Query Optimization in a Deductive Object-Oriented DBMS

Georges Gardarin[1,2], Françoise Fabret[2]

[1] CNRS - MASI/IBP Laboratories
University Pierre et Marie CURIE
45 Avenue des Etats-Unis
78 000 VERSAILLES, FRANCE

[2] INRIA, RODIN Project
BP 105, 78153 LE CHESNAY

Email: georges.gardarin@inria.fr, francoise.fabret@inria.fr

Abstract:

This paper studies the various aspects of query optimization in deductive object-oriented databases. First, a rule language is proposed for object-oriented databases. This language is a pragmatic C++ adaptation of previous proposals. The supported data model is the class model of C++. The language supports recursive rules, method calls and non monotonic updates. Given a query, an optimizer must compile a rule program into an object algebra program. Transformation techniques can then be applied to generate equivalent programs. Possible transformations include operation permutation, fixpoint reduction, integrity constraint addition, predicate simplification and method call optimization. In case of recursive updates, additional memory-based nets are useful to accelerate query evaluation. Possible choices are Rete, Treat or Cosma networks. We discuss and compare these types of accelerators. The search strategy component is in charge of determining the best choice in the solution space of all evaluation approaches and access plans. Finally, we conclude by discussing engineering techniques to develop an integrated optimizer.

1. INTRODUCTION

It is well recognized that integration of the object-oriented and logic paradigms should become a basis for real implementation of database programming languages [Bertino92]. Object-orientation will provide its ability to support reusability, extensibility and data hiding. Logic will bring its formal basis, declarativeness, modularity and readability. Thus, several attempts to combine object-oriented and logic database languages have been reported in the literature [Bancilhon86b, Maier86, Chen88, Kifer89, Abiteboul89]. Most of these proposals are first approaches and seem far from real applications. Two of the most advanced proposals are IQL [Abiteboul90] and LOGRES [Cacace90]

IQL and LOGRES integrate tools to define database schemas. A database schema is a specification of a set of types and instance recipients. In IQL and LOGRES, a type is built using a type expression from a set of primitive types, e.g., integer, real, string and Boolean. Type expressions apply constructors to existing types to derive new types. A constructor is a generic type, i.e., a type parametrized by other types. Classical constructors are the tuple, set, bag (or multiset) and list constructors. Instance recipients in IQL are either classes or relations. A class is a name for a set of object identifiers that have values of the corresponding type. A relation is a name for a set of values of the corresponding type. LOGRES is very similar, except that classes and associations (i.e., relations between objects) are declared in separated sections rather than using the class and relation key-words. Associations correspond to relationships between objects and enforce referential integrity.

In LOGRES and IQL, rules are constructed in the standard way, using positive or negative literals L, L1, L2, ..., Ln. The structure of a rule is $L \leftarrow L1, L2, \ldots, Ln$. Positive literals are truth-valued predicates obtained from terms using equality (i.e., $t1 = t2$) or membership ($t1 \in t2$, also denoted $t2(t1)$). A term is either a constant of a primitive type, or a variable, or any method or constructor applied to a term. Terms are typed; correct typing is required for constructed terms. In IQL and LOGRES, methods are functions attached to classes satisfying the typing constraints given as the method signatures. In both languages, method can be set valued. In IQL and LOGRES, inheritance is a containment relationship established between classes, which makes possible overloading and polymorphism.

IQL and LOGRES are two nice proposals to integrate the object-oriented and logic paradigms. However, they both suffer from a lack of pragmatism. For examples, they are both built upon a specific object-oriented data model and they are not integrated to a general purpose programming environment. A practical language should be based on well recognized programming environments, allowing the user to integrate libraries of existing methods to achieve true reusability. That is why we introduce ROL, a new C++ based rule language for object-oriented databases. As C++ is becoming a defacto standard for object-oriented databases, ROL is built upon the C++ data model. ROL is being developed as a rule compiler integrated to a C++ library, which can be used with any persistent C++. The target language of the rule compiler is a C++ class of algebraic operators to access persistent objects.

The implementation of ROL requires compiling rule modules in optimized programs of the object algebra. Translating rule programs in a graph of algebraic operators is a fairly simple task. Optimizing the graph is more difficult. Query optimization includes syntactic transformations such as operation permutation, redundant sub-query elimination, select migration through join and fixpoint using methods such as Magic Sets or Alexander [Beeri87], etc. It may also include semantic transformations such as query simplification using integrity constraints [Shenoy87] and operator properties (e.g., transitivity of equality). Recursive update optimization is one of the most difficult problem. Several methods have been proposed to optimize updates. Most of them implement a rule representation net to propagate differentially updates. RETE, TREAT and COSMA techniques are the most well known. A survey of these techniques shows that the selection of the net structure to optimize recursive update propagation should be part of the query optimization process.

This paper is organized as follows. After this introduction, in Section 2, an overview of the rule language is given. The ROL rule language integrates the essential concepts of IQL and LOGRES, but is built for the C++ data model. The target language of the optimizer is an extended relational algebra, which is detailed at the beginning of section 3. More generally, section 3 reviews the various transformation techniques that should be integrated in the query optimizer. It includes syntactic and semantic transformations. Section 4 concentrates on recursive update optimization techniques, including an overview of RETE, TREAT and COSMA. Section 5 gives an overview of search strategies that can be used to select the best query graph with additional update propagation nets: exhaustive search, iterative improvement and simulated annealing are surveyed. Section 6 concludes the paper.

2. THE OBJECT-ORIENTED RULE LANGUAGE

In this section, we introduce ROL, a rule language for objects. It can be seen as an extension of Datalog with object-oriented and update capabilities. The main advantages provided by the proposed rule language over a classical rule language are strong support for abstract data types, complex objects with object sharing, inheritance and polymorphism of methods, and update facilities to modify existing objects from rule actions. Furthermore, it is based on a tuple calculus, which makes conditions close to SQL query qualifications.

2.1. The Object-Oriented Data Model

ROL is a rule language to manipulate object-oriented databases. Most object-oriented systems support the C++ data model as object model. Thus, ROL is a rule language for C++ persistent databases. What is truly novel about C++ is its aggregate type class. A class is an implementation of an abstract data type (ADT). A class has members that are attributes and functions. An attribute can be mono-valued or multi-valued. A multi-valued attribute is defined using a template, which is a generic collection of type set, bag or list. An attribute can also be a pointer to an object of another class. A class has both private and public members. Only public members can be invoked from outside the class. Classes may be organized according to an inheritance hierarchy. For a complete description of the C++ data model, the reader can consult [Lippman91].

Collections are organized in different templates, including set, bag, list and array. Standard functions and predicates are supplied at the generic class level to manipulate instances. Examples of functions are: insert or remove an element from a list, convert a list into a set, count the number of elements in a set, map a function to each element of a set, etc. Examples of predicates are: member to check if an element is member of a set or a list, include to check if a set is included in another set, etc.

To illustrate the ROL object model, we give below a database example with attributes that uses the two generic classes list and set. The class PERSON describes persons with a list attribute (Caricature). ACTOR is a subclass of PERSON with an augment function associated. The class FILM describes films with set (Categories) and list (Appears_In) attributes. Note that Appears_In is a list of references (i.e., pointers) to actors. The class DOMINATE shows the result of actor comparative evaluations during film production.

[**Persistent Class Definitions**]

```
CLASS Point { Float Abs ;  Float Ord ;} ;

CLASS Person {Text Name; Text Firstname;  LIST[Point] Caricature; };
CLASS Actor : PERSON {Float Salary; Float Augment (Float Amount) ; };

CLASS Film {Int Numf; Text Title; SET[Text] Categories; LIST[Actor*] Appears_In ;};

CLASS  Dominate { Int Numf;  Actor* Best;  Actor* Worst; };
```

Figure 1: Example of a Database Schema.

2.2. The Rule Language

ROL provides a deductive capability as a rule language to derive new classes, invoke methods or update existing classes. This provides the ROL user with the power of a logic-based language using an object-oriented class model with overloading and polymorphism. Recursive rules are defined through rule heads that refers to the rule body involved classes.

Rules are organized in modules. A rule module refers to persistent or transient classes through variables. Variables are defined globally for rule modules. Variables may be used in rule conditions to select existentially an object in a class. A variable is used as a pointer in rule heads; it has to be previously instanced in the rule condition.

The ROL rule language support method calls with overloading. As in C++, overloading refer to the practice of giving several meanings to an operator or a function. The code selected depends on the types of the arguments used by the operator or function. A method can be invoked in rule heads or body. It is applied to an object using dot notation. To follow object identifiers, path expressions are used; they are denoted as nested function calls and can be understood as such. Special template predicates such as equality, membership and inclusion are supplied within the collection generic classes and can be used in rule qualification.

Figure 2 gives an example of a querying rule for such a database. Note that an attribute is designated using the attribute name as a function. Set valued attributes are unnested using the member predicate. The action of a rule is a method call of the form m(p1,...))

or a sequence of method call written within a bloc as {m1; m2, ...mi;}; for example, print is a method attached to a Terminal class that displays the given arguments on the screen.

```
VAR  Film F, Actor A;
IF F.Appears_In.Member(A) and A.Name ='Quinn' and A.salary ≥ 1,000
        and F.Categories.Member('Adventure ') THEN
        Print(F.Title, F.Categories, A.Salary) ;
```

Figure 2: Example of a ROL querying rule.

A rule head can create transient objects in memory. The transient class must be declared in the module using the TRANSIENT keyword. The attributes of a transient class need not be declared: they are inferred from the rules. A declaration might be useful for type consistency checking. Specific methods can be attached to a transient class. Constructors and destructors of classes are implicitly attached to classes, as in C++. They can be overloaded. The constructor or destructor methods of a persistent or transient class can be called in rule heads using the class name preceded by ~ for the destructor. Figure 3 illustrates the use of a transient class. The first rule constructs objects in the transient class FilmActor by unnesting the Film class; it uses the FilmActor standard constructor. The second rule selects adventure films of Queen in the FilmActor transient class, prints their titles and removes them from the transient class. Note that a rule can explicitly call the class destructor with the reference to the object to destroy as parameter.

```
TRANSIENT CLASS FilmActor ;
VAR Film F, Actor A, FilmActor R;
    IF F.Appears_In.Member(A)  THEN  FilmActor(F.Title;  F.Categories;  A.Name) ;
    IF R.Categories.Member('Adventure') and  (R.Name= "Queen")
          THEN  {Print(R.Title);  ~FilmActor(R);};
```

Figure 3: Inserting and Deleting in a Transient Class.

The rules in Figure 4 defines the transitive closure of Dominate as the transient class Better_Than. The first rule initializes Better_Than with all objects of the Dominate class. Note that the use of a variable D as predicate is a short notation for Dominate(D), which would be redundant with the variable definition. The second rule is recursive. It classifies actors according to the, evaluations. The third rule gives the actors who dominate Quinn, prints their names with a comment.

```
TRANSIENT CLASS Better_Than;
VAR Dominate D, Better_Than B1, B2 ;
        IF D THEN Better_Than(D.Best, D.Worst);
        IF  B1.Worst = B2.Best THEN Better_Than(B1.Best, B2.Worst);
        IF B1.Worst.Name ="Quinn" THEN Print(B1.Best, "was better than Quinn.");
```

Figure 4: Defining a Recursive Transient Class.

3 . QUERY OPTIMIZATION TECHNIQUES

In this section, we survey the query optimization techniques that have to be integrated within the rule compiler. We first focus on the target language of the compiler. We introduce an object algebra, called OOA, derived from an object-oriented extension of relational algebra. A query and its associated set of rules is first transformed in an OOA graph. Logical query optimization consists in transforming the query graph using syntactic or semantic transformation rules. Physical query optimization annotates the query graph with selected indexes and algorithms at each node. We only review the logical optimization techniques.

3.1 The Object-Oriented Target Algebra

Traditionally, a query optimizer translates a rule program (e.g., a DATALOG program) in a tree or a graph of relational algebra operations and optimizes the graph by successive transformations. We propose a similar approach for the ROL optimizer. However, the algebra has to be extended to support complex objects and methods. In this section, we present an object-oriented target algebra, which is called OOA (Object-Oriented Algebra).

OOA is an extended relational algebra to support method calls and objects. This language has been chosen for its power of abstraction from the physical implementation and from the query. OOA operators can easily represent ROL programs. A ROL query is an OOA expression that maps collections into a collection as defined in [Saake89]. In this section, we summarize the main OOA operators required to extend the relational algebra to support the additional ROL capabilities. The additional operators mainly include facilities to invoke methods and generic functions in qualification and projection expressions and a fixpoint operator.

Valuable Expressions and Conditions

The Codd's relational algebra allows the user to reference attribute expressions in query results or conditions. In an object-oriented environment, it is necessary to extend attribute expressions to path expressions and method calls. A path expression is a sequence of attributes of the form A1.A2...An such that Ai (i<n) is a pointer to objects of the class whose Ai+1 is a member. A method expression is a sequence of the form F1.F2...Fn that is applied to objects, where Fi is a method returning an object to which Fi+1 applies. A method expression is a generalization of a path expression. More generally, the algebra refers to valuable expressions, which are either constants, attribute names, or method expressions applied to object pointers. Arithmetic expressions can be used as specific cases of methods. Examples of valuable expressions are given in figure 5.

```
A.Salary                          // Simple attribute expression
A.Augment(100)                    // Simple method expression
D.Best.Salary                     // Path expression
F.Appears_In.First.Augment(100)   // Method expression
```

Figure 5: Examples of valuable expressions.

An elementary condition is defined as being either a Boolean valuable expression or a predicate of the form <Valuable Expresssion> <Comparator> <Valuable expression>. A comparator is chosen among {=, <, ≤, >, ≥, ≠}. A complex condition is a conjunction of disjunction of elementary conditions.

Basic Operations

The basic operations of our algebra are slight extensions of Codd's algebra, as defined for example in [Gardarin89]. These operations are:

(1) **Filter** which produces, from a complex class, a class of the same schema and whose objects satisfy a possibly complex condition.

(2) **Project** which produces a new class from a given class by computing the expressions of source attributes as target attributes.

(3) **Join** which may be defined as a Cartesian product of two classes followed by a filter.

We also include traditional set operations on relations applied to classes: **union**, **difference**, and **intersection**.

Sometimes a sequence of basic operators can be implemented efficiently in a single algorithm (e.g., filter followed by project and n-ary join). Consequently, we define a macro algebra with compound operations. The compound operations are the n-ary union (denoted **union***, which performs the union of N classes) and the n-ary join (denoted **join***, which performs the Cartesian product of N classes followed by a filtering). They also include the search operation, which is a composition of restrictions, n-ary joins and projections. These compound operations are close to generalized tuple calculus expressions referring to classes and methods. They provide the optimizer with the necessary degree of freedom to optimize correctly the queries. Tuple calculus is a good starting point for query optimization since it provides an optimizer with only the basic properties of the query; optimization opportunities may become hidden in a particular sequence of algebra operators like projection, restriction, join [Jarke84].

For illustration, a possible translation of the ROL rule program shown in Figure 2 is represented in Figure 6 to illustrate join and filter operators. The rule program could be translated in a unique search, which would be a composition of the JOIN and the FILTER. For clarity, class names are referenced by one or more letters representing variables assigned to them. Variables can be assigned as synonyms of classes or as results of operations in OOA programs. R represents the result of the program, i.e., the output file.

```
F = FILM ;
A = ACTOR ;
J = JOIN ( {F,A}; {F.Appears_In.Member(A)}; {Tuple(F.Tiltle, F.Categories,
    A.Salary, A.Name)} ) ;
R = FILTER ( {J}; {J.Name = 'Quinn' and J.Salary ≤1,000}; {Print(J.Title, J.Categories,
     J.Salary)})
```

Figure 6: Example of an OOA program.

Iterative Operation and Fixpoint

Recursive views can be easily expressed by a fixpoint operator as shown in [Ullman88], [Ceri86], [Ioannidis87]. This operator simply produces the saturation of a relation computed recursively by an algebra expression. More generally, iterative algebra expression computes recursively a class by applying at each cycle a sequence of OOA operations. The stop condition is the saturation of a class that may be expressed as the absence of objects in a differential class. We introduce a specific method IsEmpty to check wether a class C is empty. A stop condition in a fixpoint is more generally a logical expression (using not, and, or) of Boolean method calls. Thus, the syntax of a fixpoint operation is:

```
<Output Class> = FIXPOINT (<Input Classes>; <Stop Condition>)
DO {<OOA Operations>} ;
```

For example, the fixpoint of the Better_Than class may be computed by the program of Figure 7.

```
BETTER_THAN = Empty();
NEW = PROJECT({DOMINATE D}; {D.Best, D.Worst}) ;
BETTER_THAN = FIXPOINT ({BETTER_THAN, NEW};  Not NEW.IsEmpty)
DO {
    BETTER_THAN = UNION( BETTER_THAN, NEW) ;
    NEWD = JOIN({BETTER_THAN B1, BETTER_THAN B2}; {B1.Worst=B2.Best};
    {B1.Best,B2.Worst}) ;
    NEW = DIFFERENCE(NEWD; BETTER_THAN) ; };
```

Figure 7: A fixpoint computation program.

Query Graph

An OOA program can be represented as a query graph, which is a generalization of a relational algebra tree. Nodes represented by oblongs can be perceived as persistent or transient classes. More precisely, source nodes of the query graph represented by oblongs correspond to persistent classes. Nodes represented by square boxes represent operations such as filter, project and join, and more generally search. An operation has a list of input edges that correspond to persistent classes or to transient classes generated by previous operations. An operation has also a list of output edges that correspond to generated results, which can be perceived as transient classes. Figure 8 gives the query graph corresponding to the program of Figure 7. Note that a fixpoint operation (i.e., a while) is translated as a cycle.

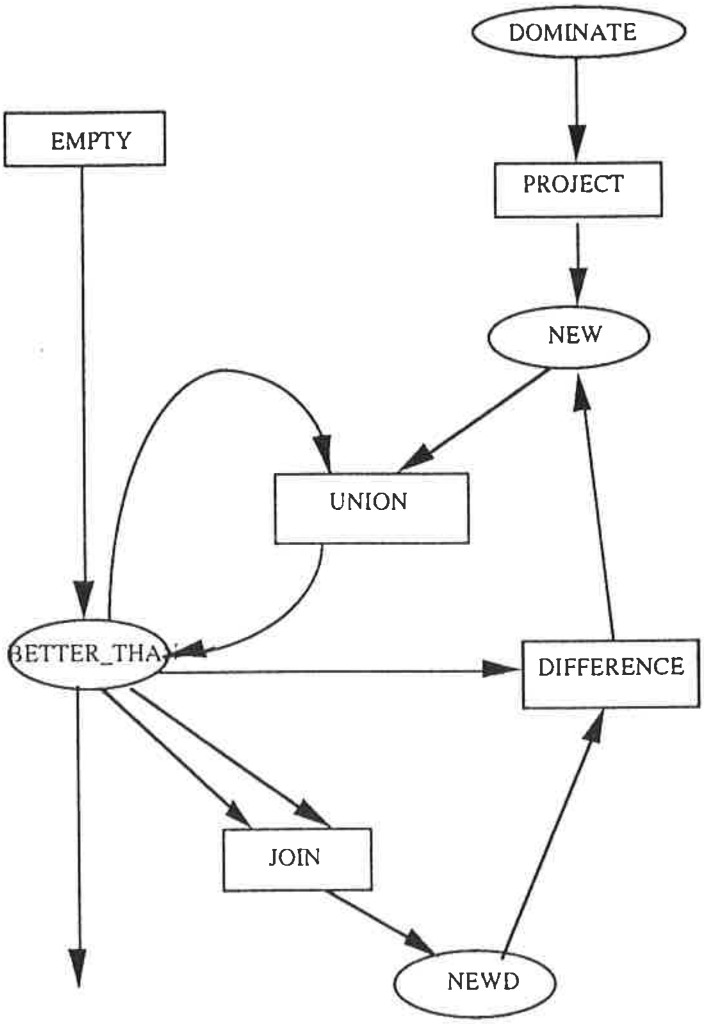

Figure 8: An example of a query graph.

3.2. Syntactic Transformations

Logical query optimization requires to transform the tree to generate equivalent trees that might be more efficiently executed than the original one. Syntactic transformations concentrate on the ordering of operations, without any knowledge of the semantics of methods and operators. In this section, we review the main syntactic transformation rules.

Composition and Decomposition of Operations

Search operations are complex operations that can be decomposed in more elementary operations such as union, filter and join. Even a filter operation with a complex criteria involving conjunctions and disjunctions can be decomposed in cascading filters or unions of filters. In the reverse direction, simple operations can be composed in complex search. Composition and decomposition of operations generate various execution plans. Simple heuristics may be used to reduce the number of combinations, such as group the operations dealing with the same class, etc.

Permutation of Operations

Permutation rules push constraints on classes stored in the database and focus the query on relevant facts. Permutation rules are heuristic and do not guarantee a better processing plan; their role is to propagate constraints on base relations as much as possible. They correspond to well known heuristics such as apply selections before joins. Joins can also be permuted. It is more difficult to choose the right ordering of joins, although semi-joins can be performed first. Permutation techniques are not different from well known relational optimization techniques [Gardarin89].

Common Sub-Query Isolation

Common sub-queries correspond to pending part of the query graph that are duplicated. They should be isolated and replaced by a common sub-graph with a unique output to avoid performing twice the same work. One difficulty is that common sub-query can be generated when decomposing search expressions.

Fixpoint Reduction

In the case of recursive predicates, the permutation between operators cannot be done so easily. However, in cases where a filter qualification is not conflicting with joins involved in the fixpoint loop (i.e., when the filtered attributes are not involved in joins), a simple push of restriction before recursion is possible using the Aho and Ullman's method [Ullman79]. In more complex cases, the application of a rewriting method such as Magic Sets [Bancilhon86a] or Alexander [Rohmer86] is recognized as useful. They transform recursive expressions into expressions that focus on relevant facts. Such methods can be implemented directly on the operation graph [Finance91]. This avoids unnecessary translation from algebra to logic, and from logic to algebra. The transformation method is applied to an algebraic expression E(R). As stated in

[Ioannidis87], an algebraic structure so obtained enables us to get more information about the mechanisms of recursion, and to get more properties of operations. Note also that the composition of operation rule (e.g., the search merging rule) is a typical case of rule that takes advantage of being applied more than once (e.g., before and after pushing selections through fixpoints).

One difficulty with fixpoint reduction in an object-oriented context is dealing with methods. When filter conditions bears upon result of methods, it is very difficult to push filters before recusion. The transformation should take into account the semantic of the method. For example, in the case of path computation, if the condition is path.lenght < 50 and if the recursion only increases the lenght attribute, it is not necessary to take into account initial path with lenght greater than 50. Thus, the filter can be pushed before recursion. More general results are needed concerning this problem.

3.3. Semantic Transformations

Traditionally in query optimizers, decisions are made mainly based on syntactic knowledge and data structures. However, this is not sufficient in object-oriented DBMSs, where users can define their own structures, their own methods and even their own operators. Thus, it is useful that the query optimizer support the introduction of new query transformation rules. In this section, we briefly demonstrate through examples that several types of semantics rules should be introduced as transformation rules on query graphs and search qualifications.

Abstract Data Type Function Rewriting

Certain queries are expressed over well-understood generic abstract data types, such as sets, lists, arrays and bags. These structures have natural algebraic operations (e.g., inclusion, intersection, etc.), and privileged predicates (equality, membership, etc.) associated with them. The properties of these algebraic operations and predicates comprise the implicit semantic knowledge. It defines properties of system constructs, which can be known in the system Let us assume for example that a Min method is defined on sets to compute the minimum value of a set. Then, it is well known that the minimum of the minimums of two sets is the minimum of the union set.

Semantic properties of user defined methods can also be introduced as rewriting rules [Gardarin92]. One further difficulty is estimating the cost for method computation. A cost estimate must also given by the method implementor. In a more sophisticated system, it could be derived from the method code.

Integrity Constraint Based Enrichment

A second kind of semantic knowledge concerns integrity constraints. Choosing appropriate integrity constraints that simplify query processing is a difficult task. Integrity constraints may be added to queries as additional filters. For example, to add a constraint specifying that the Age attribute applied to a Person gives a result between 0 and 120, we simply add the filter Person.Age > 0 and Person.Age < 120. The problem is then to determine if a query graph with this filter (or part of it) is less costly than a query graph without it. Whether it is useful or not to add integrity constraint equation(s) should be determined when choosing a query graph in the search space of all possible query plan. Note that adding integrity constraints may further make applicable certain transformation rules. For example, if the query also state Person.age > 120, then an operator simplification rule should infer an empty answer.

Operator Property Based Simplification

A third kind of semantic knowledge concerns the semantic of operators. In object oriented systems, operators can be overloaded or specific operators can be defined. Thus, it is useful to be able to introduce rules to use the properties of operators in query optimization. An example of such a rule is the transitivity of equality. It can be introduced by the knowledge plugged in the system that a = b, and b = c is equivalent to a = b and a = c. Such transformations greatly increase the search space of the optimizer.

4. RECURSIVE UPDATE OPTIMIZATION

A rule program performing updates (i.e., insertions or deletions) of classes that appear in rule conditions is said to perform recursive updates. The naïve evaluation of recursive updates requires recursive evaluation of rule conditions for modified classes. Data structures to memorize temporary results of rule conditions and to propagate updates have been proposed in the literature. They may be viewed as data flow graphs where input nodes are classes that correspond to predicates in the bodies of rules and output nodes correspond to predicates in rule heads. In an object-oriented environment, instances of such predicates can be maintained in main memory as lists of object identifiers with possibly associated attribute values.

4.1 Principles of Memorization Algorithms

Algorithms that use a memorization structure to propagate updates implement a generalization of semi-naive evaluation. They differ from one another by the choice of the memorised temporary predicates and by the mechanism of propagation in the graph.

More precisely, the choice of the memorized temporary relations may be:

- *Condition oriented*. Some heuristics determine the memorized nodes without taking into account the foreseeable migrations of data w.r.t. the rule heads.
- *Action oriented*. The choice of memorized data is advised by updated predicates in rule heads and migrations of data.

The propagation mechanisms may be:

- *Static*. The materializations are performed before the execution. When the program runs, changes are automatically propagated in the entire structure.
- *Dynamic*. The materializations are deferred and only performed when required for the first time; propagation of the changes takes into account the control strategy.

In the following, we present some algorithms that use memorization techniques. They are devoted to optimize incremental computation of extensions of Datalog programs with non-deterministic semantic. RETE and TREAT are condition oriented, and use static propagation; COSMA is action oriented and use dynamic propagation.

4.2. The RETE algorithm

The RETE match algorithm [Forgy 1982] has been proposed to optimize the execution of the AI production rule programs in main memory. Several algorithms such as DBRete and DBCond [Sellis88] have adapted RETE to the execution of production rule programs in the database context. These RETE like algorithms use a data flow network in which the inputs are the classes invoked in the bodies of the rules. For each rule, filtering conditions (i.e., selections) in the body are evaluated against the database. The corresponding nodes in the network are associated with materialized temporary relation. When an elementary condition is common to several rules, it is only materialized once.

More precisely, let r be a rule with a conjunctive condition, (c0, c1, ...cn) an evaluation order of the elementary conditions of its body and (C0, C1, ...Cn) the sequence of associated base classes. For i varying from 0 to n, RETE materializes the set of objects satisfying the select operation defined by ci in a structure referred as α_memory. Let αi be this memory. These α-nodes are entries to a "comb-like" data structure. A binary operation node in that comb is associated to a β-memory materializing the class βi defined by $(c0,\ldots,ci-1) \wedge ci$. When an update to Ci arises, it is propagated to all β_memories, on the path (αi, βi, ..., βn). The propagation mechanism is static as defined above.

The RETE algorithm is interesting to avoid evaluating several times queries against databases when classes referenced in conditions are updated. It can be extended to complex conditions involving methods. However, it is costly in memory. Thus, a rule optimizer could choose a more sophisticated algorithm that does not instanciate all conditions in main memory.

4.3. The TREAT algorithm

The TREAT algorithm [Miranker88] uses a data flow network similar to RETE. This network has the same inputs and outputs as a RETE network. However, there is no intermediate node. Only the initial selections and the final condition are memorized. The algorithm makes assumption that the system supplies object identifiers. Object identifiers of the source objects are recorded with the objects satisfying the final condition, which allows a direct propagation of deletions. The propagation mechanism is static. The changes on the inputs are propagated one by one.

More precisely, let r be a rule with elementary conditions {c0, c1, ..., cn} in the body. TREAT memorizes {α0, α1, ...αn}, the set of associated α-memories, plus the set γ of instanciations of the body of r. When a change arises on some entry, say αi, γ is updated. If the change is a deletion, then using object identifiers of the deleted objects, γ is directly modified. Else, the changes induced on γ are computed from the change on αi and the set {αj, j≠i} of the other α-memories.

The main advantage of the TREAT algorithm with respect to RETE is to reduce the amount of required memory. However, it requires recomputation of intermediate join. This can be very costly, for example in the case of rules with methods.

4.4. The COSMA algorithm

The COSMA algorithm [Fabret92] uses a data flow network that is similar to the previous ones. However, only effective modifications are recorded in output nodes, which requires a slight extension of the RETE network. For each rule, the inputs are both the classes associated with the elementary conditions in the body (restricted input classes) and the predicates in the head (output classes). Each rule is associated with output nodes that memorizes the effective set of objects to insert or delete for each updated predicate.

The choice for memorizing some intermediate temporary classes in the graph is action oriented and takes into account the explicit control (if any). As TREAT, the algorithm assumes that the system supplies tuple identifiers. Thus, COSMA can be seen as a kind of dynamic TREAT.

More precisely, let r be a rule. A network is created with the rule body and head predicates, making possible the maintenance of αi memories. δk memories may be maintained for head predicates, to memorize the effective updates to perform. A preprocessing phase analyzes how do variable migrate within predicates from the heads to the body of r. This syntactic information advises the memorization heuristic. It allows to isolate subexpressions in the rule body that produce a result useful for the computation of the output memory and that can be efficiently maintained when changes arise on the input classes of r.

4.5. Choice of Classes to Materialize

As shown above, recursive updates may benefit from the maintenance of intermediate memories, say αi for selection conditions, βj for join conditions, γ for the whole rule condition and dk for the rule heads effective updates. The choice of instances to maintain in main memory is not obvious. It depends on the rule program and the available propagation algorithms.

More simply, given a query graph representing a rule program with recursive updates, each node corresponding to a transient class can be maintained in main memory. When a class is materialized, object identifiers of the source objects can be maintained. Thus, if a deletion of a source object appears, the deletion can be directly propagated. More generally, materializing a temporary class avoids full recomputation of a condition against the database when a recursive update occurs. However, materializing a class incurs an additional cost in memory. Thus, the choice of materializing a class is a tradeoff between time and space.

As a synthesis, we could consider that a given query plan can be extended with materialization annotations. A materialization annotation attached to an intermediate class means that the objects of that class with source object identifiers should be maintained in main memory using differential propagation algorithms. Materialization annotations are similar to access path annotations and may change the cost of a given query plan. They should be considered for query plan optimization.

5. SEARCH STRATEGIES TO SELECT A QUERY PLAN

A query plan can be perceived as a query graph with physical annotations. Annotations describe the materialization policies and the selected access methods with indexes for each class node; they specify the selected algorithms for each operation node. With all the possible logical optimizations and the possible variations in annotations, the number of query plans corresponding to a given query over a large set of rules is in general very large.

The problem of query optimization is to find the best query plan. It can be decomposed in two sub-problems. The first sub-problem consists of generating query plans. This can be done using a transformative approach, which starts with a given query plan and transforms it by applying rules. An alternative is to use a generative approach, which generates directly good plans. The second sub-problem is to select the best plan among those generated. The plan selection component of a query optimized is basically a search algorithm in the space of query plans. A plan can be generated by transforming a current plan using a set of transformation rules; a cost can be given to a plan using a cost function. Specifying the cost function is beyond the scope of this paper. The problem we would like now to address is that of selecting the best plan.

Several search strategies have been proposed to explore the search space of query plans [Swami88, Ioannidis90]. Object-oriented implementations of several of them are nicely described in [Lanzelotte91]. In exhaustive search, the entire search space is investigated for the optimal solution. The system R optimizer [Selinger79] exemplifies a restricted case of exhaustive search in which only binary trees without Cartesian products are considered for joins. The search strategy is improved using a branch and bound algorithm. In randomized search, start solutions are first selected to sample the search space. Then, the start solutions are improved until a local optimal solution is obtained. Two variations of randomized search are iterative improvement [Swami88] and simulated annealing [Ioannidis87]. In the following, we give specific methods for implementing variations of exhaustive search, iterative improvements and simulating annealing.

5.1. Exhaustive Search

Full exhaustive search starts from the initial plan derived from the parser, then applies all possible transformations in sequence and finally select the best plan among all the generated ones. An algorithm performing some kind of exhaustive search is given in Figure 9. The Optimal method is supposed to return the plan having a minimum cost among those considered. However, the search time must be limited to avoid looping, for example by the maximum time allowed to the exploration of the search space. StopCond returns true when the search time is exhausted.

```
Function Exhaustive(Query)
{    p:= Parse(Query) ;  Set the intial plan
     S := {p} ; S is the set of  all investigated plans
     while not StopCond()
     {         p' := Transform (p) ; Apply a transformation rule
               if  p' ∉ S then
               {         p ::= p';
                         Insert (S, p') ; Maintain the set of investigated plans
               }
     }
     return Optimal(S) ;  Select best plan among all generated plans
}
```

Figure 9: Exhaustive Search Algorithm.

5.2. Iterative Improvement

Iterative improvement selects initial plans at random in the search space. From an initial state, it applies profitable transformations also selected at random from the applicable rules, to reach a local minimum. The best traversed plan is kept as the optimum The selected initial plans must represent a good sample of the search space. Several solutions exist to generate the initial plans. The algorithm corresponding to iterative improvement is given in figure 10. We start from the plan generated by the parser; the next plan to explore is generated by a random perturbation of the parsed query (Random method). As with exhaustive search, StopCond returns true when the maximum allowed search time is exhausted. The number of profitable transformations that can be applied is limited to MaxMoves, which can depend on the complexity of the query.

```
Function Iterative(Query)
{    p:= Parse(Query) ; Set the initial plan
     S := {} ; S is the set of locally optimum plans
     while not StopCond()
     {        nmoves := 0;
              while nmoves < MaxMoves(Query)
              {        p' := Transform (p) ; Apply a transformation rule
                       if Cost(p') < Cost(p) then
                       {        p ::= p';
                                nmoves ::= nmoves + 1;
                       }
              Insert (S, p') ; Maintain the set of interesting plans
              p := Random(Parse(Query)); Generate a new initial plan at random
              }
     }
     return Optimal(S) ; Select best plan among all generated ones
}
```

Figure 10: Iterative Improvement Search Algorithm.

5.3. Simulated Annealing

Simulated annealing starts from the initial plan and tries to improve it as iterative improvement does. However, a temporary deterioration in cost of the plan is permitted to explore in a better way the search space. A temperature is defined for the modified plan, which varies from a maximum value to 0. At 0, the system is frozen and only profitable moves are accepted. A possible choice for the temperature of a plan p is MaxMove - nmove. Before the system being frozen, the criterion for accepting a transformation is different from that of iterative improvement: a transformation is accepted if it is profitable, but also if it is not with a probability that depends on the temperature and the augmentation of cost. The basic idea is to accept a move that deteriorates the cost with a threshold depending on the system temperature. For example, the probability to accept a bad move may be given by the formula [Ioannidis87]:

$$\text{Prob(temperature, CostBefore, CostAfter)} = e^{-(\text{CostAfter}-\text{CostBefore})/\text{temperature}}.$$

Figure 11 gives the algorithm that implements simulated annealing. MaxTemp is a method setting the initial temperature, which can depends on the query. Equilibrium is a counter that counts the number of successive unchanged states in the inner loop.

Probability(x) is a Boolean function that returns true with a probability given by the argument. To fix ideas, we set the frozen temperature to 0.1 and the maximum number of unchanged states to 4. These could be parameters depending on the query complexity.

```
Function Annealing(Query)
{    p:= Parse(Query) ; Set the initial plan
     temp := MaxTemp(Query) ; Set the temperature at maximum
     S := {} ; S is the set of locally optimum plans
     equilibrium := 0 ; Count the number of successive unchanged states
     while temp  > 0.1 and equilibrium < 4
     {        nmoves := 0;
              while nmoves < MaxMoves(Query)
              {        p' := Transform (p) ; Apply a transformation rule
                       if  Cost(p') < Cost(p)  or Probability(e**-(Cost(p') - Cost(p))/temp)
                       then
                       {        p ::= p';
                                nmoves ::= nmoves + 1; One more move
                                equilibrium := 0 ; Plan have been changed
                       }
                       else equilibrium := equilibrium + 1 ;  No change for one more time
              Insert (S, p') ; Maintain the set of interesting plans
              temp := 0.95* temp ; Decrease temperature
              }
     }
     return Optimal(S) ;  Select best plan among all generated ones
}
```

Figure 11: Simulated Annealing Search Algorithm.

6. CONCLUSION

This paper has presented a rule language for persistent C++ programming environment and sketched the basic techniques required for compiling in optimized operations such an object-oriented rule language. The originality of the language is to integrate nicely the C++ class model. The techniques for optimization includes compiling rule programs into object-oriented algebra graphs, traditional syntactic query graph transformations, less traditional semantic based optimization and materialization of recursively updated classes. All these techniques should be integrated in an intelligent optimizer as OOA graph transformation rules.

One of the main problem of the optimizer is to support a good search strategy. Examples of the most well known strategies have been given as typical programs. Chosing one strategy is probably difficult, although exhaustive search is surely not viable. Organizing query processing knowledge in transformation rules whose application is directed by a strategy component yields a modular architecture for the query optimizer, which is clearly parametrized by the supported transformation rules. Thus, an extensible optimizer could be easily integrated within the proposed framework.

Further research is necessary to better understand what can and cannot be done. Clearly, it appears that very powerful rules and strategies can be integrated within the optimizer. Such rules may lead to long processing if the limits in number of moves and elapsed time are too high. If one stops too early (low limits), then the optimization can be very poor. Thus, a tradeoff has to be found. The limits given to the optimizer could also be allocated dynamically, according to the complexity of the query. Incremental compilation of rule programs could also be considered. Several queries can be issued on the same deduced class. Thus, the same rule program can be considered several times when compiling similar or different queries. More research is required to know what optimized plans can be reused. This could be integrated in the search strategy component, as already known "good" starting plans for the optimization process.

Acknowledgements:

The authors are indebted to Rosana Lancelotte for pointing out the problem of extensible search strategies. They wish also to thank Béatrice Finance, Eric Simon and Patrick Valduriez for helpful discussions on topics covered by this paper.

REFERENCES

[Abiteboul89] Abiteboul S., Kanellakis P., "Object Identity as a Query Language Primitive", ACM SIGMOD Int. Conf., Portland, Oregon, June 1989.

[Abiteboul90] S. Abiteboul, "Towards a Deductive Object-Oriented Database Language", Data & Knowledge Engineering, Vol. 5, No. 4, Oct. 1990.

[Bancilhon86a] Bancilhon F., Maier D., Sagiv Y., Ullman J.D., "Magic Sets and other strange ways to implement logic programs", ACM PODS Symposium, Cambridge, Ma., 1986.

[Bancilhon86b] Bancilhon F., Khoshafian S., "A Calculus for Complex Objects", Proc. ACM PODS, Pp. 53-60, 1986.

[Beeri87] Beeri C., Ramakrishnan R., "On the Power of Magic", ACM PODS Symposium, 1987.

[Bertino92] Bertino E., Montesi D., "Towards a Logical-Object Oriented Programming Language for Databases", Extended Database Technology Conference EDBT'92, Lecture Notes in Computer Science, N° 580, Springer Verlag, March 1992.

[Cacace90] Cacace F., Ceri S., Crespi-Reghizzi S., Tanca L., Zicari R., "The Logres Project: Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm", ACM SIGMOD Int. Conf., Atlantic City, NJ, May 1990.

[Ceri86] Ceri S., Gottlob G., Lavazza L., " Translation and Optimization of logic queries: the algebraic approach", VLDB Int. Conf., Kyoto, August 1986.

[Chen88] Chen Q., Gardarin G., "An Implementation Model for Reasoning with Complex Objects", Proc. ACM SIGMOD Conf., Chicago, 1988.

[Chakravarthy86] Chakravarthy U.S. Grant J., Minker J., "Foundations of Semantic Query Optimization", Int. Workshop on Foundations of Deductive Database in Logic Programming, 1986, chapter 7, p243-273.

[Fabret92] Fabret F., Régnier M., Simon E., "Optimizing Incremental Computation of Datalog Programs with Non-deterministic Semantics", ICDT Internat. Conf., Berlin, October 1992.

[Finance91] Finance B., Gardarin G., "A Rule-Based Query Rewriter in an Extensible DBMS", Data Engineering Int. Conf., Kobe, Japan, April 1991.

[Forgy82] Forgy C., "Rete, a Fast Algorithm for the Many Patterns Many Objects Match Problem", Journal of Artificial Intelligence, N° 19, Pp. 17-37, 1982.

[Gardarin89] Gardarin G., Valduriez P., "Relational Databases and Knowledge Bases", Book, Addison-Wesley, 1989.

[Gardarin92] G. Gardarin, R. Lanzelotte, "Optimizing Object-Oriented Database Queries Using Cost-Controlled Rewriting", Int. Conf. on Extended Data Base Technology (EDBT), Vienna, March 1992.

[Haas90] Haas L.M., Chang W., Lohman G.M., McPherson J., Wilms P.F., Lapis G., Lindsay B., Pirahesh H., Carey M.J., E. Shekita E., "Starbust Mid-Flight: As the Dust Clears", IEEE Transactions on Knowledge and Data Engineering, Vol.2, N.1, March 90.

[Hasan88] Hasan W., Pirahesh H., "Query Rewrite Optimization in Starburst", IBM Research Report RJ 6367 (62349).

[Ioannidis87] Ioannidis Y., Wong E., "Query Optimization by Simulated Annealing", ACM SIGMOD Int. Conf., San Fransisco, CA, June 1987.

[Ioannidis90] Ioannidis Y., Younkyung C.K., "Randomized Algorithms for Optimizing Large Join Queries", ACM SIGMOD Int. Conf., Atlantic City, NJ, May 1990.

[Jarke84] Jarke M., Koch J., "Query optimization in Database Systems", ACM Computing Survey, Vol. 16, p 111-152, June 1984.

[Kiernan90] Kiernan G., De Maindreville C., Simon E., "Making Deductive Database a Practical Technology: a step forward", ACM SIGMOD Int. Conf., Atlantic City NJ., May 1990.

[Kifer89] Kifer M., Lausen G., "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme", ACM SIGMOD Int. Conf., Portland, Oregon, June 1989.

[Lanzelotte91] Lanzelotte R., Valduriez P., "Extending the Search Strategy in a Query Optimizer", Int. Conf. on VLDB, Barcelona, Spain, September 1991.

[Lippman91] Lippman S., "C++ Primer", Book, 2nd Edition, Addison Wesley, 1991.

[Maier86] Maier D., "A Logic for Objects", Proc. of Workshop on Foundations od Deductive Databases and Logic Programming", Washington, USA, 1986.

[Miranker88] Miranker D.P., "Treat: A Better Match Algorithm for AI Production Systems", National Conference on Artificial Intelligence, Seattle, Washington, 1987.

[Rohmer86] Rohmer J., Lescoeur R., Kerisit J.M., "The Alexander-Method, a technique for processing of Recursive Axioms in Deductive Databases", New Generation Computing, N.4, pp. 273-285, Springer-Verlag Ed., 1986.

[Saake89] Saake G., Linnemann V., Pistor P., Wegner L., "Sorting, Grouping and Duplicate Elimination in the Advanced Information Managment Prototype", 1989, VLDB '89, p.307-316.

[Sellis88] Sellis T., Lin C., Raschid L., "Implementing Large Production Systems in a DBMS Environnement: Concepts and Algorithms", ACM SIGMOD Int. Conf., Chicago, May 1988.

[Sciore90] Sciore E., Sieg J., "A Modular Optimizer Generator", IEEE Transactions on Knowledge and Data Engineering, Vol.2, N.1, March 1990.

[Selinger79] Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., Price T.G., "Access Path Selection in a Relational Database Management System", ACM SIGMOD Int. Conf., Boston, June 1979.

[Shenoy87] Shenoy S., Ozsoyoglu Z., "A System for Semantic Query Optimization", ACM SIGMOD Int. Conf., San-Fransisco, June 1987.

[Swami88] Swami A., Gupta A., "Optimization of Large Join Queries", ACM SIGMOD Int. Conf., Chicago, May 1988.

[Swami89] Swami A., "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques", ACM SIGMOD Int. Conf., Portland, Or., June 1989.

[Ullman79] Ullman J.D., Aho A.V., "Universality of Data Retrievial Languages", ACM POPL Int. Conf., 1979.

[Ullman88] Ullman J.D., "Principle of Database and Knowledge-Base Systems", Book, Computer Science Press, Rockville, MD, 1988.

[Wong89] Wong E., Hong W., "Separable Semantic Optimization", Internal Report from EECS Department, U.C. Berkeley, Ca., 1989.