A Rule-Based Query Rewriter in an Extensible DBMS*

Béatrice FINANCE, Georges GARDARIN

INRIA & MASI Laboratories
45 Avenue des Etats-Unis, 78000 VERSAILLES, FRANCE
Email : gardarin@madonna.inria.fr, beatrice@zeus.ibp.fr

Abstract

Query optimizers transform user submitted queries into a canonical form and then translate the canonical form into an efficient access plan. The first step of query optimization is known as query rewriting. This paper describes an integrated approach for query rewriting in an extensible database server supporting ADTs, objects, deductive capabilities and integrity constraints. The approach is extensible through a uniform high level rule language used by the database implementor to specify optimization techniques. This rule language is compiled to enrich the strategy component and the knowledge base of the rewriter. Rules can be added to specify various aspects of query rewriting, including operation permutation, recursive query processing, integrity constraint addition, predicate simplification and method call simplification.

1 Introduction

Query optimization in relational systems extended with abstract data types, object oriented systems and deductive systems is a key issue in current database research. Traditionally, query optimization translates a high level user query into an efficient plan for accessing the database. Query optimization can be divided into two phases [Hasan88]: query rewriting transforms queries into equivalent simpler ones with better expected performance and query planning primarily determines the method for accessing objects. Query rewriting includes syntactic transformations such as query modification [Stonebraker76] with views, redundant sub-query elimination, select migration and fixpoint using methods such as magic sets or Alexander [Beeri87], etc. It also includes semantic transformations such as query simplification using integrity constraints [Shenoy87] and operator properties (e.g., transitivity of equality). The diversity of the tasks which should be integrated in a query rewriter makes it one of the most complex components to write in a DBMS.

Recently, extensible optimizers have been proposed [Freytag87, Graefe87, Hasan88] to deal with the complexity of query rewriting and/or query planning. The key idea is to generate a query optimizer from rules for transforming plans into alternative plans. A few extensible optimizers are operational, among them the EXODUS optimizer [Graefe87] and the STARBUST rewriter [Haas88]. EXODUS was the first system to include a query optimizer generator based on a rule language. The generator produces optimization procedures in the C programming language. Transformation rules specify legal transformations of query trees. They are specified by the database implementor in a language that requires operators and methods arity declaration and rules specifications for operator permutations and implementations. STARBUST has demonstrated that the rule-based approach to query rewriting assures both great flexibility and extensibility [Haas90]. The STARBUST rewriter rule language is the C language; more precisely, a rule is of the form if <C procedure> then <C procedure>. STARBUST designers have chosen to permit the full generality of C rather that commit to a fixed, declarative language. The advantage of such an approach is that it keeps the rule language open (i.e., the use of C avoids the requirement to design a fixed set of primitives for query rewriting) and it facilitates the optimization of the rules. However, a disadvantage is that the rule language is hard to use without deep knowledge of the STARBUST internal structure.

In this paper, we propose a rule language for expressing query transformations that integrates syntactic and semantic optimization for relational, object-oriented and deductive databases. We show that the uniformity and the high level of abstraction of a language based on term rewriting rules associated with a built-in ADT library can model many aspects of logical query optimization. For example, rules may be used to specify relational algebra permutations as well as integrity constraint based simplifications. Rules can be easily added or modified by a database implementor, who can also use the DBMS ADTs facility to extend the optimizer library. Rule conditions and actions are written with a syntax similar to query qualifications, which makes the rule language easy to use. The rule language we propose can be seen as a

* This research is supported by the EDS ESPRIT project (EP 2025).
generalization of that proposed in [Sciore90]. It is mainly extended with ADT method calls and constraints. We demonstrate that such a rule language is powerful enough to express all query rewriting rules in extensible DBMS with object-oriented and deductive capabilities. Our approach makes extensive use of equivalence properties of functions defined on ADTs and shows how extensibility and reusability can be achieved together. This approach has been successfully used for implementing a first version of the query rewriter for the EDS parallel database server [Lopez90]. The server query language is called ESQL (Extended SQL). ESQL integrates the essential concepts of relational, object-oriented and deductive databases. The operational version of the query rewriter is written in PROLOG.

The organization of this paper is as follows. In Section 2, an overview of the ESQL language is presented. Section 3 describes the extended relational algebra, which is the target language of the extensible rewriter (also called the logical optimizer). Section 4 presents the rule language for the extensible rewriter. Section 5 and 6 give significant examples of rewriting rules for syntactic and semantic optimization. Finally, Section 7 concludes the paper.

2 The Extended SQL Language

Basically, the EDS database server uses an extended SQL as a query language, which is the source language of the query rewriter. The extended SQL (ESQL) is intended for traditional data processing applications written in standard SQL as well as non-traditional, data-intensive applications. To promote the access of SQL users to ESQl's new functionalities, the language extension is provided with minimal impact to the SQL syntax. The main advantages provided by ESQL over SQL are strong support for abstract data types, complex objects with object sharing, and a deductive capability to infer new data from stored data.

2.1 Object Oriented Capabilities

The support of ADTs provides a rich typing capability. It makes the fixed set of system-defined types extensible by the users to accommodate their application specific requirements. An ADT is a new type with methods applicable to data of that type. The value of any ADT (e.g., map) can be stored in the database system and manipulated using the associated methods (e.g., map intersection). ESQL supports this ADT capability by extending the notion of domain traditionally supported by relational database systems. The ADT implementation language is C++.

ESQL supports both values and objects. A value is an instance of an ADT while an object has a unique identifier with a value bound to it. Data not declared as objects are values by default as in the relational model, which means that they have no system identifier. Therefore, ESQL data is partitioned into objects and values and only objects may be referentially shared using object identity. To support complex values, ESQL generalizes the relational model with a library of generic ADTs which may be combined at multiple levels. Generic ADTs are higher-order constructors that take as arguments values or objects of any type. The primary generic ADTs are tuple, set, bag, list and array. By combining objects and generic ADTs, arbitrary complex objects can be supported.

Generic ADTs are mainly useful for handling collections. They are organized along an inheritance hierarchy whose root is collection. Collections are subdivided in different types: bag, set, list and array. General functions are supplied at the collection level to convert a collection of one type into another (for example, a bag to a set, which removes duplicates in the set), to check whether a collection is empty, to check if two collections (of the same type) are equal, and to insert and remove an element from a collection. For each collection type, additional functions are included in the system, as illustrated in Figure 1. For example, the MakeSet method creates a new set from a given enumeration of elements, while choice(x) selects an arbitrary element in a set x [Manna85].

To illustrate the ESQL complex object facilities, we give below a database example with queries which uses the two generic ADTs list and set. The list and set generic ADTs are built-in as subtypes of the collection types, as illustrated in Figure 1. The syntax of the type command is defined precisely in [Gardarin90]. Figure 2 gives an example of a database mixing objects and values in relations. The relation FILM describes films with list (Title) and set (Categories) attributes. The relation APPEARS_IN indicates which actor (as a reference to an object describing the actor) appears in which film. The relation DOMINATE shows the result of tennis matches.
organized between actors during film production. Figure 3
gives one example of queries for such a database. Note that
an attribute in a nested tuple is designated using the
attribute name as a function. The system will
automatically apply the appropriate type conversion since
applying an attribute as a function is a projection on a
tuple. As objects are built by combining generic types at
multiple levels, the system offers generic functions for
manipulating user ADTs. This point is explained further
in the following sections.

TYPE Definition

<table>
<thead>
<tr>
<th>Type Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE Category ENUMERATION OF ('Comedy', 'Adventure', 'Science Fiction', 'Western');</td>
</tr>
<tr>
<td>TYPE Point TUPLE (ABS: REAL, ORD: REAL);</td>
</tr>
<tr>
<td>TYPE Person OBJECT TUPLE (Name: CHAR, Firstname: SET OF CHAR, Caricature: LIST OF Point);</td>
</tr>
<tr>
<td>TYPE Actor SUBTYPE OF Person OBJECT TUPLE (Salary NUMERIC);</td>
</tr>
<tr>
<td>FUNCTION IncreaseSalary (This Actor, Val NUMERIC);</td>
</tr>
<tr>
<td>TYPE Text LIST OF CHAR;</td>
</tr>
<tr>
<td>TYPE SetCategory SET OF Category;</td>
</tr>
<tr>
<td>TYPE Pairs LIST OF TUPLE (Pros: INT, Cons: INT);</td>
</tr>
</tbody>
</table>

[Relation Definitions]

TABLE FILM (Numf: NUMERIC, Title: Text, Categories: SetCategory)
TABLE APPEARS-IN (Numf: NUMERIC, Refactor: Actor)
TABLE DOMINATE (Numf: NUMERIC, Refactor1: Actor, Refactor2: Actor, Score: Pairs)

Figure 2: Example of a Database Schema.

SELECT Title, Categories, Salary(Refactor)
FROM FILM, APPEARS-IN
WHERE FILM.Numf = APPEARS-IN.Numf
AND NAME(Refactor) = 'Quinn'
AND MEMBER('Adventure', Categories);

Figure 3: Example of an ESQL query.

2.2 Deductive Capabilities

Knowledge traditionally embedded (with redundancy) in
application programs, can be abstracted in a rule base. The
rule base provides centralized representation of knowledge
and is primarily useful to infer new facts from the facts
stored in the database. ESQL provides this deductive
capability as an extension of the SQL view mechanism.
This provides the ESQL user with the power of the
DATALOG logic-based language using statements already
available in SQL.

To illustrate the ESQL deductive capabilities, we give
in Figure 4 and 5 two examples of view definitions with a
few queries on the deduced relations. The view in Figure 4
gives for each film title the set of categories and the set of
actors playing in the film. Note that the MakeSet function
states that the Actors result attribute is a set (rather than
the default, a bag). The query in Figure 4 finds films with
category 'Adventure' where all actors earn more than
10,000 dollars. It illustrates that the application of the
projection function to a set of tuples gives the set of
projected tuples. It also shows the use of the set quantifier
ALL (EXISTS is also possible) which is defined as a
function applied to a set that returns a Boolean. The view
in Figure 5 is recursive. It classifies actors (by their
reference) according to tennis results. The query in Figure
5 gives the actors who dominates Quinn.

CREATE VIEW FilmActors (Title, Categories, Actors) AS
SELECT Title, Categories, MakeSet(Refactor)
FROM FILM, APPEARS-IN
WHERE FILM.Numf = APPEARS-IN.Numf
GROUP BY Title, Categories;

SELECT Title
FROM FilmActors
WHERE MEMBER('Adventure', Categories)
AND ALL (Salary(Actors) > 10 000);

Figure 4: A nested view definition with an associated
query.

CREATE VIEW BETTER_THAN
(Refactor1, Refactor2) AS
(SELECT Refactor1, Refactor2
FROM DOMINATE
UNION
SELECT B1.Reactor1, B2.Reactor2
FROM BETTER_THAN B1, BETTER_THAN B2
WHERE B1.Reactor2 = B2.Reactor1);

SELECT NAME(Refactor1)
FROM BETTER_THAN
WHERE NAME(Reactor2) = 'Quinn';

Figure 5: A recursive view and an associated query.

3 The Extended Relational
Algebra

The target language of the query optimizer is an extended
form of relational algebra, called LERA. This language
has been chosen for its power of abstraction from the
physical implementation and from the query. LERA
operators can easily represent ESQL queries. In fact an
ESQL query is a LERA expression which maps
collections into a collection as defined in [Saake89]. In
this section, we summarize the main LERA operators
required to extend the relational algebra to support the
additional ESQL capabilities. The additional operators
include a fixpoint operator, operators to transform simple
values into collections and facilities to invoke ADT
functions in qualification and projection expressions.
3.1 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 relation, a relation of the same scheme and whose tuples satisfy a possibly complex condition.
2. Project which produces a new relation from a given relation by computing the expressions of source attributes as target attributes.
3. Join which may be defined as a Cartesian product of two relations followed by a filter.

We also include traditional set operations on relations: union, difference, and intersection.

A LERA tree is the internal representation of a user query for both logical and physical optimization. Sometimes a sequence of basic operators can be implemented efficiently in a particular order (e.g., filter followed by project). Consequently, we define a macro algebra with compound operations including n-ary union (denoted union, which performs the union of N relations), n-ary join (denoted join*, which performs the Cartesian product of N relations followed by a filtering), and a compound operation of both projection, restriction and n-ary join (denoted search). These compound operators are close to tuple calculus expressions and provide the system with the necessary degrees of freedom to logically, and later physically, optimize 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, the ESQL query shown in Figure 3 is translated in LERA using one of the more powerful compound operators in LERA, a search, as follows: search ((APPEARS_IN, FILM), [1.1=2.1 ∧ name(1.2) = 'Quinn' ∧ member('Adventure',2.3)], (2.2,2.3, salary(1.2)))

3.2 The Fixpoint operation

Recursive views can be easily expressed by a fixpoint operator as shown in [Ulman88], [Ceri86], [Ioannis87]. This operator simply produces the saturation of a relation computed recursively by an algebra expression. More precisely, the algebra expression computes new tuples at each cycle. Thus, the general form of the fixpoint operator is fix (R, E(R)) where E(R) is a relational expression using R. The fix operator computes the relation R such that R=E(R). For example, the fixpoint of the BETTER_THAN relation may be computed by the expression:

fix (BETTER_THAN, union ((DOMINATE, search((BETTER_THAN, BETTER_THAN),[1.2=2.1],[1.1,1.2])))

3.3 Support of ADT functions

Complex objects support is provided in LERA by the generalization of the relational operators restriction, projection, and join. Mainly, built-in and user defined function symbols can now appear in the criterion of a restriction, of a join and in the attribute list of a projection. More generally, they can appear in the complex condition or attribute list of a search. Going from an object identifier to a value is performed by the VALUE function, a system built-in function that may also be used in ESQL. The value of an attribute of a tuple is obtained by using the attribute name as a function. Note that, contrary to ESQL where certain syntactic abbreviations are permitted, all function arguments must be correctly typed in LERA. For example Salary(Refactor)>1,000 is a valid condition in ESQL, and should be rewritten with the correct generic function in LERA, as PROJECT(VALUE(Refactor),Salary) > 1,000. Thus, one role of the LERA rewriter is to correctly infer types and add the necessary conversion functions.

3.4 Support of Collections

Collections in LERA are first supported through the use of the collection built-in functions. For example, the MEMBER function can be used to check if an element is a member of a set in a complex condition (e.g., MEMBER('Adventure', Categories)). More generally, all collection manipulation functions accessible in ESQL are included in LERA. In addition, the standard nest and unnest operators are added to LERA to transform columns into collections and conversely (i.e., to support nested relations).

4 Rule Language for Query Rewriting

In this section, we introduce the rule language used to specify the query transformations. The interesting property of this language is that it provides a uniform formalism for writing query transformation rules. The formalism is general enough to incorporate both syntactic and semantic optimization [Jarke84].

4.1 Rule definition

A term rewriting formalism has been chosen to unify the various aspects of query optimization. Functions may use collections as arguments, and this can improve performance. For example, using sets as arguments eliminates the use of permutation rules, as sets are unordered. In addition, as constraints on function arguments are often necessary, the condition part of a rule can include constraints. Unification of terms and changes
in conditions are often necessary when a term (e.g., a LERA expression or an ADT functional expression) is rewritten. Thus, a set of method calls is added in the conclusion of rules. Complex optimization problems can also be handled with such method calls. Note that these methods can be defined by the database implementor as methods of specific ADTs.

The syntax of the rule language is given in Figure 6. A rule can be read as "if the left term appears in the query under the given set of constraints, it is rewritten as the given right term after the application of the given set of methods". A term may contain variables, and also collection variables. Collection variables are symbols representing collections. Rewriting rules using collection variables allow the specification of strategies involving long lists of arguments For example, \( F(\text{SET}(x^*, G(y, f))) \), where \( f \) is a formula and \( x^* \) is a collection variable, is a valid functional expression. Any function known in the system may be used, including LERA operators interpreted as functions, ADT functions or optimizer built-in functions (external functions). A constraint is an additional Boolean condition, which often bears upon the functional expression arguments. It can be empty (no additional condition). Note that the ISA function is used for subtype checking: ISA(\( x, y \)) is true if \( x \) is a subtype of \( y \) or of type \( y \). A rule is only applied to translate a term into another term if all the constraints are true.

\[
\text{Syntax of the rule language.}
\]

Method calls can be useful to deal with complex optimization problems, that require external functions programmed in C. For example, if all variables in a criteria are bound, it can be useful to apply an evaluation function. Methods modify input parameters of the right term, and return them as output parameters used in the left term. These external functions should be defined in the ADT function library of the database. A minimal set of basic functions is built-in to increase the power of the language.

An example of a syntactically correct rule is:

\[
F(\text{SET}(x^*, G(y, f))) / \text{MEMBER}(y, x^*) / \text{true} \rightarrow F(x^*)
\]

Note that a rule has a context, which is the query and the database on which it is applied. Also, all functions including the constraints should be written using known ADT functions. For example, MEMBER is the function defined on the set generic ADT.

4.2 Control and meta-rules

One key aspect of query optimization is the control strategy, which determines what rule to apply and when. Several severe problems appear at this point.

First, certain rules can loop for ever. Thus one important challenge in our approach is to guarantee the termination of the reduction process. Termination of a rewriting rules system is undecidable. However, subsets of rewriting rules can be isolated that either increase or decrease the number of terms in a query. For example, the rule given above decreases the number of terms in a query. Thus, if the rule is applied alone, this process will terminate. However, for the extensible rewriter, termination cannot be guaranteed in a safe way because the database implementor can add or delete rewriting rules.

Second, optimization strategies may require the application of one or more rules up to saturation before applying other rules. For example, rules pushing restrictions before joins may be applied totally before permuting joins. This requires meta-rules that apply a block of rules before another, as done for example in the RDL/1 language [Kiernan90].

To solve both problems, we extend the rule language with a meta-rule language. The meta-rule language uses rule names and allows the database implementor to define blocks of rules which are run up to a certain limit. Blocks of rules are then organized in sequences. The syntax to define a block of rules is: block([rules], value)

The set of rules specifies the rules which are in the block. The value is the maximum number of rule applications
allowed for the block. Each time a rule condition is checked, the limit of the block is decreased by one. An infinite limit means application up to saturation.

The syntax of a meta-rule to force blocks to be run in sequence is: \( \text{seq} \left( (\text{block}), \text{value} \right) \). It defines the order in which the list of blocks in arguments must be applied. The value is the maximum number of applications of the list of blocks (applied in sequence). Thus, any optimizer generated with the rule language is a sequence of blocks of rules which can be applied multiple times. Note that the same rule may appear in different blocks and that the same block may be executed several times. Changing block definitions or the list of blocks in the sequence meta-rule may completely change the generated optimizer.

5 Syntactic Rewriting Rules

Syntactic rewriting rules are used after parsing the query to perform a straightforward translation of an ESQL query into a LERA functional expression. They structure and order LERA operations to yield a canonical form of queries. More precisely, syntactic transformations are divided into three important activities: type checking function rules, syntactic optimization of non recursive predicates, and syntactic optimization of recursive predicates. The first one infers generic functions by doing type checking. The second one permutes and/or groups LERA operations. The third one provides a rewriting method to push selection before recursion. In the following, we detail the second and third types of rules.

5.1 Operation merging rules

Merging rules reduces the size of a LERA program. The goal is to provide a compact representation for the query using search, union, difference, fixpoint and nest/unnest operators. Unnecessary temporary relations are removed. Qualifications are merged together to detect inconsistencies and to ease predicate simplification. In Figure 7, we illustrate the operation merging rules with the search merging rule: two successive search operators are simply merged and their qualifications are connected by "and". The substitute function maps all attributes in the qualification \( f \) or in the expression a of the external search by their corresponding attribute in the internal search. We also give the union merging rule.

```
[Search Merging Rule]
SEARCH ( LIST(x, SEARCH(z, g, b, v), f, a)) /--->
SEARCH( append(x, v, z), f & g, a) / SUBSTITUTE (f, z, f'), SUBSTITUTE (a, z, a')

[Union Merging Rule]
UNION (SET(x, UNION(z))) /---> UNION( set_union(x, z))
```

Figure 7: Examples of merging rules.

These rules eliminate the arbitrary processing order imposed by the user-written (views) through normalization. This normalization provides more opportunity to find the best access plan as all constraints are merged together.

5.2 Operation permutation rules

Permutation rules push constraints on relations 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. We illustrate by giving in Figure 8 two rules which push a search down the tree. The first one is simple: it decomposes a search applied to a union of two relations into a union of two searches. The second one is more complex: it pushes a search through a nest when the search condition does not refer to nested attributes. This additional condition is added to the rule as a constraint using the REFER Boolean external function. This function checks whether the attributes of a given qualification belong to a given list of attributes.

```
[Search through Union Pushing Rule]
SEARCH (LIST(x, UNION (SET (u,v)), y), f, a)) /--->
UNION ( set_union ( SET (SEARCH (append(x,LIST(u), y), f, a)),
SET (SEARCH (append(x,LIST(v), y), f, a)))) /-->

[Search through Nest Pushing Rule]
SEARCH( LIST (x, NEST(z, a, b), y), quali & quali, exp) / REFER(a, quali) -->
SEARCH(LIST(x, NEST (SEARCH (z, quali, exp), a, b), y), quali, exp) / SUBSTITUTE quali, z, a, quali, SCHEMA (a, exp)
```

Figure 8: Examples of permutation rules.

5.3 Fixpoint reduction rules

In the case of recursive predicates, the permutation between operators cannot be done so easily. The application of a rewriting method such as Magic Sets [Bancilhon86] or Alexander [Rohmer86] is recognized as useful. They transform recursive expressions into expressions which focus on relevant facts. In this section, we do not detail the rewriting method but it is implemented directly on the algebra expression. This avoids unnecessary translation from algebra to logic, and from logic to algebra. The rule which invokes the Alexander method is given in Figure 9. In this rule, the Alexander method is applied once only for every recursive predicate. One possibility is to add a condition for triggering the rule. This can be very useful, if we characterize typical recursion that can be computed with an efficient algorithm, without applying the Alexander method.
The fixpoint transformation rule given in Figure 9 pushes selection before recursion. 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 search merging rule is a typical case of rule which takes advantage of being applied more than once (e.g., before and after pushing selections through fixpoints).

6 Semantic Rewriting Rules

Traditionally in query optimizers, decisions are made mainly based on syntactic knowledge and data structures. Semantic query optimization explores all semantically equivalent queries. This increases the search space of the optimizer. However, research in semantic query optimization has demonstrated the potential time saving that can be realized with proper use of inference rules; this should be more significant with complex objects and deductive rules. Database semantic properties are defined at the schema level, and should be integrated in the logical optimizer as one possibility to improve the query for the physical optimizer. Profitable predicates can be added. Predicate elimination is also possible to simplify the qualification in case of inconsistencies.

ESQL queries are expressed over well-understood structures such as sets, lists, arrays, 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 are specified by the database implementor. In the same manner, integrity constraints define properties that are always satisfied on objects, but which are declared by the user. As noted in [Wong89], implicit and explicit semantic knowledge play important and complementary roles to pass from one representation to another one. We now study the integration of these two kinds of semantic knowledge in our extensible optimizer.

6.1 Semantic knowledge definition

The explicit semantic knowledge is composed of the set of integrity constraints declared by the database administrator. An integrity constraint is an axiom that must be satisfied by all data inserted in the database. Integrity constraints can also be expressed on complex objects. Thus, we need a language which integrates the power of ESQL for the database administrator to declare explicit semantic knowledge.

The language we propose for defining constraints is the rules language for defining optimization rules. In addition to the left hand side, the typing predicate ISA can be used to type a relation variable, an object variable or a value variable. For example, we give in Figure 10 integrity constraints, that can be defined on the database of Figure 2.

\[
\begin{align*}
F(x) & \rightarrow F(x) \land \text{ABS}(x) > 0 / \\
F(x) & \rightarrow F(x) \land \text{ORD}(x) > 0 / \\
F(x) & \rightarrow F(x) \land x \in \{\text{Comedy}, \text{Adventure}, \text{Science Fiction}, \text{Western}\} /
\end{align*}
\]

Figure 10 : Examples of integrity constraints.

Implicit semantic knowledge corresponds to well-known properties of ADT functions. For defining implicit semantic knowledge, we propose that the database implementor use the language for making explicit integrity constraint. For illustration, we give in Figure 11 some fundamental properties of the equality operator and the generalization relationship, as defined in [Wong89].

\[
\begin{align*}
(1) \text{ transititivity of operations } & \quad x=y \land y=z \rightarrow x=z /
\end{align*}
\]

Figure 11 : Examples of implicit semantic knowledge.

The addition of semantic knowledge to queries may be useful to further simplify predicates in qualifications and to improve access planning. The addition of a constraint to a query is simply done by the rules given in Figure 10 and 11. Choosing appropriate integrity constraints or implicit constraints that guarantee profitability for the query is a hard problem. In our language, this can be achieved by strengthening the rule condition. A classification of semantic inference rules can be made in order to choose appropriate ones. For example, domain integrity constraints can be very useful; if there exists another constraint on the same attribute, an inconsistency can be detected quickly. For example, MEMBER ('Cartoon', Categories) is inconsistent because MEMBER ('Cartoon', Categories)
('Adventure', 'Adventure', 'Science Fiction', 'Western')) is false.

6.2 Predicate simplification

The predicate simplification block applies rewriting rules to the query qualification to simplify a conjunction or disjunction of predicates. It can perform simple rewriting, but also detect inconsistencies to eliminate part of the qualification. In general, determining if an expression is inconsistent is an NP-complete problem. However, simple inconsistencies can be easily detected. Figure 12 illustrates a few predicate simplification rules, which are expressed with the presented rule language.

[Simplification rules]

- \( x > y / x \leq y / --> \text{true}; f \land \text{false} / --> \text{false} /; \)
- \( x \cdot y = 0 / \text{ISA}(x, \text{constant}), \text{ISA}(y, \text{constant}) --> x = y /; \)
- \( F(x,y) / \text{ISA}(x, \text{constant}), \text{ISA}(y, \text{constant}) --> a /; \)
- \( \text{EVALUATE}(F(x,y), a); \)

Figure 12 : Examples of predicate simplification rules.

7 Conclusion

This paper presented an extensible rewriter for relational databases with complex objects and rules. A high level rule language has been proposed to express simple and complex optimization strategies. The rule language is uniform (based on term rewriting under constraints) and supports syntactic and semantic optimization rules. The DBMS query language (ESQL) provides a good interface for adding new ADTs, and their associated methods that are used in the language. Consequently, reusability of generic methods for developing an extensible optimizer is important. The extensibility is given by offering a rule-based system that will be extended by database designers.

Further research is necessary to better understand what can and cannot be done. Clearly, it appears that very powerful rules can be added to the optimization knowledge base. Such rules may lead to long processing if the application limit is too high. If one stops too early (low limit), then the logical optimization can actually complicate the query. Thus, a trade-off has to be found, mainly for semantic query optimization. The limit given to a block of rule could also be allocated dynamically, according to the complexity of the query. Simple queries (e.g., search on a key) do not need sophisticated optimization: a 0 limit can then be given to all blocks of the query rewriter. Complex queries need rewriting: a high limit can then be given to each rewrite block. Limits can even be adjusted during the query rewriting process.

The next step of query optimization is query planning, also called physical optimization, in the EDS database server project. We believe that the ideas developed in this paper might be applicable to query planning. The association of a high level rule language with appropriate ADTs function calls is a promising approach to generate highly adaptable and extensible physical optimizers.

Acknowledgements

We thank our colleagues Patrick Valduriez from INRIA and Lois Delcambre from LaFayette Louisiana for their careful reading of a first version of the paper and their valuable comments.

References


