [13] M. Carey and D. DeWitt, An overview of the EXODUS project, *Data Engineering Bulletin* **10**(2) (1987) 47–54.
- [14] Y. Caseau, An object-oriented deductive language, *Annals of Mathematics and Artificial Intelligence* **3** (1991) 211–258.
- [15] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, *The Object Database Standard: ODMG 2.0* (Morgan Kaufmann, 1997).

- [16] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases* (Springer-Verlag, 1990).
- [17] S. Ceri and R. Manthey, Chimera: A model and language for active DOOD systems, in: *Proc. of the East/West Database Workshop* (1994) 3–16.
- [18] S. Cluet, Designing OQL: Allowing objects to be queried, *Information Systems* **23**(5) (1998) 279–305.
- [19] S. Cluet and C. Delobel, A general framework for the optimization of object-oriented queries, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1992) 383–392.
- [20] B. Dermuth, A. Geppert, and T. Gorchs, Algebraic query optimization in the CoOMS structurally object-oriented database system, in: J. Freytag, D. Maier, and G. Vossen, eds., *Query Processing for Advanced Database Systems* (Morgan Kaufmann, 1994) 121–142.
- [21] L. Fegaras, An experimental optimizer for OQL, Technical Report TR-CSE-97-007, CSE, University of Texas at Arlington, 1997.
- [22] L. Fegaras and D. Maier, Towards an effective calculus for object query languages, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1995) 47–58.
- [23] M. Franklin, B. Jonsson, and D. Kossman, Performance tradeoffs for client-server query processing, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1996) 149–160.
- [24] O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille, Applications of deductive object-oriented databases using DEL, in: R. Ramakrishnan, ed., *Applications of Logic Databases* (Kluwer Academic Publishers, 1995) 1–22.
- [25] O. Friesen, A. Lefebvre, and L. Vieille, VALIDITY: Applications of a DOOD System, in: *Proc. EDBT* (1996) 131–134.
- [26] J. Frohn, H. Himmeroder, P. Kandzia, G. Lausen, and C. Schleppehorst, FLORID - a prototype for f-logic, in: *Proc. of the Intl. Conference on Data Engineering* (1997) 583.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994).
- [28] S. Gatzju, A. Koschel, G. Bultzingslowen, and H. Fritschi, Unbundling active functionality, *ACM Sigmod Record* **27**(1) (1998) 35–40.
- [29] A. Geppert and K. Dittrich, Bundling: Towards a new construction paradigm for persistent systems, *Networking and Information Systems Journal* **1**(1) (1998) 69–102.
- [30] G. Graefe, Encapsulation of parallelism in the volcano query processing system, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1990) 102–111.

- [31] G. Graefe, Query evaluation techniques for large databases, *ACM Computing Surveys* **25**(2) (1993) 73–170.
- [32] G. Graefe, Iterators, schedulers, and distributed-memory parallelism, *Software-Practice and Experience* **26**(4) (1996) 427–452.
- [33] E. Hansom, TriggerMan: an asynchronous trigger processor as an extension to an object-relational DBMS, Technical Report TR-97-024, University of Florida, 1997.
- [34] M. Jarke, R. Gallersdorfer, M. Jeusfeld, and M. Staudt, ConceptBase - a deductive object base for meta data management, *Journal of Intelligent Information Systems* **3** (1994) 167–192.
- [35] M. Jarke and J. Koch, Query optimization in database systems, *ACM Computing Surveys* **16**(2) (1984) 111–152.
- [36] M. Kifer, G. Lausen, and J. Wu, Logical foundations of object-oriented and frame-based languages, *Journal of the ACM* **42**(4) (1995) 741–843.
- [37] M. Kifer and E. Lozinskii, On compile-time query optimization in deductive databases by means of static filtering, *ACM Transactions on Database Systems* **15**(3) (1990) 385–426.
- [38] K. Knight, Unification: A multidisciplinary survey, *ACM Computing Surveys* **21**(1) (1989) 93–124.
- [39] R. Lanzelotte, P. Valduriez, and M. Zait, Optimization of object-oriented recursive queries using cost-controlled strategies, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1992) 256–265.
- [40] T. Leung, H. Pirahesh, P. Seshadri, and J. Hellerstein, Query rewrite optimization rules in IBM DB2 universal database, in: M. Stonebraker and J. Hellerstein, eds., *Readings in Database Systems, 3rd Edition* (Morgan Kaufmann, 1998).
- [41] M. Liu, ROL: A deductive object base language, *Information Systems* **21**(5) (1996) 431–457.
- [42] B. Ludascher, R. Himmeroder, G. Lausen, W. May, and C. Schleppehorst, Managing semistructured data with FLORID: a deductive object-oriented perspective, *Information Systems* **23**(8) (1998) 1–25.
- [43] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan, Magic conditions, *ACM Transactions on Database Systems* **21**(1) (1996) 107–155.
- [44] I. S. Mumick and K. A. Ross, Noodle: A language for declarative querying in an object-oriented database, in: *Proc. of the Third Intl. Conference on Deductive and Object-Oriented Databases* (Springer-Verlag, 1993) 360–378.
- [45] M. T. Oszu and J. A. Blakeley, Query Processing in Object-Oriented Database Systems, in: *Modern Database Systems* (Addison-Wesley, 1995) 146–174.

- [46] N. W. Paton and P. R. F. Sampaio, Extending the ODMG architecture with a deductive object query language, in: *Proc. of the 16th British National Conference on Databases* (Springer-Verlag, 1998) 149–164.
- [47] Poet Software Corporation, *Poet ODL Compiler - Reference Guide*, 1998.
- [48] Poet Software Corporation, <http://www.poet.com/techover/>, *Poet Technical Overview*, 1998.
- [49] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri, Implementation of the CORAL deductive database system, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1993) 167–176.
- [50] J. Richardson and M. Carey, Programming constructs for database systems implementation in EXODUS, in: *Proc. of the ACM SIGMOD Intl. Conference on Management of Data* (1987) 208–219.
- [51] P. R. Falcone Sampaio, *Design and Implementation of a Deductive Query Language for ODMG Compliant Object Databases*, Ph.D. thesis, University of Manchester, 2000.
- [52] P. R. F. Sampaio and N. W. Paton, Deductive object-oriented database systems: A survey, in: *Proc. of the 3rd Intl. Workshop on Rules in Database Systems* (Springer-Verlag, 1997) 1–19.
- [53] P. R. F. Sampaio and N. W. Paton, Deductive queries in ODMG databases: the DOQL approach, in: *Proc. of the 5th Intl. Conference on Object-Oriented Information Systems* (Springer-Verlag, 1998) 57–74.
- [54] S. Sampaio, N. Paton, P. Watson, and J. Smith, A parallel algebra for object databases, in: *Proc. of the 2nd DEXA Workshop on Parallel Databases: Innovative Applications and New Architectures* (IEEE Press, 1999) 56–60.
- [55] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*, 3rd ed. (McGraw-Hill, 1997).
- [56] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan, Coral++: Adding object-orientation to a logic database language, in: *Proc. of the 19th VLDB Conference* (1993) 158–170.
- [57] A. Stepanov and M. Lee, *The Standard Template Library*, Hewlett-Packard Laboratories, 1995.
- [58] B. Stroustrup, *The C++ Programming Language*, 3rd ed. (Addison-Wesley, 1997).
- [59] J. Ullman, A comparison between deductive and object-oriented database systems, in: *Proceedings of the 2nd Intl. Conference on Deductive and Object-Oriented Databases* (1991) 263–277.
- [60] J. Ullman, Information integration using logical views, in: *Proc. of the Intl. Conference on Database Theory* (1997) 19–40.

- [61] J. Ullman and C. Zaniolo, Deductive databases: Achievements and future directions, *ACM - SIGMOD Record* **19**(4) (1990) 75–82.
- [62] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. 1* (Computer Science Press, 1988).
- [63] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. 2* (Computer Science Press, 1989).
- [64] S. Urban, A. Karadimce, S. Dietrich, T. Abdellatif, and H. Chan, CDOL: A comprehensive declarative object language, *Data & Knowledge Engineering* **22**(1) (1997) 67–111.
- [65] S. Vandenberg, *Algebras for Object-Oriented Query Languages*, Ph.D. thesis, University of Wisconsin-Madison, 1993.
- [66] V. Vianu, Rule-based languages, *Annals of Mathematics and Artificial Intelligence* **19** (1997) 215–259.
- [67] Z. Xie and J. Han, Normalization and compilation of deductive and object-oriented database programs for efficient query evaluation, in: *Proc. of the 4th Intl. Conference on Deductive and Object-Oriented Databases* (1995) 485–502.
- [68] K. Yokota, H. Tsuda, and Y. Morita, Specific Features of a Deductive Object-Oriented Database Language QUIXOTE, in: *Proc. of the Workshop on Combining Declarative and Object-Oriented Databases* (1993) 89–99.