The Implementation of a Deductive Query Language Over an OODB

Andrew Dinn[1], Norman W. Paton[2], M. Howard Williams[1], Alvaro A. A. Fernandes[1], Maria L. Barja[1]

[1]Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland
{andrew,howard,alvaro,marisa}@cee.hw.ac.uk
[2]Department of Computer Science, Manchester University, Manchester, England
norm@cs.man.ac.uk

Abstract The ROCK & ROLL database system cleanly integrates deductive and object-oriented capabilities by defining an imperative programming language, ROCK, and a declarative, deductive language, ROLL, over a common object-oriented (OO) data model. Existing techniques for evaluation and optimization of deductive languages fail to address key requirements imposed by ROLL such as: strict typing; placement of deductive methods (predicates) within classes; encapsulation; overriding and late binding. This paper describes the task of implementing an evaluator and optimizer for ROLL, explaining how existing implementation techniques for deductive languages were adapted to meet these requirements and extended to support novel types of optimization.[a]

[a] Accepted for publication in: Proc. 4th Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)

1 Introduction

This paper describes the task of implementing of an evaluator and optimizer for a deductive query language (DQL) operating over an object-oriented database. ROCK & ROLL is a deductive object-oriented database (DOOD) system which cleanly integrates a Horn clause logic language ROLL with an imperative database programming language ROCK, thereby providing complementary programming paradigms for database application development [2]. Both languages operate over a common object-oriented data model OM and hence are cleanly integrated with no impedance mismatch.

Unlike most logic languages proposed as part of DOOD systems ROLL is fully object-oriented catering for encapsulation, multiple inheritance, overriding and late binding. Deductive methods are associated with a defining class, either in its public or private interface, and may be overridden by redefinition in subclasses. Late binding ensures the solution space for a query is partitioned, with solutions referring to subclass or superclass instances derived using the respective subclass or superclass methods. Few previously proposed DOOD systems have combined deductive programming with the benefits of the object-oriented paradigm; even fewer have also been implemented.

The presence of these OO features poses new challenges to existing techniques for deductive language evaluation and optimization. Furthermore, it provides new scope for optimization, for example using type (and subtype) information to restrict the solution space for queries. Currently favoured implementation methods must be reconsidered, questioning whether they are still *feasible* and, if so, whether they are the most *suitable* means of implementing such a language.

ROLL was deliberately specified as a pure logic language with stratified negation, no function symbols, and no extra-logical features such as updates or explicit control mechanisms. This ensures that ROLL has a well-defined and well-known semantics and concedes little in expressive power since the omitted features are compensated for by the availability of an OO data model and the imperative language ROCK. However, another important motivation was that a simple logical language would not rule out the use or adaptation of many existing evaluation and optimization methods. The implementation presented here is based on an existing DQL evaluation and optimization method for relational databases, adapted to cope with the OO nature of ROLL and extended to enable novel types of optimization.

Section 2 provides a brief description of the OO data model, OM, and the DQL, ROLL. Section 3 identifies salient differences between ROLL and a conventional relational DQL. Section 4 justifies the choice of evaluation and optimization strategy and describes the selected techniques. Alternative candidates are assessed in section 5 which presents related work. Section 6 draws some conclusions.

2 OM and ROLL Overview

2.1 The Data Model OM

Type Definitions The OM data model consists of *primary* and *secondary* objects, which model atomic values and compound data, respectively. Each object is assigned an *object type*. Built-in object types, integers, reals and strings, are all primary and have predefined behaviour. User-defined primary types are always aliases for built-in types sharing their behavior. Secondary type definitions define the structure of their instances and declare their imperative and deductive method interfaces.

```
type city:                            ROCK:
  properties                            ...
   public:                            ROLL:
    name : string,                     public:
    population : integer;               fast_route(city, motorway)
    ...                                 ...
                                    end-type
```

Attributes A type definition specifies the *structure* of its instances by defining their *properties* and, optionally, their *construction*. The definition names a possibly empty set of referenced types which are its attributes, either public or private. Attributes defined in a type are inherited by its subtype. The above definition establishes a type `city` whose instances have two attributes, `name` and `population`, which are aliases for the primary types `string` and `integer` (the `ROCK:` and `ROLL:` interface specifications will be explained later).

Constructions A type can be defined as being *constructed* from another type by association, sequentiation or aggregation, which model sets, sequences and records, respectively. If a type is defined as having a construction, all its subtypes conform to that construction. However, a type without construction may have subtypes with distinct constructions. The following definitions include a type `road` with subtype `motorway` constructed as a sequence of `city` objects (indicated by the square brackets). Thus, a motorway is modeled by the sequence of cities it visits. Subtype `major_road` might reference its destinations differently e.g. as a sequence of junctions.

```
type road:
  properties:
   public:
    road_name : string,
  ROCK:
   public:
    length() : integer;
  ROLL:
   public:
    connects_to(road);
end-type
```

```
type motorway:
  specializes: road;
  public [city];
  ROLL:
   public:
    connects_to(road);
end-type

type major_road:
    specializes: road
    ...
```

Interface Specifications A public method associated with a particular type may be implemented in either the declarative language ROLL or the imperative language ROCK. This is indicated in a type definition by the keywords `ROCK:` preceding the declaration of the type's imperative methods and `ROLL:` preceding the declaration of the type's declarative methods. The declarations merely specifies the interface for public methods. Public and private method definitions are supplied in a separate class definition.

For example, the class `city` implements a deductive method `fast_route` which is true if there is a route from the recipient to the `city` argument via the `motorway` argument. `fast_route` has no distinguished return type; ROLL methods are not used as functions but rather to validate or instantiate logical relationships. ROCK methods may be functions, as in the case of the method `length` on class `road` which has integer return type, or they may be procedures with void type.

2.2 The Deductive Language ROLL

ROLL is a function-free Horn clause logic language which resembles Datalog but with the following differences. In place of the conventional first order logic syntax $pred\_symbol(a_0, ..., a_n)$, clause heads and body goals are written using the message oriented syntax $message\_id(a_1, \ldots, a_n)@a_0$ which is read as stating "send *message_id* to object a_0 with arguments a_1 through a_n". ROLL variables range over OM types, and clauses are strictly, statically typed (types being inferred, where possible). Clauses are grouped by recipient type, providing the implementation of an interface declared in the type. This enables both encapsulation and overloading of methods. Private methods may only be invoked from within the class on which they are defined or its subclasses. Methods located in subclasses override any method with the same name in parent classes. *Late binding* is used to select the relevant method for execution. `==` denotes unification `=\=` its negation.

System Generated Methods Some methods are automatically generated by the compiler to provide access to object structure. For example, `get_road_name@R == N` allows access to the `string` object `N` occuring as the `road_name` property of instance `R` of class `road`; `is_member(C)@M` allows access to a `city` instance `C` occuring as a member of the construction of `motorway` instance `M`. Note that these system-generated methods are declarative i.e. depending upon the bindings of the variables, `get_road_name@R == N` may test if a `road` has a given name, retrieve the name of a `road`, retrieve `road` instances with a given `road_name` or retrieve `road` and `road_name` pairs. Syntactic sugar has been provided to make programs more readable, for example the `is_member` goal above can be rewritten as `C isin M`.

Example User Defined Method The following example defines the method `fast_route` on class `city`.

```
class city
 public:
  ...
  fast_route(city, motorway)
  begin
   fast_route(End, MWay)@Start :-
     Start isin MWay,
     End isin MWay,
     Start =\= End.
   fast_route(End, MWay)@Start :-
     Start isin MWay,
     Mid isin MWay,
     Start =\= Mid,
     MWay =\= Connect,
     fast_route(End, Connect)@Mid;
  end
end-class
```

The first clause can be read as stating that there is a route from `Start` to `End` via `MWay` if `Start` and `End` are distinct exits in `MWay`. The second clause states that there is a route from `Start` to `End` via `MWay` if `Start` and `Mid` are distinct exits in `MWay` and there is a route from `Mid` to `End` via motorway `Connect`.

ROLL Query Expressions ROLL queries use a syntax similar to set comprehensions to specify objects to be retrieved from the database. A query is an expression whose value can be either an object, a set of objects or a boolean indicating whether the query goal is true or false.

```
[{S} | fast_route(F, M)@S:city, get_name@F == "Edinburgh"] (2.1)
```

The above example query retrieves the set of cities for which there is a fast route to Edinburgh. The result is an association (set) whose members are the relevant instances of `city`. Note that the variable `S` is qualified to be of type `city`. In this case the annotation merely serves to identify the recipient type as an aid to type *inference*. However, such annotations may also be used to impose a type *constraint*, restricting the range of solution bindings to lie within a subtype.

The goals on the right hand side of the vertical bar must be solved using system-generated or user-defined methods. The goal arguments are either free variables for which solutions must be found, constant values such as the string `"Edinburgh"` or (ROCK) program variable inputs identified by the ! prefix.

The projection on the left hand side of the vertical bar specifies which bindings are to be collected from the solution. The keyword `any` requests retrieval of a single solution; braces indicate the set of all solutions. Bindings for several variables may be retrieved by listing the projected variables in angle braces, in which case solutions are returned as an aggregation (tuple) of objects or a set of aggregations. An empty projection merely tests for the presence of solutions, returning `true` if present, otherwise `false`. The following query uses the current binding for the ROCK program variable `start` and finds a `city` to which there is a fast route and the `motorway` to follow.

```
[ any <D, M> | fast_route(D, M)@!start ]                  (2.2)
```

The result is an aggregation with two fields, a `city` and a `motorway`.

3 Defining an Intensional DB over OM

3.1 Using OM as an Extensional DB

It is possible to regard the OM data model relationally, indeed this view is at the heart of the definition of ROLL as a logic language. The system-generated methods associated with each class correspond to implicit relations between the associated types. For example, the method `get_name` on class `city` corresponds to an implicit relation get_name_{city} with arity 2 and schema $(city, string)$. Similarly, the `is_member` method on `motorway` has a corresponding relation $is\_member_{motorway}$ with schema $(motorway, city)$. Similarly, to each secondary class, there corresponds a unary relation true of all instances of the class.

These implicit relations play the part of an extensional database (EDB) over which ROLL methods define an intensional database (IDB) of virtual relations in much the same way as a relational deductive language. Note however, that the domains of these EDB relations are objects rather than values and that while the domain type is defined at the level of generality specified in the type declaration, "tuples" in these relations may reference instances of subtypes.

Although in theory it would be possible to implement the OM data model as a set of relational tables for these implicit relations, in practice the requirement for fast navigation and update of object structure means that it must be implemented as a network structure. Thus, OM is implemented as a network of linked persistent objects using the Exodus Storage Manager and the Exodus E programming language [5, 19]. Each class directly lists its instances. Each secondary object stores direct references to its attributes, and each constructed instance directly references the members of its construction.

Solving Queries over the EDB Solutions to queries over the EDB, i.e. invocations of system-generated methods, are defined as those tuples in the corresponding EDB relation whose entries unify with the bound arguments to the query. If a query contains the goal $m(a_1, \ldots, a_n)@a_0$ whose bound arguments are $a_{i_1}, \ldots, a_{i_k}$ and the method m corresponds to the implicit relation S with signature $(t_0, \ldots, t_n)$ then solutions are defined as $\sigma_{t_{i_1}=a_{i_1}, \ldots t_{i_k}=a_{i_k}}\ S$

In practice such queries may be solved either by fetching objects directly referenced from the recipient argument in the EDB goal or, in the absence of any such argument, by scanning a class extent to resolve the recipient. For each scanned recipient the related objects are either generated or tested for equality depending upon the presence of other argument bindings. For example, the query:

```
[{C} | C:city isin !mway]                                    (3.1)
```

involves traversing from the `motorway` object bound to the ROCK variable `mway` to its members to generate solutions for `C`. So, the system-generated method `is_member` can be regarded as being indexed by the motorway argument.

In ROCK & ROLL inverses are automatically maintained for all secondary-object references [16]. So, given the query:

```
[{M} | !city isin M:motorway]                                (3.2)
```

the `motorway` objects which have the supplied `city` as an exit can be directly generated using the inverse for the $is\_member_{motorway}$ relation. In the absence of such an inverse the `motorway` class would have to be scanned and each instance containing the appropriate city as member used as a solution for `M`.

Solutions for negated EDB goals are defined when all unbound goal arguments are of secondary type i.e. their domain is finite. In such a case bindings for the non-negated goal may be 'subtracted' from the cross-product of the unbound arguments' domains to obtain solutions for the negated goal. So, if a *non-negated* EDB goal whose arguments are the constants $c_1, \ldots, a_k$ and variables $a_{k+1}, \ldots, a_n$ with types $t_1, \ldots, t_n$ has solution S then the corresponding *negated* goal has solution $(\{c_1\} \times \ldots \{c_k\} \times S_{t_{k+1}} \times \ldots, S_{t_n}) \backslash S$ where S_{t_i} is the unary relation defining the class extent of type t_i.

Combining Queries over the EDB Solutions to queries which are a conjunction of EDB goals are defined as the join of solutions to the individual EDB goals, with an equijoin required where goals have shared variables. If the query contains the EDB goals $M_1, \ldots, M_n$ whose solutions are represented by $S_1, \ldots, S_n$ and shared variables in $M_1, \ldots, M_n$ give rise to the equations $a_{i_1} = a_{j_1}, \ldots, a_{i_k} = a_{j_k}$ then solutions to the conjunction are defined as $\sigma_{a_{i_1}=a_{j_1}, \ldots, a_{i_k}=a_{j_k}} S_1 \times \ldots \times S_n$

In practice goals are solved in a particular order and bindings for shared variables derived from one goal are used to restrict the search for solutions from later goals which mention the variable. Note that there is rarely a requirement to perform a full cross product. Evaluation of a goal M_i usually involves processing each tuple $\langle o_1, \ldots, o_{n_i} \rangle$ which solves goals $M_1, \ldots, M_{i-1}$, selecting a binding o_j for a variable mentioned in M_i and traversing to related objects to derive one or more candidate bindings for the other variables appearing in M_i. These are either tested against bound arguments of M_i or appended to the original tuple to form new solutions. A full cross product is only required when goals cannot be ordered to feed each other with relevant bindings. Note also that conjunctions containing negated EDB goals can be solved so long as any variables of primary type which occur as arguments to the negated goal are shared with a positive goal.

3.2 Solving Queries over the IDB

Solutions to queries over ROLL methods can be defined recursively as the union of solutions for each method clause body. This ultimately reduces to combining solutions for conjunctions of EDB goals. If a method has clauses $C_1, \ldots, C_n$ which only contain EDB goals in their body and each clause body has solution $S_1, \ldots, S_n$ respectively then the method has solutions $\bigcup_{j=1}^{n} \pi_{i_{j_1},\ldots,i_{j_k}} S_j$ where $i_{j_1}, \ldots, i_{j_k}$ are the indices at which the method parameters for clause C_j appear in solutions derived using the clause. This definition extends in the obvious way to cope with clauses containing non-recursive ROLL method invocations (IDB goals) and can cope with recursive and negated method invocations by introducing a fixed point operation and imposing a stratification requirement on negated method calls.

Both top-down resolution based methods and bottom-up naive or semi-naive evaluation methods for relational data models [6, 22] can be modified to derive solutions to ROLL queries. They employ combinatoric algorithms which are independent of whether the values combined are tuples of objects or tuples of values. They merely require some method of retrieving solution tuples from the EDB and of identifying or comparing elements of these tuples as described in the previous section. Negation and arithmetic may be included so long as the usual restrictions regarding safeness and stratification are obeyed. The only novelty is the requirement to respect explicit typing of goal variables and method overriding.

3.3 IDB Queries In the Presence of Typing and Overriding

The presence of *subtyped* variables and method overriding adds some complexity to the solution of an IDB query. If the type of a variable appearing as a message argument is more restrictive than the corresponding formal parameter then the restriction serves to reject method solutions. For example, in class `road` the method `connects_to` has one parameter of type `road`. In the query:

```
[{M} | connects_to(M:motorway)@R:road]                    (3.3)
```

the explicit subtyping of variable `M` serves to reject solutions where the binding for argument `M` is not a `motorway`. The method `connects_to` can construct solutions for `M` in `road` or any of its subtypes. So, implicit in this query is a type membership test on candidate solutions for `M`.

Specialization of methods, as is the case with `connects_to` in class `motorway`, further affects the construction of the IDB. The relation for $connects\_to_{road}$ must be partitioned into two separate relations, $connects\_to_{road \setminus motorway}$ and $connects\_to_{motorway}$ depending upon the type of the recipient object. Candidate tuples in the $connects\_to_{road \setminus motorway}$ relation must not only satisfy the clauses for `connects_to` in class `road` but also have a recipient element which is *not* an instance of class `motorway`. This is because candidate tuples with a recipient in subclass `motorway` must be derived using the clauses from the overriding method defined on `motorway`.

Any query or rule which invokes the method `connects_to` with a recipient of type `road` will use the union of these two partition relations as the solution for the `connects_to` goal. If the recipient type is `motorway` then only solutions from the $connects\_to_{motorway}$ relation are valid. If the recipient is of type `major_road`, which inherits its method definition from `road`, then the valid solutions are those in the $connects\_to_{road \setminus motorway}$ relation which have an instance of `major_road` as the recipient element. This treatment can obviously be generalized to cope with method specialization in multiple subclasses.

So, in summary: overridden methods contain implicit *negative* type restrictions excluding recipient bindings from the overriding class(es); method invocations with arguments whose type is stricter than the type in the method interface imply *positive* type restrictions limiting bindings to instances of the subtype. Evaluation of ROLL must recognize and apply these implicit type restrictions during evaluation. Such type restrictions can be satisfied by including negative and positive type membership goals in queries or method clauses wherever the restriction arises. These are EDB goals which can be solved by scanning class extents either including or excluding relevant subclasses. They obviously do not invalidate the IDB query derivation procedure outlined above.

4 Evaluation and Optimization of an OO DQL

4.1 Selection of an Evaluation and Optimization Method

Query evaluation and optimization methods tend to be defined together as they are usually highly interdependent. Evaluation methods divide into top-down methods based on resolution and bottom-up methods based on some form of naive or semi-naive evaluation method. For either approach the dominant concern for any optimization strategy is order of goal evaluation since the essential problem is to limit the combinatoric explosion of solutions. Query transformation techniques which enable the early application of constraints such as inequalities, equalities and type restrictions are also important.

Evaluation Method Bottom up evaluation is attractive for IDB queries because it relies upon conventional DB bulk operations (selects, joins, etc), and hence should allow the use of conventional DB implementation and optimization techniques. Top-down backtracking approaches can be much more selective than bottom-up methods in well-constrained queries. However, in less selective queries where large amounts of data are to be processed, backtracking may require revisiting the same objects many times. Bulk operations are more likely to make localized references to objects, whereas the scattered reference pattern of backtracking methods may lead to much higher object paging overheads. Although top-down techniques have been developed which perform bulk operations they require complex strategies for managing the evaluation process [8, 13] and appear not to improve on the performance of bottom-up methods [23].

We have chosen to implement a bottom-up evaluation method initially, as this copes best with bulk data yet still is able to solve highly specific queries. The intention is that a backtracking top-down evaluation method be developed later, possibly allowing combined top-down and bottom-up evaluation.

Optimization Method A predicate move-around method has been adopted for optimization as it provides a comprehensive strategy dealing with both goal evaluation order and early application of constraints. It also enables reuse of existing RDB query optimization techniques extended to deal with recursion and an OO data model. In particular, constraint propagation methods can be generalized to allow propagation of system-generated method calls and type restrictions as well as inequality and equality constraints, providing more opportunities for optimization than in RDBs.

4.2 Compilation Process

System Graphs Methods and queries are first compiled to produce a modified form of System Graph [11], a possibly cyclic, directed graph whose nodes represent either method clauses (constraint nodes) or method invocations (goal nodes). In the context of deductive relational databases, system graphs constraint nodes are labeled with the inequalities or identities occuring in the clause body. Goal nodes are created for each EDB or IDB predicate mentioned in the query calls graph. IDB goal nodes are linked to the constraint nodes associated with their defining clauses. Constraint nodes are linked to the nodes for the IDB or EDB goals invoked in the related clause. All leaf nodes are EDB goals.

ROLL system graphs are constructed in the same way except that invocations of system-generated methods are treated as constraints rather than goals, hence they are included in the constraint node. Any implicit positive and negative type constraints which arise from variable typing or method overriding are also added to the constraint node. All leaf nodes in the graph are constraint nodes.

Queries can be regarded as a method with one clause whose head contains the variables in the projection and whose body is the query goals, so the compilation technique is uniform for methods or queries.

Processing Trees The system graph is transformed into a Processing Tree [12], a graphical query execution plan expressed in terms of class scans, constraints (selects), projects, unions and joins. ROLL processing trees also employ structure constraints corresponding to system-generated methods and fixed point nodes which schedule recombination of solutions for recursive methods.

A processing tree T is a set of tree sections $\{T_i\}$ each of which is a quadruple $\langle G_i, M_i, C_i, J_i\rangle$. G_i is a goal corresponding to a user defined method or a query. M_i is a merge section which consists of a union node U_i fed by N_i inputs. It may also contain a fixed point node FP_i fed by the union node. M_i merges outputs from the constraint sections $C_{i,1} \ldots, C_{i,N_i}$ which constitute the set C_i. Each constraint section $C_{i,j}$ comprises a possibly empty sequence of constraint nodes $c^1_{i,j} \ldots, c^{n'}_{i,j}$ and a projection node $p_{i,j}$. Each $C_{i,j}$ has a corresponding join section $J_{i,j}$ in the set J_i. Each join section $J_{i,j}$ contains a possibly empty sequence of goal invocations, references either to goal nodes G_k or to fixed feed nodes FF_k, the latter employed where necessary to break cycles (see figure 1).

Each constraint and join section pair corresponds to a exactly one of the clauses in the definition of the goal predicate. Identities, type restrictions and system-generated method invocations appear as constraints in the constraint

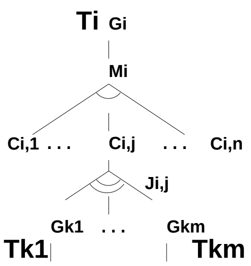

Fig. 1. tree section organization

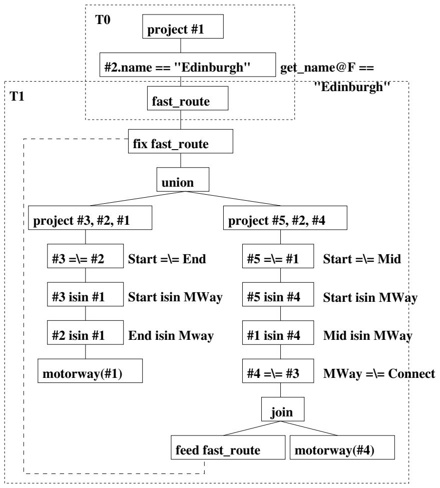

Fig. 2. `fast_route` processing tree

section and user defined method invocations appear as goals in the join section. Identities are added to the constraint section where necessary to equate goal arguments which are bound or shared in the original clause. Tree sections are partially ordered by the rule $T_i > T_k$ *if for some* $J_{i,j} \in J_i$ *goal* G_k *appears in* $J_{i,j}$. The partial order must also have a maximal element T_0, which is the called the root tree section. Leaf tree sections, minimal elements under the partial order, have goal sections which are either empty or only contain fixed feed nodes.

An example processing tree for the query 1.1 is displayed in figure 2. The example tree has two tree sections: T_0, corresponding to the query, and T_1, corresponding to the method `fast_route`. Bindings generated by constraints and by the `fast_route` goal are numbered in each tree section in the order they appear. For example. in T_0 `#1` denotes `S` the recipient of the `fast_route` message, `#2` the `city` argument `F`. To ease readability, constraint nodes have been labeled in the figure with the source lines from which they arise to ease identification of variables.

T_0 only has the `fast_route` goal node in its join section. It contains an attribute constraint in its constraint section which selects solutions with the correctly named `city`.

T_1 has two constraint sections, each of which ends with a projection which feeds into the union node. The left constraint section is fed by a scan over class `motorway`. The right hand constraint section is fed by a join section which combines a scan of `motorway` with solutions from the fixed feed node. The fixed point node feeds solutions to the fixed feed node after removing duplicates, as well as feeding them up through the goal node.

Type	**Input**	**Output**
TypeScan		1-tuple
Constraint	n-tuple	n+k-tuple
FixPoint	n-tuple	n-tuple
FixFeed	n-tuple	n-tuple
Sink	n-tuple	

Type	**Input**	**Output**
XProduct	n-tuple, m-tuple	n+m-tuple
EquiJoin	n-tuple, m-tuple	n+j-tuple ($j < m$)
Union	n-tuple, . . . , n-tuple	n-tuple
Project	n-tuple	m-tuple ($m \leq n$)

Table 1. Processing tree node types, inputs and outputs

Processing tree nodes process zero or more streams of inputs and generate a stream of output tuples. The output stream may be copied to more than one node as input. For example goal outputs will be copied to all join sections which contain an invocation of the goal. The set of available node types with their input and output specifications is provided in Table 1. A list of constraint node types and their parameters is given in Table 2. Note that invocations of side-effect free ROCK methods are allowed as constraints.

Type	**Parameters**
Equal	Var, Var/Value
Attribute	Owner, Name, Value
Set	Collection, Member
Sequence	Sequence, Index, Member
Aggregation	Owner, Field, Value

Type	**Parameters**
Comparison	Left, Op, Right
PosType	Type, Var
NegType	Type_1, ..., Type_k, Var
ROCK Meth	Rcpt, Arg_1, ..., Arg_k, Res
Arith	Left, Op, Right, Result

Table 2. Constraint node types and parameters

4.3 Processing Tree Execution

Execution proceeds bottom up by propagating tuples of objects through the tree. These tuples store intermediate solutions for the query in the same way as tuples stored in temporary tables used for a relational query. However, the tuples themselves are not OM objects so they do not side-effect the query solution process.

In the example tree the scan node on the left subtree of T_1, `motorway(#1)`, generates unary tuples containing `motorway` instances. The member constraints append exits obtained from the `motorway` producing first pairs then triples. These are reordered by the project node to generate solutions for `fast_route`. The fixed point node feeds solutions back into the feed node where they are joined with scanned values forming quadruples which are then propagated through the member constraints in the right hand constraint section to identify indirect exits. Solutions for the goal are projected out and merged with previous solutions by the union node, and possibly passed back to the fixed feed node to be recombined. Solutions from both branches are propagated through the goal node to T_0 where they are filtered by the attribute constraint to select those which for which the destination `city` has name `"Edinburgh"`. The project node at the top of T_0 projects out unary tuples containing bindings for variable `S`.

Ordering Execution Lower nodes are executed before higher nodes to ensure that scans are processed in their entirety wherever possible. To achieve this, fixed point and join nodes store input tuples rather than propagating them immediately. This breaks the dataflow, allowing all inputs to complete processing before generating outputs. It also allows duplicate solutions to be dropped which avoids redundant computation and ensures termination of fixed point computations.

Once a node has been executed it can usually drop all stored solutions, freeing intermediate storage. In the presence of recursions however, some nodes will be repeatedly executed. For example, the join in the right subtree of `fast_route` must retain its inputs after execution because it may be re-executed if the feed node receives new recursive solutions. In such cases the node drops its inputs when all possible feed nodes have finished executing.

4.4 Global Processing Tree Optimization

Optimization of ROLL processing trees proceeds in two stages. *Global* optimization involves moving constraint nodes *downwards* to lower tree sections. *Local* optimization involves reordering joins, constraints and projections *within* a tree section.

ROLL global optimization extends the method of [11] for the propagation of comparison and projection constraints in RDB system graphs to ROLL processing trees. The crucial difference is that system-generated methods, ROCK methods and type restrictions can be treated as constraints to be propagated down the graph. This substantially extends the range of possible optimizations and allows for optimization even in the absence of bound arguments in the query. For example, the explicit type restriction of the `motorway` argument in query 3.3 can be propagated through the `connects_to` goal. The validity of this extension relies on the fact that the treatment in [11] merely treats constraints as predicates and that these new kinds of constraints can also be regarded as predicates for the purpose of propagation.

Rather than perform transformations on ROLL system graphs we have adapted the method to operate direct on processing trees. Query 2.1 can be optimized using this method, propagating the attribute constraint through the fixpoint to each constraint section in the `fast_route` tree section.

The propagation algorithm employs two auxiliary data structures. Associated with each constraint section $C_{i,j}$ in tree section T_i is a set of propagated constraints $Prop_{i,j}$, initially empty, whose members are conjunctions of constraints propagated from callers of the goal G_i. Associated with each invoked goal G_k or fixed feed FF_k in $J_{i,j}$ is a set of relevant constraints $Rel_{i,j,k}$ whose members are conjunctions of constraints which can be propagated to tree T_k. A relevant set initially contains $true$ if the join section references a goal node and $false$ if it refers to a fixed feed node. Note that these sets can be regarded as a logical formula in disjunctive normal form.

Scheduling Algorithm Global optimization propagates constraints from each tree section to the tree sections directly below it in the tree. Initially all tree sections are scheduled for propagation and tree sections are processed top-down starting from the root section.[1] After a tree section has propagated its constraints it normally remains descheduled. However, propagation through a fixed feed node may require the tree section containing the fixed point to be rescheduled, allowing constraints propagated through the fixed feed to be recursively propagated down the tree. Propagation terminates when no tree sections are scheduled.

Propagation Algorithm The propagation step in the algorithm is performed for each scheduled tree section T_i. Constraints are propagated from each constraint section $C_{i,j}$ in a T_i through each goal G_k or fixed feed FF_k in the corresponding join section $J_{i,j}$. Propagation requires identifying the set of *relevant*

[1] The algorithm need not operate top-down but it is terminates quicker if it does.

constraints, i.e. those which apply to the goal in question, and for each constraint section $C_{k,l}$ in the target tree section T_k *mapping* goal arguments in the relevant set to variables projected out of the constraint section, thereby producing a *propagated* constraint set. Propagated constraints may be combined with the *original* constraints in $C_{k,l}$ for further propagation.

A formal description of the propagation algorithm requires some preliminary definitions. A constraint c is relevant to a goal G, written $relevant(c, G)$, if the arguments of c are either constant or equated to arguments to the goal G. $constraints(C_{i,j})$ identifies the constraint sequence in constraint section $C_{i,j}$. The relevant subset of a constraint sequence C with respect to a goal G, written $relevant(C, G)$, is the set $\{c \in C \mid relevant(c, G)\}$.[2] A constraint c is mapped by projection p to a constraint $c' = map(p, c)$ by replacing all goal variables mentioned in c with the variables or values projected by p. If set C contains constraints $c_1, \ldots, c_n$ then then $map(p, C)$ is the set $\{map(p, c_1), \ldots, map(p, c_n)\}$.

The propagation algorithm is given below. Given a tree T_i it computes the propagated constraint set $Prop_{i,j}$ for each constraint section $C_{i,j}$ in T_i and the relevant set $Rel_{i,j,k}$ for each goal G_k or fixed feed FF_k in each join section $J_{i,j}$.

Note that reduction of constraints and testing for contradiction are straightforward operations based solely on simple syntactic considerations. Leaving aside equalities and inequalities, two constraints are equivalent if they are of the same type, their arguments are equal constants or variables and they are both negated or both non-negated. If only one constraint in such a pair is negated then they are inconsistent. Equivalence and consistency of sets of equalities and inequalities can be determined using e.g. the algorithms in [22].[3]

Evaluation in Presence of Propagated Constraints The propagated constraints $Prop_{i,j}$ in constraint section $C_{i,j}$ represent a *disjunctive filter*, each element of which filters out tuples irrelevant to one or more invocation of goal G_i. The evaluation model must be modified accordingly. Each disjunct in $Prop_{i,j}$ is ordered to produce a sequence of constraint nodes[4]. Tuples output by the original constraints in $C_{i,j}$ are fed to each such disjunctive sequence. The outputs of all disjunctive sequences are merged into the projection node. The result is that the filter only rejects a tuple if it is irrelevant to all goals.

Termination of Algorithm The propagation algorithm does not introduce any new constants or variables in constructing the propagated constraint set, so the number of distinct disjuncts which can be added to any set is finite. Furthermore, the construction of the relevant set for a goal can only maintain

[2] Note that *relevant* maps an ordered sequence of constraints to a set of constraints. Constraints in a constraint section must be ordered because some constraints introduce new variables. The output is a set since relevant constraints only mention variables appearing in the goal, hence do not need to be ordered

[3] Equivalence of sets containing positive and/or negative type constraints must ensure that the valid range in the type hierarchy for the type of constrained variables is the same for each set.

[4] Any order will be valid. Identification of an optimal order is attempted during local optimization

or reduce the number of conjuncts added to the relevant set. So, even in the presence of recursive feeds, the propagation algorithm must eventually stabilize with no new constraints propagated.

```
GLOBAL PROPAGATION ALGORITHM:
WHILE deschedule tree section T_i                  – Iterate while still tree sections
  Set R_inh = ∅                                      – to process
  FOR EACH R_p,q,i relating to G_i or FF_i DO        – Collect inherited constraints from
    Set R_inh = R_inh ⋃ R_p,q,i                      – callers of G_i
  END DO
  Reduce R_inh removing implied disjuncts
  FOR EACH constraint section C_i,j ∈ C_i DO        – Iterate over constraint sections
    Set R_map = ∅
    FOR EACH r_inh ∈ R_inh DO                        – Transform inherited constraints
      Set R_map = R_map ⋃ map(p_i,j, r_inh)          – by mapping goal args to local
    END DO                                           – constraint vars
    IF R_map ≠ Prop_i,j THEN
      Set Prop_i,j = R_map                           – Store updated inherited constraints
      SetR = ∅, r_orig = constraints(C_i,j)          – and propagate them to subtrees
      FOR EACH r_map ∈ R_map DO
        Set r_comb = concatenate(r_orig, r_map)      – Combine local and inherited
        IF r_comb is inconsistent                    – constraints
          Set r_rel = FALSE
        ELSE
          Set r_rel = relevant(r_comb, G_k)          –Remove irrelevant conjuncts
          Reduce r_rel removing
              implied conjuncts
        END IF
        Set R = R ⋃ r_rel                            – Combine all relevant conjunctions
      END DO                                         – in a disjunction
      Reduce R removing implied disjuncts
      IF R ≠ Rel_i,j,k THEN
        Set Rel_i,j,k = R                            – Store updated relevant constraint
        Schedule T_k                                 – and schedule inferior tree section
      END IF
    END DO
  END IF
END WHILE
```

Validity of Algorithm In the case of a goal which is not invoked via recursion, the construction of the propagated constraint $Prop_{i,j}$ clearly ensures that the disjunctive filter only rejects tuples which are irrelevant to all invocations of goal G_i. If G_i is called recursively then the final state of $Prop_{i,j}$ cannot contain constraints which reject relevant tuples. Firstly, these constraints could not be propagated directly to the filter since the rejected tuples would then be irrelevant by the argument above. Note, however, that recursively propagated constraints can only displace the originally propagated constraints in the disjunction $Prop_{i,j}$ if they are equivalent to or implied by the originals. So, if the final state of

$Prop_{i,j}$ rejects relevant tuples then the constraints originally propagated directly to $Prop_{i,j}$ would also reject these tuples.

Removal of Original Constraints After constraint propagation has terminated it may be possible to the remove the original constraints which give rise to propagated constraints. If a constraint c in $C_{i,j}$ appears in each formula in the relevant set $Rel_{i,j,k}$ for some goal G_k and if every formula in the propagated set $P_{k,l}$ in tree T_k implies c then it may be safely removed from $C_{i,j}$, since solutions derived from G_k will already have been filtered to ensure that they satisfy c.

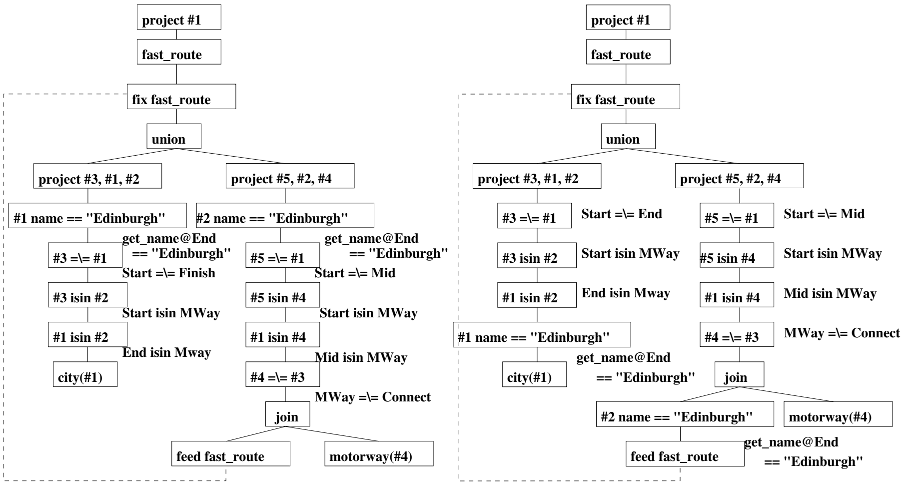

Fig. 3. `fast_route` processing after global (left) and local (right) optimization

4.5 Local Optimization

Local optimization of processing trees uses a heuristic search to reorder the join and constraint nodes in each tree section. This may involve factoring common constraints out of disjunctive constraint sections and moving constraints into the join section. After reordering, projections may also be moved down the tree avoiding generation of redundant outputs. As well as decreasing the size of intermediate solutions this can serve to reduce alternative solutions to duplicates, decreasing the number of tuples stored in join or fixpoint nodes. This stage of the transformation works bottom-up using cost-estimates for subordinate tree sections to cost and rank alternative orderings for a tree section.

Local optimization has not yet been fully implemented. The current compiler does some ordering of constraints before global optimization has been performed based on the constraint type and number of bound arguments. It also reorders constraints after global optimization by locating propagated constraints as low down as possible in subordinate constraint sections.

Figure 3 shows the tree for query 1.1 transformed first by global and then by local optimization.

4.6 Method Precompilation

Both methods and queries can be compiled and optimized as they are added to the DB. Queries embedded in persistent methods are compiled and optimized from scratch as they are expected to be executed many times. Existing, partially optimized processing trees for methods may be incorporated into transient queries and subjected to further local and global optimization as required. In some cases this may mean that query execution repeats work by executing a method several times over the same data. However, it allows more rapid compilation by reusing previous work. It may be the case that for complex queries the saving in compilation time will outweigh the additional execution time.

4.7 Performance Figures

Query 2.1 and query 3.3 have been executed on an example database containing 28 cities, 200 roads and 9 motorways. Motorways have between 3 and 8 exits (average 4.2) and cities appear between 1 and 6 times in a motorway (average 1.4). Execution times for optimized and unoptimized versions of the queries running on persistent[5] and non-persistent environments are provided in Table 3. The optimized `fast_route` query profits from propagation of an attribute constraint through a fixed point. The optimized `connects_to` query profits from propagation of a type constraint.

Query	**Compile Mode**	**Non-Persistent**	**Persistent**
fast_route	Unoptimized	12.5 seconds	184.0 seconds
fast_route	Optimized	0.2 seconds	3.3 seconds
connects_to	Unoptimized	2.6 seconds	67.8 seconds
connects_to	Optimized	0.6 seconds	6.3 seconds

Table 3. Example Query Execution Times

5 Related Work

Much previous work has concentrated on evaluation and optimization methods appropriate to a relational data model. A good summary can be found in [22] or [6]. Two of the best known optimization methods are Query-SubQuery [13] and Magic Sets [1]. These methods only perform propagation of query bindings, presuming that the problem of selecting an efficient order for goal evaluation has been solved[3]. This is a crucial omission since different evaluation orders can lead to vastly different execution times. Other methods (see below) do attempt to deal with the goal ordering problem.

Systems which employ Magic Sets optimization have been extended to cope with set terms, non-ground terms and updates [4, 18]. However, support for these features significantly complicates evaluation and limits opportunity for optimizations. In ROCK & ROLL the need for such features is lessened by the presence of constructed types in OM and the integration of ROLL with ROCK.

[5] There are two implementations of ROCK & ROLL. One locates compiled programs, classes, instances and intermediate data generated during evaluation in a database stored on disk, the other uses virtual memory storage. Performance issues aside, they function identically with the exception that databases created using the persistent program may be reused when the program is rerun.

ROLL is powerful enough to serve as a query language yet simple enough that opportunities for optimization are not lost.

Algebraic rewriting techniques for recursive RDB queries [7] are not dissimilar to the graph reordering method [11] from which the ROCK & ROLL optimization method was derived. [14] describes a graph reordering optimization method for non-recursive RDB queries which is similar to our global optimization strategy. [17] describes another graph reordering method for OODBs, also based on processing trees [12], although this fails to provide a description of the constraint propagation algorithm for recursive queries, and the query language is not truly object-oriented i.e. does not support deductive methods, overriding and late-binding.

None of the above methods employ type information as the basis for optimization. In fact they all need to be modified in order to cope with object types and subtypes and in most cases this adds compilation and evaluation overheads.

Most OODB implementations have not explicitly addressed the question of query optimization. Where optimization has been considered the query language usually does not allow for recursive queries and is rarely truly object-oriented i.e. does not support deductive methods, overriding and late binding [20, 21, 24]. Work which has addressed recursive query optimization has concentrated on very specific concerns such as redundancy elimination or low-level storage structures [10, 15] rather than general-purpose optimization schemes. F-logic has been proposed as a general logical foundation for fully object-oriented deductive query languages although there has been little progress on implementation, let alone optimization. [9] reports progress towards the implementation of a subset of F-logic.

6 Conclusions

New requirements for database systems do not necessarily imply a need for radically new database technology. This paper has shown that existing evaluation and optimization techniques for relational DQLs are certainly applicable to an object-oriented DQL. However, their effectiveness has been reassessed. Well-known optimization techniques such as Magic Sets and QSQ do not allow the nature of the underlying data model to be used in judging the efficiency of the optimization and they do not provide a flexible compilation method which reflects the evolving nature of object-oriented systems.

By contrast the ROLL implementation supports a flexible compilation strategy which allows varying amounts of work to be done in compiling queries and enables reuse of work done at method definition time. This allows quick compilation of ad hoc queries where desired but still enables thorough compilation of queries where required. The optimization and evaluation methods recognize the OO nature of the underlying data model and are geared towards efficient processing of database objects. The implementation has not required the invention of any radically new approaches, merely the judicious selection, adaptation and extension of the most suitable existing techniques.

Acknowledgments This work is supported by grant GR/H43847 from the UK Engineering and Physical Sciences Research council, whose support we are pleased to acknowledge.

References

1. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *SIGMOD*, 1986.
2. M. Barja, N. Paton, A. A. A. Fernandes, M. H. Williams, and A. Dinn. An Effective DOOD Through Language Integration. In *VLDB*, 1994.
3. C. Beeri and S. Naqvi. Sets and Negation in a Logic Database Language (LDL1). In *PODS*, 1987.
4. C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, 1987.
5. M. Carey and D. DeWitt. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, 1990.
6. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. 1990.
7. S. Ceri and L. Tanca. Optimization of systems of algebraic equations for evaluating Datalog queries. In *VLDB*, 1987.
8. S. Dietrich. Extension Tables: Memo Relations In Logic Programming. In *IEEE Symposium on Logic Programming*, 1987.
9. G. Dobbie and R. Topor. A Model for Sets and Multiple Inheritance in deductive Object-Oriented Systems. In *DOOD*, 1993.
10. R. Helm. Detecting and Eliminating Redundant Derivations in Logic Knowledge Bases. In *DOOD*, 1989.
11. M. Kifer and E. Lozinskii. On Compile-Time Query Optimization In Deductive Databases By Means Of Static Filtering. *ACM TODS*, 15(3), 1990.
12. W. Kim. A Model Of Queries For Object-Oriented Databases. In *VLDB*, 1989.
13. A. Lefebvre and L. Vieille. On Deductive Query Evaluation in the DedGin* System. In *DOOD*, 1989.
14. A. Levy, I. Mumick, and Y. Sagiv. Query Optimization by Predicate Move-Around. In *VLDB*, 1994.
15. C. C. Low, H. Lu, and B. C. Ooi. Efficient Access Methods in Deductive and Object-Oriented Databases. In *DOOD*, 1991.
16. N. Paton and A. Abdelmoty. An Object Store for the DOOD Object Model, 1993.
17. R. Lanzelotte and P. Valduriez and M. Zaït. Optimization of Object-Oriented Recursive Queries Using Cost Controlled Strategies. In *SIGMOD*, 1992.
18. R. Ramakrishnan. Magic templates, a spellbinding approach to logic evaluation. In *ICLP*, 1988.
19. J. Richardson, M. Carey, and D. Schuh. The Design of the E Programming Language. *ACM TOPLAS*, 15(3), 1993.
20. G. Shaw and S. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *DOOD*, 1989.
21. D. Straube and T. Öszu. Execution Plan Generation for Database Programming Languages. In *DOOD*, 1991.
22. J. Ullman. *Principles of Database and Knowledge Based Systems*. 1988.
23. J. Ullman. Bottom-up beats top-down for Datalog. In *PODS*, 1989.
24. P. Valduriez and S. Danforth. Query optimization for Database Programming Languages. In *DOOD*, 1989.