Rewrite Rules for Quantified Subqueries in a Federated Database

Graham J.L. Kemp, Peter M.D. Gray and Andreas R. Sjöstedt

Department of Computing Science, King's College,
University of Aberdeen, Aberdeen, Scotland, U.K. AB24 3UE
{gjlk,pgray}@csd.abdn.ac.uk

Abstract

Transforming queries for efficient execution is particularly important in federated database systems since a more efficient execution plan can require many fewer data requests to be sent to the component databases. Also, it is important to do as much as possible of the selection and processing close to where the data are stored, making best use of facilities provided by the federation's component database management systems. In this paper we address the problem of processing complex queries including quantifiers, which have to be executed against different databases in an expanding heterogeneous federation. This is done by transforming queries within a mediator for global query improvement, and within wrappers to make best use of the query processing capabilities of external databases. Our approach is based on pattern matching and query rewriting. We introduce a high level language for expressing rewrite rules declaratively, and demonstrate the use and flexibility of such rules in improving query performance for existentially quantified subqueries. Extensions to this language that allow generic rewrite rules to be expressed are also presented. The value of performing final transformations within a wrapper for a given remote database is shown in several examples that use AMOS II — an SQL3-like system.

1 Introduction

The importance of wide-area querying is increasing, as systems need to bring data together from various sites distributed over the Internet. Where the sites concerned hold large amounts of engineering or scientific data, there is often an underlying database at the site, with its own storage structure and query language. We are developing a federated architecture with a mediator to integrate access to heterogeneous, distributed biological databases [14, 15]. The role of the mediator in this architecture is to receive queries expressed against an integration schema, to rewrite and split these queries into subqueries that refer to the external schemas of distributed databases, to coordinate calls to the external databases, and to combine the results retrieved. Calls to the federation's component database management systems are made through wrappers that translate subqueries into the query language used by the external databases, and process the data values retrieved.

With distributed queries, it is important to do as much as possible of the selection and processing close to where the data are stored, using facilities provided by the federation's component database management systems. To enable this, the subqueries sent to the remote DBMS should contain as much selection information as possible so that the remote DBMS can make best use of its own indexes and optimisation techniques. Also, it is sometimes possible to rewrite the original query in a way that significantly reduces the number of calls that need to be made to remote database.

In this paper, we are interested in the problem of processing complex queries including quantifiers (*some, all*) which have to be executed against different databases. This is an important issue because such queries often involve a lot of iteration, resulting in inefficient execution plans that require many "penny packet" data requests to be sent to the component databases. Therefore, we are investigating how to transform these queries for efficient execution in a growing network of sites accessed through a mediator. These transformations are done at two levels. First, the mediator uses rewrite rules to recognise structural patterns in the entire query that can be replaced with equivalent expressions that require fewer database accesses. Second, after global improvement have been made to the query and it has been split into subqueries destined for remote sites, the wrappers apply further

rewrite rules that are based on classical equivalences to convert a subquery into an alternative form. Local optimisers are often very poor at this for quantified queries, and just use the structure of the query as given. Thus, our mediator architecture incorporates rewrite rules to recognise particular logical structures in the query and casts them in a form that suits the strategy of the remote optimiser.

The existing papers on optimising and evaluating queries with quantified expressions date mostly from the early 80's and assume relational storage on a single processor. A recent paper [2] deals with a semantic data model like FDM, but again the context is fast execution on a single processor or high performance parallel processor. Instead, we focus on adapting the early techniques for reuse in distributed object-based systems, which may include a mix of platforms.

In the next section, we give an overview of the steps involved in processing distributed database queries using our P/FDM mediator. In section 3 we introduce a high level language for expressing rewrite rules and demonstrate the use of such rules in improving queries with existentially quantified subqueries. In section 4 we describe how further transformations can be performed in wrappers to improve the efficiency of queries sent to component databases in a federated system. In section 4.1 we discuss the situation where a component database is itself another P/FDM system [8]. In section 4.2 we show how quantified subqueries in the query fragment destined for execution on a remote AMOS II system [23] can be transformed by the wrapper so that the code generated makes best use of this SQL3-like system. Finally, we discuss the alternative ways of transforming quantified subqueries described in this paper, compare these with other work and discuss the performance gained by rewriting queries as proposed.

2 Processing distributed database queries

We have been experimenting with a system based on Shipman's original high level functional data model (FDM) [24], for distributed heterogeneous databases that are loosely coupled. Queries expressed against this model in the Daplex language (like OQL, but more declarative) are translated to use the query languages on the remote sites. To illustrate the kind of quantified subqueries that are addressed in this paper, Figure 1 shows a simple Daplex query that contains a quantified subquery.

Daplex:

```
for each u in undergrad such that
    some c in takes(u) has level(c) > 3
print(name(u));
```

ZF-expression:

```
{ name(u) | u <- undergrad;
            Exists { c | c <- takes(u);
                         level(c) > 3 } }
```

AMOSQL:

```
SELECT name(u)
FROM   undergrad u
WHERE  some(
    SELECT c
    FROM   course c
    WHERE  c = takes(u)
           AND level(c) > 3
    );
```

Figure 1: This Daplex query returns the name of all undergrads taking at least one course with level above 3. The quantified expression is quantifying the set of all the courses taken by a certain undergrad. If at least one of these courses has a level higher than 3 the result of the existentially quantified expression is true, otherwise it is false, determining whether the name of the undergrad should be returned or not. The corresponding ZF-expression and AMOSQL code are shown.

2.1 ZF-expressions

Daplex queries are normally processed in P/FDM by first being compiled into an internal Intermediate Code ("ICode") [6]. The formal basis for ICode is ZF-expressions[1] (also called "list comprehensions").

A ZF-expression, as shown in Figure 1, consists of a pattern of variables, a list of one or more generators that produce values for the variables and a number of restrictions on those values. The *Exists* quantifier succeeds if the following ZF expression returns a non-empty set. A *generator* is specified as `<var> <- <set>` while a *restriction* is any boolean expression. This makes ZF-expressions very suitable for representing database queries, which are basically descriptions of sets of values.

[1] Zermelo-Fraenkel set expressions, a name taken from the Miranda Functional language [26, 20].

The syntax of the ICode format is very similar to that of ZF-expressions. ICode provides several different generator constructs and a range of possible predicate constructs for the restrictions [3]. These can be combined arbitrarily and without particular order as long as each variable has a generator.

In a simple situation where all data are in a single database, a single code generator is used to transform the ICode into executable code. If the data are spread across different databases in the federation then the P/FDM mediator must split the ICode into fragments, and each fragment is translated into code that can be executed by a particular component database.

2.2 Compiling Daplex queries

Before looking in detail at how the P/FDM mediator improves queries that contain quantified subqueries (sections 3 and 4) we present here an overview of how queries are processed by the system [15].

The P/FDM mediator has several modular components. First, a parser converts the query into ICode. Then the four modules described below are used to improve the original ICode. Each of these modules takes ICode as its input and produces ICode as its output.

- The *simplifier*'s role is to produce shorter, more elegant, and more consistent ICode, mainly through removing redundant variables and expressions (e.g. if the ICode contains an expression equating two variables, then that expression can be eliminated, provided that all references to one variable are replaced by references to the other), and flattening out nested expressions where this doesn't change the meaning of the query.
- The rule-based *rewriter* matches expressions in the query with patterns present on the left-hand side of rewrite rules, and replaces these with the right-hand side of the rewrite rule, after making appropriate variable substitutions. Examples of this are shown in Section 3.
- The *optimiser* [11, 12] performs generic query optimisations.
- The *reordering module* reorders expressions in the ICode to ensure that all variable dependencies are observed.

Typically, queries are improved by first invoking the simplifier, and then the rewriter to perform semantic optimisations. Simplifying the original ICode prior to rewriting makes the rewriter's task easier. The amount of pattern matching code needed in the rewriter to detect whether any subset of expressions indeed matches the left-hand side of a rewrite rule is much less, and the task of matching is more efficient. After all user-defined rewrites have been applied to perform semantic optimisation, the resulting ICode is passed to the optimiser so that generic optimisations can be applied, and alternative execution paths can be costed. The ICode produced by the optimiser is passed once more to the simplifier to streamline the ICode before this is passed to the reordering module.

3 Rewrite rules for transforming quantified subqueries

3.1 Rewrite rule syntax in P/FDM

Our rewrite rules are expressed in a syntax which is designed to be easier to read and maintain than the Prolog that underpins it. We have also used this system to rewrite constraints, which have a very similar syntax [6]. To introduce this syntax, we shall first consider a transformation that is described by Jiao [11, 12]. Jiao did some optimisation work regarding existentially quantified expressions in P/FDM. That work focussed on semantic query optimisation and on rewrites using key values of entity classes.

Example 1

```
for each u in undergrad such that
    some c1 in takes(u) has code(c1) = 'C_331'
print(forename(u), surname(u));
```

Based on a schema declaration that `code(X)` is the uniquely valued key used to identify courses and thus that there can be at most one such course that is taken, we can rewrite this to work more efficiently as:

```
for the u in takes_inv(the c2 in course such that
    code(c2) = 'C_331')
print(forename(u), surname(u));
```

The improved speed comes from re-ordering the query so that it can use an index on `code` for direct access followed by a system-maintained inverse of the `takes` function, instead of by iteration. This example shows how we are able to use the principle of referential transparency to replace one expression by an equivalent one.

This particular transformation can be expressed directly as a rewrite rule:

```
with common
    s in string
rewrite
    u in undergrad such that
```

```
        some c1 in takes(u) has code(c1) = s
into
    takes_inv(the c2 in course such that
                code(c2) = s);
```

This rule is compiled into two ICode expressions with certain parameters in common:

```
u <- undergrad;
Exists{c1 | c1 <- takes(u); code(c1) = s}

u <- takes_inv({c2 | c2 <- course;
                     code(c2) = s})
```

There may be one or more `common` variables which stand for common expressions denoting atomic values (strings, integers, reals, booleans(predicate values)) or entity values (object identifiers). Note that variable names chosen (`u`, `c1`, etc.) have no significance.

Rewrite rules are stored internally in P/FDM as Prolog term structures. Rewriting is done by pattern matching to find whether the left hand side of any rewrite rule is present as a sub-expression. If a match is found, the sub-expression is replaced by the right hand side of the rewrite rule with appropriate variable substitutions. This process is repeated until no more rewrites can be done, and assumes stratified rules as in most such systems.

Note that the rules are Horn-Clause rules based on unification and substitution, working top-down, rather than the bottom-up rules, as used for example in [22] which have right-hand side actions that change the state of various flags. The Horn Clause rules are thus much easier to maintain because they depend on equivalences rather than on side-effects.

The Prolog implementation of this rewrite [12] matches a wider variety of expressions. However, our rule syntax is easier to read and, as we show in section 3.3, recent extensions to the Daplex compiler enable generic rewrites to be expressed declaratively using this syntax.

3.2 Rewrites for distributed queries

Another important consideration is to adapt the rewrite rules for the case of distributed databases when the queries are split and sent to several data sources. Thus, we wish to transform quantified queries so that we can split the work up sensibly between two sites and not send many *penny packet* queries that attract a relatively large communications overhead.

Consider the following example, where `pred(c1)` is a boolean function involving only `c1`:

```
for each u in undergrad such that
    some c1 in course such that
        pred(c1) has level(c1) = year(u)
print(forename(u), surname(u));
```

Suppose that the information on `course` is maintained on the remote database at site 2, while information on `undergrad` is held at site 1. The straightforward way to execute it is to iterate over undergrads on one site and send across one at a time to be used to execute the inner loop (possibly with recompilation as for AMOS II, see section 4.2). However we can rewrite it to generate an inner query whose results are calculated once only and then sent back to the first site thus:

Example 2

```
with common u in undergrad
rewrite
    some c1 in course such that pred(c1)
        has level(c1) = year(u)
into
    year(u) in level(c2 in course such that
                        pred(c2));
```

Note that the rewritten expression denotes a predicate value and that the common expression denotes an `undergrad` entity. Effectively this produces a query executed once only on site 2:

```
print(level(c1 in course such that pred(c1)))
```

with result e.g. *{1,4}* this result is then incorporated into the follow-on query on site 1:

```
for each u1 in undergrad such that
    year(u1) in {1,4}
print(forename(u), surname(u));
```

This follow-on query does not have the overheads of executing an inner loop on another machine across the network. Further, this technique extends recursively to more than two sites.

3.3 Generic rewrites on quantified expressions

The rewrite rules shown in earlier sections have the limitation that their applicability is restricted to the specific classes and functions mentioned in the rule. Many optimisations are more generic, and it would be tedious to have to define a new rewrite rule for all appropriate combinations of classes and functions. Therefore, we have extended the Daplex compiler to accept rewrite rules that are effectively higher order, allowing variables to denote function or class names

(these variables are recognised by a capital initial letter, or an underscore at the start of the variable name). Using this extension, we can express generic rewrite rules, e.g.

```
rewrite
    U1 in ClassU such that
        some C in Rel(U1) has Prop(C) > Val
            and ( some U2 in Rel2(C) has
                    Prop2(U2) < Val2 )
into
    U1 in ClassU such that
        some C in Rel(U1) has Prop(C) > Val
            and Prop2(Rel2(C)) < Val2;
```

This rule can be compiled into an ICode pattern containing Prolog variables, and this pattern will unify with the ICode representation of queries to which the rewrite rule can be applied. It is important to note that the queries that will be pattern matched are themselves well-formed and have been type checked before any rewrite rules are applied.

We are able to express instances of rewrite rules originally noted by Jarke and Koch [10], who used them in an efficient algorithm to process sets of relational tuples, by evaluating nested quantified expressions a block at a time on one processor.

One of these rules uses an inequality comparison, which means that the inner query need only return a single value. In P/FDM form this becomes:

```
rewrite
    E1 in Class1 such that
        all E2 in Class2 such that
            Pred(E2) have Fun1(E1) > Fun2(E2)
into
    E1 in Class1 such that
        Fun1(E1) > maximum(Fun2(E2 in Class2
                        such that Pred(E2)));
```

Similar rules can be written either by using $>=$ everywhere in place of $>$, or else by using $<$ everywhere in place of $>$ and replacing `maximum` by `minimum`, and similarly for $=<$.

We have also introduced a *where-clause* to state conditions that must hold before the rule can be applied, e.g.

```
where
    function KeyFn is the key function
        of class Class and
    function RelInv is the inverse
        of function Rel
rewrite
    Instance in Class such that
        some X in Rel(Instance)
                has KeyFn(X) = KeyValue
into
    RelInv( the X in RelClass such that
                KeyFn(X) = KeyValue )
```

This is a generic version of the rewrite rule presented in section 3.1. We do not use a "`with common`" clause to introduce `KeyValue` since we do not want to put a restriction on its type. The statements in the where-clause are translated into Prolog goals that examine the mediator's metadata before the rewrite rule is applied.

Currently, generic rewrites can include variables representing class names, function names, instance variables, and values. We aim to extend this so that rewrite rules can include variables that represent arbitrary ICode expressions and predicates. At present, such arbitrary rewrites must be coded directly in Prolog, as described in section 4.3.

3.4 Combination of rewrites

We can use the generic rules that flatten nested quantifiers in combination with other domain specific rules, such as those that use indexes. For example, in our database of antibody proteins [13] we model the sequence of amino-acid residues forming a protein chain in several ways. Thus we can find a residue at a given position in any chain either (i) by iterating over residues in each chain and checking `pos(r) = N`, or (ii) by looking for a residue by name via a function acting as a secondary index `r = res_by_name(chain,'CYS')`, or (iii) by a function giving direct access to any residue in position `N` of a chain by `r=absolutepos(chain,N)`. Rewrite rules are used to replace iteration over residues by direct access thus:

Example 3

```
with common i in integer
rewrite     r in residue such that pos(r) = i
into        absolutepos(chain,i);
```

We can then transform the following nested query:

```
for each c in structural_cdr such that
    name(c) = "L1"
    and some r in residue has
            name(r) = "CYS"
            and c in structural_cdr_domain_inv(
                      ig_domain_chain_inv(
                       residue_chain(r)))
            and pos(r) = start(c)
print(protein_code(domain_structure(
       structural_cdr_domain(c))));
```

as if it was written:

```
for each c in structural_cdr such that
    name(c) = "L1"
    and some r in absolutepos(chain,start(c)) has
            name(r) = "CYS"
            and c in structural_cdr_domain_inv(
                      ig_domain_chain_inv(
                       residue_chain(r)))
print(protein_code(domain_structure(
       structural_cdr_domain(c))));
```

If instead of the test `pos(r) = start(c)` we have

```
pos(r) in {start(c) to end(c)}
```

then we could use the following rule to eliminate the existential quantifier:

```
with common
    r in residue, st in integer, fin in integer
rewrite
    some i in {st to fin} has pos(r) = i
into
    pos(r) >= st  and  pos(r) =< fin
```

4 Transforming quantified subqueries in wrappers

We have been testing our strategy in a federated database system, coupling P/FDM [8] to other databases as external data sources, including AMOS II [23]. This enables us to gain experience with SQL3 for an object-relational database. Here our middleware component rewrites the queries to semantically equivalent forms that suit the strategy of the remote optimiser. This is done in various ways, including flattening and reordering.

4.1 Quantified expressions in P/FDM

Where a ZF expression resulting from the distributed execution strategy is to be sent to a remote P/FDM database, it can be sent either as ICode, or else as generated Daplex, which is exactly equivalent. The runtime system of a P/FDM database is implemented mainly in compiled Prolog, with call-outs to C to access local storage modules. The ICode is also compiled directly to Prolog. This is mainly for ease and generality of combining different pieces of generated code with stored procedures, and the convenience of the well-known pattern matching and unification features of Prolog [9, 20].

Unlike the compiler of AMOS II described below, the P/FDM compiler recurses through subqueries making a complete Prolog program of the Daplex query, and does not recall the compiler at runtime. Thus it commits itself to an execution plan based on estimates of class sizes and other statistics known at compile time. It evaluates the `Exists` construct in the ZF expression by means of a Prolog predicate `atleast` that simply keeps track of how many times the subprogram has succeeded and exits as soon as possible from lazy evaluation of a list of values. Thus flattening a number of nested `atleast` constructs will have very little effect on execution times.

4.2 Rewrite rules for AMOS II generation

When AMOS II is an external data source in the federation, ZF-expressions will be translated to AMOSQL, the language of AMOS II. The semantics of quantified expressions are equivalent in the two systems and the translation from ZF-expressions to AMOSQL is largely very straightforward. The main difference [25] lies in that AMOS II supports multiple inheritance, which can only be approximated in Daplex. A similar difference, of course, exists between C++ and Java. Daplex does allow an object to be an instance of more than one supertype, but requires the choice of supertype in a specific query to be made in the query by means of a cast operation (thus the required method definition can be bound at compile time). Daplex also supports a number of arithmetic functions, and a constructor for lists of constants. If these rather unusual constructs are avoided, then queries are easily translated, as our implementation shows.

When AMOS II executes a generated query it starts by compiling it, producing a query plan in a form of set expressions similar to ZF-expressions. However, the compiler does not traverse subqueries during this compilation phase.

This means that after the compilation phase, during the evaluation of the global query, the compiler will be called to compile the inner quantified query for each member retrieved from iteration over outer sets. This means a large overhead in execution time, making it very important to minimise the nesting of quantified expressions so as to reduce calls on the compiler during execution. We can do this conveniently by incorporating a final phase of rewriting that is target specific, as part of the mediator. This architecture conveniently accommodates known restrictions on remote DBMS optimisers, which may be given expres-

sions they are not used to! Such expressions are a common side-effect of query transformation and mapping between heterogeneous systems. Thus the extra rewriting phase is very important in practical applications.

4.3 Unnesting nested quantified expressions

Fortunately, nested existentially quantified expressions can be unnested without altering the semantics. Thus we start with a ZF expression containing nested quantifiers such as:

Example 4 This returns the name of all the undergrads taking at least one course with level above 3 having at least one enrolled undergrad of age below 19.

```
{name(u1) | u1 <- undergrad;
   Exists {c | c <- takes(u1); level(c) > 3;
      Exists {u2 | u2 <- enrolled(c);
                   age(u2) < 19} } }
```

The expression translates straightforwardly into nested AMOSQL thus:

```
SELECT name(u1)
FROM   undergrad u1
WHERE  some(
   SELECT c
   FROM   course c
   WHERE  c = takes(u1)
   AND    level(c) > 3
   AND    some(
      SELECT u2
      FROM   undergrad u2
      WHERE  u2 = enrolled(c)
      AND    age(u2) < 19
   )
);
```

However, the ZF expression form makes it easy to spot and remove a level of nesting thus:

```
{name(u1) | u1 <- undergrad;
  Exists {c | c <- takes(u1);
              level(c) > 3;
              u2 <- enrolled(c);
              age(u2) < 19} }
```

Here we make use of the fact that extra variables introduced by a generator in the body of a ZF expression are effectively existentially quantified, since if the generator produces no values or they are all filtered out by the following predicate, then the whole expression evaluates to an empty set. We can formalise this by a rewrite rule as follows:

```
Exists{x | x <- generator; P(x);
           Exists{y | y <- h(x); Q(x,y)}}  =
Exists{x | x <- generator; P(x);
           y <- h(x); Q(x,y)}
```

Here P and Q represent any boolean expression which involve x (or y). These rules are currently implemented directly in Prolog [25], since this has an ability to pattern-match such expressions in ICode, which is more general than that provided in our rewrite rule language described earlier.

The rewritten, semantically equivalent, AMOSQL query is:

```
SELECT name(u1)
FROM   undergrad u1
WHERE  some(
   SELECT c
   FROM   course c, undergrad u2
   WHERE  c = takes(u1)
   AND    level(c) > 3
   AND    u2 = enrolled(c)
   AND    age(u2) < 19
);
```

The real gain in execution time is due to the fact that the number of subqueries that must be evaluated is reduced. In the nested original query there will be $O(card(undergrad) \cdot card(course))$ subquery evaluations. Applying the same kind of reasoning on the rewritten query yields $card(undergrad)$ subquery evaluations, which means a reduction from quadratic to linear time.

Note that we depend on the AMOSQL optimiser to use the join predicates and selections concealed within P and Q to avoid a simplistic iteration over the Cartesian product of x(course) and y(undergrad), otherwise the rule could actually worsen performance for a very selective P. This shows the importance of knowledge about the remote query evaluation.

5 Related work

Related work in the bioinformatics field includes the TAMBIS system [21], which writes query plans in CPL (the Collection Programming Language) [1]. CPL is a comprehension based language in which the *generators* are calls to library functions that request data from specific databases according to specific criteria. Plans in TAMBIS are based on following a classification hierarchy, whereas our plans are oriented towerds *ad hoc* SQL3-like queries. However, the overall approach is similar in using a high level intermediate code translated through wrappers.

The Kleisli system [27] is a sophisticated system for querying and data integration over heterogeneous databases, with impressive applications in bioinformatics. It also uses the CPL comprehension language. It has recently been rewritten in standard ML, a functional language, and so uses functions to implement rewrite rules in the optimiser. These functions are currently built into the optimiser but are extensible by the implementors. They work directly on their internal monad composition form of queries, which plays a similar role to our ICode. By contrast our rule syntax endeavours to make rules both writeable and readable by end users (domain specialists), hiding the complexity of ICode. Our rules work within a unification paradigm, whereas theirs can be executed under various alternative control regimes. Their rules do have the advantage of being more generic than ours, but pay a price in readability, as discussed later.

They also use rewrite rules to generate SQL code, including joins and selections to be executed remotely, by successive transformations on a null SQL query. However, they do not give any rules for improving existential queries. Currently we generate a rewritten ICode query which is then translated directly into the target language inside a wrapper. Since some of our rules are specific to particular wrappers, corresponding to their ones customised to SQL, the approaches are effectively similar; only a closer comparison can tell which is more easily maintainable.

Methods to optimise the query processing of nested queries in relational databases, including queries with an existentially quantified subquery, have been thoroughly investigated. Kim suggested an unnesting method in 1982 [16]. Strictly speaking, this method dealt with nested queries in general and not specifically with quantified expressions. It was later improved by Ganski and Wong [5], who also showed how to rewrite some expressions for evaluation using aggregation functions such as the `COUNT` operator. Further improvements came from Muralikrishna [19] and Freytag [4].

In 1983, Jarke and Koch [10] described a set of transformations based on logical identities to evaluate quantified expressions more efficiently in limited memory. Although their work concentrated on an algorithm for efficiently evaluating nested quantified queries by successive operations on an intermediate relation, their mathematical transformations are useful nowadays in a distributed setting, as explained in section 3.3.

The 1997 paper by Claußen *et al.* [2] is most closely related to our work. It states *"Our discussion focuses on modern data models which use set-valued attributes to represent M:N-relationships — such as in the object-oriented model or the object relational model. In such a data model queries with universal quantification can usually be formulated in a much more natural way than in a flat relational model."* These remarks also apply to the FDM, since it is a modern semantic data model based on entities with subtypes, much like objects with inheritance. The examples in this paper use a Daplex syntax adapted from Shipman's original and bear out the readability of quantified queries expressed with set-valued attributes as functions. We should also remember that Dayal's original optimisation work was done with heterogeneous distributed databases on the Multibase project for which the FDM was designed.

Universally quantified expressions

The expressions that Claußen *et al.* [2] concentrate on closely resemble the perfect nested expressions of [10], but they restrict themselves to universal quantifiers as in

$$\{e1 \mid (\forall e2)p(e2) \Rightarrow q(e1, e2)\}$$
$$\{e1 \mid (\forall e2)p(e1, e2) \Rightarrow q(e1, e2)\}$$

They were able to show how to evaluate them efficiently for large relations by using alternative techniques including aggregation operations (as mentioned above), division and anti-semijoin. This would complement our approach, which is to use rewrite rules to adapt queries into a form where efficient processing techniques at a remote site can be brought to bear.

In this paper we have concentrated on existentially quantified expressions. However, many of the results can be adapted to use *all* or *any* quantifiers, simply by negating the *Exists* quantifier in ZF expressions.

$$\sim Exists\{c \mid c \leftarrow gen(u); \sim pred(c)\} =$$
$$(\forall c)(c \leftarrow gen(u)) \Rightarrow pred(c)$$

The pragmatic difference is that *Exists* may succeed quite quickly with only partial evaluation of the set expression (especially in Prolog), but looking for the absence of a counterexample will usually need exhaustive enumeration. Nevertheless, the principle of sending whole expressions for remote evaluation and avoiding *penny packet* evaluation continues to be important, and it can be extended to universal as well as existential queries by this technique.

6 Conclusions

We have discussed the role of rewrite rules in transforming quantified queries for execution on heterogeneous databases. We advocate the use of an object-oriented data model with a high level language including quantifiers that does not tie us either to relational or object storage. This was indeed the motivation of the original Multibase project [18], which was way ahead of its time.

Given such a model, in which we can express queries in a referentially transparent way, we can then apply sets of rewrite rules that adapt the query to the characteristics of the remote database. These rules depend on substitution of equivalent expressions and unification of variables, and do not involve bottom-up production rules with side-effects. Thus we believe this approach to be extensible and scaleable, and we are testing it in a mediator in use for a bioinformatics application, which has so far worked with simpler queries [15].

We have shown how the rewrite rules provide a great deal of flexibility. They can implement the logical rules given by Jarke and Koch [10]. They can implement many forms of rewrites based on data semantics as in King [17]. They can spot opportunities to replace iteration by indexed search, possibly of a materialised view. They can implement flattening and unnesting transformations that save wasting time compiling subqueries in AMOSQL. We believe they can be similarly adapted to features of other DBMS. Most importantly, we can perform rewrites that change the relative workload between two processors in a distributed query. Finally, we can combine all these rewrites, since some of them will enable others to take place. Thus we can deal with many combinations without having to foresee them and code them individually.

We have introduced a simple but powerful rewrite language that includes FOL quantifiers and allows a very general form of parametrisation, which suits unification in Prolog. There is, admittedly, a trade-off between readability and degree of abstract parametrisation. Rules that refer to very domain-specific situations involving specific named attributes are much easier to read and understand, which is what we want. One way to extend this could be to include carefully formatted specific instances of abstract rules as comments, for ease of understanding. Another problem is that users could make rules more general than intended by not putting enough checks in the where-clause of a rule. This is a direction for future work involving cross-checks with metadata.

We have concentrated on quantified queries because they usually involve a lot of iteration, possibly on a remote database. Rewrite rules enable us to spot cases where the iteration can be all be done on one database, or replaced by faster indexed searches, with greatly improved performance. They also allow us to have rewrites in more than one phase, with a final phase for rules specific to a target DBMS. This leads us to a uniform optimisation framework within which we can cope with an expanding network of remote data sources with different characteristics. This is essential in an Internet environment, where data sources continue to appear, using different data management systems [7].

Acknowledgements

Part of is work was supported by a grant from the BBSRC/EPSRC Joint Programme in Bioinformatics (Grant Ref. 1/BIF06716). The work of Andreas Sjöstedt was supported by a travel grant under the Socrates Program of EU. We are grateful to Prof. Töre Risch and team at Uppsala University for help and advice on interfacing to AMOS II.

References

[1] P. Buneman, S.B. Davidson, K. Hart, G.C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, pages 158–169. Morgan Kaufmann, 1995.

[2] J. Claußen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 286–295. Morgan Kaufmann, 1997.

[3] S. M. Embury and P. M. D. Gray. A modular compiler architecture for a data manipulation language. In *Advances in Databases, 14th British National Conference on Databases, BNCOD 14, Edinburgh, UK, July 3-5, 1996, Proceedings*, volume 1094, pages 170–188. Springer-Verlag, 1996.

[4] J.C. Freytag. A Rule-Based View of Query Optimization. In U. Dayal and I. Traiger, editors, *SIGMOD 87 Conference*, pages 173–180, San Francisco, May 1987. ACM Press.

[5] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 23–33. ACM Press, 1987.

[6] P.M.D. Gray, S.M. Embury, K. Hui, and G.J.L. Kemp. The Evolving Role of Constraints in the Functional Data Model. *J. Intelligent Information Systems*, 12:113–137, 1999.

[7] P.M.D. Gray, P.J.H. King, and L. Kerschberg. Guest Editor Introduction: Functional Approach to Intelligent Information Systems. *Journal of Intelligent Information Systems*, 12:107–111, 1999.

[8] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton. *Object-Oriented Databases: a Semantic Data Model Approach.* Prentice Hall Series in Computer Science. Prentice Hall International Ltd., 1992.

[9] P.M.D. Gray, D.S. Moffat, and N.W. Paton. A Prolog Interface to a Functional Data Model Database. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Extending Database Technology Conference*, pages 34–48. Springer-Verlag, 1988.

[10] M. Jarke and J. Koch. Range nesting: A fast method to evaluate quantified queries. In D. J. DeWitt and G. Gardarin, editors, *SIGMOD'83, Proceedings of Annual Meeting, San Jose, California, May 23-26, 1983*, pages 196–206. ACM Press, 1983.

[11] Z. Jiao. *Optimisation Studies in a Prolog Object-Oriented Database.* PhD thesis, University of Aberdeen, Department of Computing Science, November 1992.

[12] Z. Jiao and P.M.D. Gray. Optimisation of Methods in a Navigational Query Language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Second International Conference on Deductive and Object-Oriented Databases*, pages 22–42, Munich, December 1991. Springer-Verlag.

[13] G. J. L. Kemp, J. Dupont, and P. M. D. Gray. Using the Functional Data Model to Integrate Distributed Biological Data Sources. In P. Svensson and J.C. French, editors, *Proceedings Eighth International Conference on Scientific and Statistical Database Management*, pages 176–185. IEEE Computer Society Press, 1996.

[14] G.J.L. Kemp, N. Angelopoulos, and P.M.D. Gray. A Schema-based Approach to Building a Bioinformatics Database Federation. In *Proceedings IEEE International Symposium on Bio-Informatics and Biomedical Engineering*, pages 13–20. IEEE Computer Society Press, 2000.

[15] G.J.L. Kemp, C.J. Robertson, P.M.D. Gray, and N. Angelopoulos. CORBA and XML: Design Choices for Database Federations. In B. Lings and K. Jeffery, editors, *Advances in Databases: Proceedings of 17th British National Conference on Databases (LNCS 1832)*, pages 191–208. Springer Verlag, 2000.

[16] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7:443–469, 1982.

[17] J.J. King. *Query Optimisation by Semantic Reasoning.* UMI Research Press, 1984.

[18] T. Landers and R. L. Rosenberg. An Overview of MULTIBASE. In H.-J. Schneider, editor, *Distributed Data Bases.* North-Holland Publishing Company, 1982.

[19] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In L. Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 91–102. Morgan Kaufmann, 1992.

[20] N.W. Paton and P.M.D. Gray. Optimising and Executing Daplex Queries Using Prolog. *The Computer Journal*, 33:547–555, 1990.

[21] N.W. Paton, R. Stevens, P. Baker, C.A. Goble, S. Bechhofer, and A. Brass. Query Processing in the TAMBIS Bioinformatics Source Integration System. In *11th International Conference on Scientific and Statistical Database Management, Proceedings*, pages 138–147. IEEE Computer Society Press, 1999.

[22] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992*, pages 39–48. ACM Press, 1992.

[23] T. Risch, V. Josifovski, and T. Katchaounov. AMOS II Concepts. Available at http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html, November 1999.

[24] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[25] Sjöstedt, A.R. *Using AMOS II as Data Source for P/FDM and Rewriting Quantified AMOSQL Queries.* Project Report, Department of Computing Science, University of Aberdeen, 2000.

[26] D.A. Turner. Miranda: a Non-Strict Functional Language with Polymorphic Types. In J.-P. Jouannaud, editor, *Proceedings of the IFIP Int. Conf. on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computing Science, Vol. 201, pages 1–16, Nancy, France, September 1985. Springer-Verlag.

[27] L. Wong. Kleisli, a functional query system. *J. Funct. Prog.*, 10:19–56, 2000.