Elimination of Views and Redundant Variables in an SQL-like Database Language for Extended NF² Structures

Norbert Südkamp, Volker Linnemann
IBM Scientific Center Heidelberg, Tiergartenstr. 15
D-6900 Heidelberg, West Germany
Tel: (+49 6221) 4040
e-mail: SUEDKAMP at DHDIBM1.BITNET, LINNEMAN at DHDIBM1.BITNET

Abstract

The Advanced Information Management Prototype (AIM) is an experimental database system, developed and prototyped at the IBM Scientific Center in Heidelberg, Germany. The underlying data model is an extension of the NF^2 data model. It is founded on the notions of tuple, set and list. These three constructors may be applied to any valid structure in any order, starting with some atomic domain(s), to get a valid AIM database structure. The corresponding database language, called HDBL (Heidelberg Data Base Language) is an SQL-type language meeting the requirements of the extended NF^2 data model.

In this paper we investigate the problem of improving the evaluation of HDBL queries by transformation of the query. The first kind of transformation deals with view processing. Given an HDBL query containing a view it is often not necessary to materialize the view. We will give some rules on how to eliminate the view and evaluate the query against the base relations. Another kind of transformation will be used to remove variables from the query. Some of these transformations are equivalent to the removal of redundant join operations in the relational algebra; a second way to remove variables is the introduction of "complex projections" which can directly be mapped to operations at the storage access interface of the DBMS.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

The Advanced Information Management Prototype (AIM) is an experimental database system which has been developed and prototyped by the IBM Scientific Center in Heidelberg. This system is based on an extended NF^2 (Non First Normal Form) data model [5]. The motivation to build such a system was the need to store and manage huge amounts of data in areas as office, science and engineering. The requirements for such database management systems are rather different from the requirements in business and administrative applications. There the concept of a table, i.e. a relation is a reasonable data structure to model all necessary data types. To model other structures, such as highly structured documents or geometrical information coming from CAD workstations, the relational model is rather cumbersome to use and becomes inadequate and also inefficient.

The development of the NF^2 data model [2; 23] was an attempt to gain more flexibility in modelling adequate data structures for complex problems while retaining the advantages of the relational model, such as a sound mathematical foundation and a nonprocedural query language.

DBMSs such as AIM, also called Non Standard DBMSs, are intended to cover a wide range of applications. In one of our application projects AIM was used as an integration vehicle for a set of stand alone applications in the area of robot modelling, programming and simulation. Geometrical robot models had to be managed, description of handling tasks, sequences of operations, or data to determine the trajectories of a robot arm [6; 7]. To be an adequate tool for integration, such a system not only has to provide powerful modeling capabilities but it also has to support appropriate user interfaces. Different tasks need different data structures and different operators dedicated to the specific problems to be handled. To meet this requirement the AIM prototype provides user defined data types and user defined operators to be added dynamically to the system [17]. Besides the AIM project there are several other research projects which approach these problems by other techniques, see for example [3; 4; 10; 13; 22].

Another feature which is helpful in supporting several interfaces is the introduction of views. Views are already widely used in relational systems. They allow to present all or part of the data in a structure adapted to and better suited for a given application than the original structure. In this paper we are interested in the processing of queries in such an environment. Users formulate queries and operations against views, which in turn have to be evaluated against the stored database. Often it is not a good solution to materialize all the views first and then use them as intermediate results to produce the final result. A straightforward strategy would be to rewrite the query by inserting the view definitions. Because there may be several "layers" of views, i.e. a view may be defined using another view and so on, this process may be performed several times. It is obvious that the processing of such a modified query may become very inefficient if it is evaluated directly against the stored database. It is the task of the query optimizer to produce a reasonable evaluation strategy [16]. One important aspect of query optimization is to detect and to remove redundant operations, i.e. the query has to be rewritten again before the optimizer selects suitable access paths and so on.

There has been some work on detection and elimination of redundant operations, mostly done in the context of the relational data model [19; 26] . Some recent work on rewriting of SQL-queries has been presented in [14] where the authors used a graphical representation of the query as a basis for their transformation rules. In [24] an evaluation strategy is developed for queries against a relational interface, implemented via an NF^2 kernel. If certain restrictions concerning the mapping are obeyed the elimination of redundant joins can be performed by applying the original relational methods.

Query transformations can also be defined for expressions of nested relation algebras. They are based on the properties of the operators of an algebra as given for example in [23] or in [21]. These transformations are restricted if nesting and unnesting are involved. A recent paper [15] investigates these restrictions in detail and the author suggests in his conclusion that one should also look at these optimization problems from a calculus point of view.

Our approach is to take the calculus oriented query language HDBL (Heidelberg Data Base Language) of our data model as a basis for query transformations. In this paper we will present techniques to eliminate views and to detect and remove redundant operations from a query in the context of our extended NF^2 data model. Applying the rules for view elimination allows, for example, to avoid some unnecessary intermediate results which are not needed for the production of the final query result.

We will identify situations where variables can be removed from a HDBL query. Each eliminated variable means to avoid the unnecessary access to the range of this variable. So the number of variables appearing in a query provides a criterion for a "better" query.

Another type of query rewriting is the introduction of complex operators. To process a query the query processor analyzes the query string and produces a query evaluation plan which can be interpreted by the evaluation program. The access to the database in such an evaluation plan is specified by calls to the storage access system. The AIM access system is structurally object oriented which means that a complex object or subobject may be transferred to the evaluation system by one call to the access system. We will give some transformation rules to introduce so called complex projections, which will be mapped directly to calls of the access system. Introducing such operators avoids the transfer of atomic data from the database and the reconstruction of the complex object needed for the query evaluation.

The rest of this paper is organized as follows. We start with an introduction to the data structures of AIM and the query language HDBL. Then we will present an example showing the scope of our query rewriting and transformation rules. These rules will be given in detail in the following chapters. We will conclude by an overview of open problems and some future directions.

2 The AIM data model

The data model of the AIM prototype (Advanced Information Management System) is an extension of the NF^2 data model which itself is an extension of the relational data model. We will give a brief introduction to the data structures of the AIM data model and the query language HDBL being the basis for our investigations on query transformations.

Data Structures

Starting with atomic types integer, real, string, char, ... any of the constructors tuple, set or list may be used to generate new types.

Some examples of types are

```
INTEGER
SET(REAL)
LIST(TUPLE(a:REAL, b:STRING))   .
```

A database schema is a set of named types like

```
AUTH=SET(TUPLE(authors:LIST(TUPLE(name:STRING)),
               rep_no :STRING,
               title  :TEXT,
               descr  :SET(TUPLE(keyword:STRING,
                                 weight  INTEGER)))).
```

This schema describes a structure for the information about reports and their authors, including a set of keywords and a number, indicating the weight of each of the keywords.

As stated in the introductory remarks this data model is an extension of the relational data model. Figure 1 presents the different data structure capabilities of the relational model, the NF² data model and of the AIM data model. The relational model only allows sets of tuples, whose attributes must have an atomic domain. The types allowed in the NF² model are sets of tuples whose components may again be sets of tuples and so on, whereas the AIM data model allows any sequence of constructors.

Query Language

The AIM query language HDBL (Heidelberg Data Base Language) and is an extension of SQL, a query language for the relational data model [9] .

We give examples of HDBL queries. They refer to the database schema containing the type AUTH, given in a previous example.

```
(1) select tuple(r: x.rep_no,
                 p: (select(tuple(keyw: y.keyword,
                                  wght:  y.weight))
                     from y in x.descr)
                     )
    from   x in auth

(2) select x.authors
    from   x in auth

(3) select tuple(r: x.rep_no, n: y.name)
    from   y in x.authors, x in auth
    where  exists (v in x.descr):
           (v.keyword contains 'database'
            and v.weight > 50)
```

In [25] a formal definition of the syntax and also the semantics of HDBL is given. For our purposes it is sufficient to have in mind the following model for the evaluation of a select-from-where expression:

> Each variable in the from-list is bound to a domain, i.e. a variable may take all the values of its domain
>
> For each possible combination of values taken from the respective domains evaluate the predicate in the where-clause. The variables from the from-list are the only free variables in the predicate. Therefore, replacing the variables by their respective values allows to evaluate the predicate.
>
> If this evaluation of the predicate yields true then apply the select-clause to the values of the variables, i.e. construct an element of the result collection. This construction may again include the evaluation of a (nested) select-from-where expression.

A view is a named HDBL query. All the queries given as an example above may be regarded as views.

3 An Example

In this section we present a detailed example to demonstrate the scope and the benefits of query transformations. As a database schema we use the definition of DEPARTMENTS, a database object containing information about departments and their managers together with a list of all employees working for a department. We define two views. V1 gives the number of the department and the list of the employee numbers of all employees of this department, view V2 gives all the information about the employees of all depart-

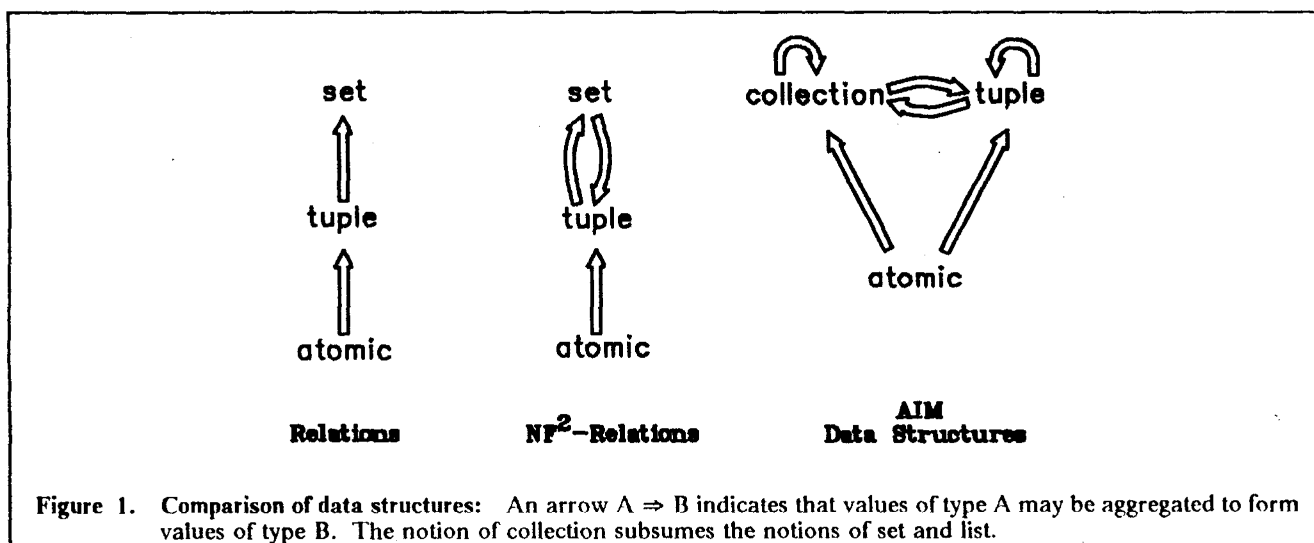


Figure 1. Comparison of data structures: An arrow A ⇒ B indicates that values of type A may be aggregated to form values of type B. The notion of collection subsumes the notions of set and list.

ments. We assume that the employee number e_no is unique within DEPARTMENTS.

The query 'Find all employees of a department, grouped by the department number' can be formulated as follows:

```
(Q1) select tuple(d: x.dep,
                  e:(select tuple(no: y.eno,
                                  nm: z.nme)
                     from z in V2, y in x.emp
                     where y.eno = z.eno))
     from x in V1
```

Another query which gives the same result in our example is the following HDBL-expression.

```
(Q2) select tuple(d: x.dep,
                  e:(select tuple(no: z.eno,
                                  nm: z.nme)
                     from z in V2
                     where exists (y in x.emp):
                           y.eno = z.eno))
     from x in V1
```

If the two views V1 and V2 are materialized these formulations could be a reasonable basis for the query evaluation. If they are not materialized the terms V1 and V2 have to be replaced by their corresponding view definitions. This replacement yields a valid HDBL expression. A straightforward evaluation of such a query would require the construction of two intermediate results. It is possible to avoid these intermediate results and build the result of the query directly out of the basic database objects as will be shown below.

If we take a look at the database object called departments we could formulate the above query against this object yielding the following query:[1]

```
(Q1') select tuple(d: x.dep_no,
                   e: x.empl)
      from   x in departments
```

In the following we are interested in general rules which can perform this kind of query rewriting. Given a query like (Q1) or (Q2) we want to apply some rewriting rules and come up with query (Q1'). The resulting query does not need any construction of intermediate results and uses the structural capabilities of the data model and of the prototype implementation. In the following we will present several types of transformation rules. They will be applied to our example and we eventually will produce the desired form of the query.

4 Query Modification to Eliminate Views

First we will give general rules to eliminate views from a query. To be precise we will present transformation rules which allow to replace ***select-from-where*** ex-

Schema:

```
DEPARTMENTS = SET(TUPLE(dep_no :integer,
                        dep_mgr:string,
                        empl: LIST(TUPLE(e_no:integer,
                                         name:string
                                   )   )
                  )   )
```

DEPARTMENTS

dep_no	dep_mgr	empl	
		e_no	name

View definitions:

```
V1 = select tuple(dep: x.dep_no,
                  emp: (select tuple(eno: y.e_no)
                        from   y in x.empl) )
     from   x in departments
```

View V1

dep	emp
	eno

```
V2 = select tuple(eno: y.e_no,
                  nme: y.name)
     from   x in departments, y in x.empl
```

View V2

eno	nme

Schema and View definitions

[1] In fact, there is a slight difference between the result of query Q1 and Q1': Query Q1 specifies two new attribute names no and nm whereas Q1' takes the attribute names e_no and name from the database object DEPARTMENTS

pressions (sfw-expressions for short) appearing in a ***from-list*** or in the range definition of a quantified variable. Both cases are a notation for the explicit construction of an intermediate result which, in many cases, is not necessary to evaluate the query. On the other hand we will discover that such a transformation may provide a basis for further transformations to speed up query processing.

Inserting the definition of view V1 into query (Q1)and renaming the variables to avoid duplicate variable names, we get the following query:

```
(Q1) select tuple(d: x.dep,
                  e:(select tuple(no: y.eno,
                                  nm: z.nme)
                     from z in V2, y in x.emp
                     where y.eno = z.eno))
     from x in
        (select tuple(dep:x'.dep_no,
                      emp:(select tuple(eno:y'.e_no)
                           from   y' in x'.empl) )
         from   x' in departments )
```

We highlighted all parts which are related to the view V1. The range of the variable x is the set of tuples, specified by the highlighted view definition. The idea to eliminate this view is to replace each occurrence of x by an expression related to the objects from the from_list of the view definition. In our example we will replace x.dep by the definition of the attribute dep in view V1 which is x'.dep_no. The term x.emp in the from-list is replaced by the sfw-expression

```
(select tuple(eno: y'.e_no)
 from   y' in x'.empl)
```

and the view definition is replaced by the from-list of the view definition. The result of these transformations is

```
(Q1.1)
select tuple(d:x'.dep_no,
          e:(select tuple(no: y.eno,
                          nm: z.nme)
             from z in V2,
                  y in (select tuple(eno: y'.e_no)
                        from   y' in x'.emp)
             where y.eno = z.eno))
from x' in departments
```

Here we have introduced a sfw-expression as part of the from-list of a nested sfw-expression. This expression can be replaced by applying the same procedure again yielding the expression

```
(Q1.2) select tuple(d:x'.dep_no,
                  e:(select tuple(no: y'.e_no,
                                  nm: z.nme)
                     from z in V2, y' in x'.emp
                     where y'.e_no = z.eno))
       from x' in departments
```

Elimination of view V2 then yields:

```
(Q1.3) select tuple(d:x'.dep_no,
                  e:(select tuple(no: y'.e_no,
                                  nm: y".name)
                     from x" in departments,
                       y" in x".empl, y' in x'.emp
                     where y'.e_no = y".e_no))
       from x' in departments
```

This query does not contain any views or any sfw-expression in the from-list and may be evaluated against the database object departments. This query will be simplified by applying rules to be introduced in chapter 5.

The general form of a sfw-query containing a second sfw-query in the from-list is the following:

```
SFW1 = select t_expr1
       from   f_list1, x in (select t_expr2
                             from   f_list2
                             where  pred2), f_list1'
       where  pred1
```

We assume that there will be no declaration of two variables with the same name in the query. Therefore the variable x may appear in the terms t-expr1, f_list1, f_list1' and pred1 but not in f_list2. We will replace every occurrence of the variable x by a term derived from t_expr2.

The sfw-expression SFW1 is equivalent to the sfw-expression SFW2:

```
SFW2 = select t_expr1[ x/t_expr2 ]
       from   f_list1[ x/t_expr2 ], f_list2,
              f_list1'[ x/t_expr2 ]
       where  pred1[ x/t_expr2 ] ∧ pred2
```

Here t[x/t_expr2] denotes a substitution. All occurrences x.a are replaced by the term exp, where exp is a definition of the attribute a within t_expr2, i.e. t_expr2 contains a subterm of the form ...a: exp,... , and all other occurrences of the variable x in term t are replaced by t_expr2.

This transformation can be applied repeatedly if necessary. Applying our rule to query (Q2) as far as possible will give us

```
(Q2.1)
select tuple
       (d:x'.dep_no,
        e:(select tuple(no:y".e_no,
                        nm:y".name)
           from x" in departments, y" in x".empl
           where exists (y' in
                    (select tuple(eno:y'''.e_no
                     from y''' in x'.empl))):
                  ( y'.eno = y".e_no)))
from x' in departments
```

Here we introduced a sfw-expression as the range definition of the existentially quantified variable y'. This can't be removed by our rules so far. But again we are able to transform the existence predicate, based on the view definition into an existence predicate using only variables defined in the from list of the view definition.

The general form of a quantified expression appearing in a where-clause is as follows:

```
SFW3 = select t_exprl
       from   f_listl
       where  predl θ
              exists (x in (select t_expr2
                            from   x_1 in range_1,
                              ..., x_n in range_n
                            where  pred2)) : pred_x
              θ' pred3
```

where $range_i$ is an expression of type collection, i.e. a set or list, $\theta, \theta' \in \{ \wedge, \vee \}$. Again we assume that there are no duplicate variable names. Therefore x only appears in the predicate $pred_x$.

SFW3 is equivalent to SFW4:

```
SFW4 = select t_exprl
       from   f_listl
       where  predl θ
              exists (x_1 in range_1)
              ...exists (x_n in range_n):
              (pred_x[ x/t_expr2 ] ∧ pred2)
              θ' pred3
```

Using this rule to eliminate the sfw-expression from the where-clause in our example (Q2.1) we end up with the following:

```
(Q2.2)
select tuple(d:x'.dep_no,
           e:(select tuple(no:y".e_no,
                           nm:y".name)
              from x" in departments,y" in x".empl
              where exists ( y' in x'.empl):
                      ( y'.eno = y".e_no)))
from x' in departments
```

There is an analogous rule for universally quantified variables over sfw_expressions. These rules allow the elimination of sfw-expressions from where-clauses and from-lists. Based on a formal semantics of HDBL we are able to prove that these rules are sound.[2] This proof can be found in [25] .

These transformations justify the notion of standard sfw-expressions, being expressions without sfw-expressions in the from-list or in the where-clause. In the following we will always refer to sfw-expressions in this standard form if we talk about sfw-expressions.

5 Redundant Variables

Our examples in the previous chapter show that rather strange query formulations may occur if views are involved or, even worse, if queries are generated automatically. Let's take again query (Q1.3) as an example:

```
select tuple(d:x'.dep_no,
           e:(select tuple(no: y'.e_no,
                           nm: y".name)
              from x" in departments,
                  y' in x'.emp, y" in x".empl
              where y'.e_no = y".e_no)
from x' in departments
```

We have four variables, each variable requires the access to the respective domain. There are hierarchical relationships between these variables and x′ and x″ as well as y′ and y″ refer to the same domain.

Though it is very unlikely for a user to produce this query directly, the data base system has to deal with it in a reasonable way because the query may have been automatically generated as discussed in sections 3 and 4. Assuming that the attribute e_no is an identifier for a whole department object, i.e. an e_no is unique within all departments, we can deduct that variables x″ and y″ are redundant and may be eliminated, i.e. substituted by the variables x′ and y′.

A similar problem arises in the context of the relational data model. In [19] the authors give algorithms to detect and to remove redundant joins. Their treatment is formally based on the relational algebra and the theory of functional dependencies. As our data model is an extension of the relational model it seems natural to extend the methods proposed in [19], using an adequately extended algebra. Unfortunately, the algebras developed for nested relations, e.g. [23] or [21]are not powerful enough to cover the complete data model we use. In addition, recent investigations on algebraic optimization of nested relations [15] reveal difficult problems if nesting and unnesting operations are involved in the transformation of algebra expressions.

In this chapter we will give some rules to detect and to eliminate redundant variables in the context of the AIM data model. The basis of our investigation is the calculus oriented language HDBL. Explicit variables are declared which specify the access to a certain domain, i.e. a set or a list of database objects which may be a sub-structure of a complex structure. A variable is redundant if there is a second variable, bound to the same domain, such that both variables always have the same value if the filtering predicate becomes true. This property depends on a given database state. Of course we are interested in some state independent criteria for redundancy of variables. Therefore we need something like key properties in the relational model. Such key properties or derived functional dependencies are used as a criterion in [19] .

First we introduce some more notations. Q and R will denote sfw-expressions in standard form, R is nested

[2] This proof only holds if we restrict ourselves to queries where there is no application of the built-in function 'position' to non base objects, i.e. views.

in Q if R appears in the select-clause of Q. The variable x is declared in a from-list if there is an expression *x in exp* in the from-list and from_var(Q) denotes the set of all variables declared in the from-list of Q. The set of variables valid in Q is denoted by var(Q).

var(Q) = from_var(Q) if Q is not nested within another sfw_expression

var(Q) = from_var(Q) ∪ var(R) if Q is nested within R.

The declarations of all variables in var(Q) define a set of hierarchies of variables. If *x in y.exp* is the declaration of x, x depends on y. The root of a hierarchy is a variable bound to a database object, i.e. its declaration is of the form x IN dbobject. In the following we assume that there are no variables with the same name, all hierarchies are rooted to database objects and all variables are declared.

Given a variable $x \in var(Q)$ we associate a path to x by the following definition:

path(x) = dbo if x in dbo is the declaration of x

path(x) = path(y)$ if x in y is the declaration of x

path(x) = path(y)$.att if x in y.att is the declaration of x.

Two variables x and y are compatible if path(x) = path(y).

Referring to the example query (Q1.3) the variables y′ and y″ are compatible because path(y′) = path(y″) = departments$.empl. In the relational context variables may be redundant if they are bound to the same relation. If we restrict our data model to flat relations, the path of any variable is the name of a relation. Then two variables are compatible if they refer to the same relation.

Besides compatibility we need something similar to the key property in the relational data model. Intuitively, if x and y are compatible variables with respect to a sfw-expression Q, one of them is redundant if the predicate of the sfw-expression evaluates to true only if x and y are replaced by the same object. To formalize this concept we introduce an identification function which is a generalization of the key concept.

The type T_n is a subtype of T_1 if there is a sequence $T_2,...,T_{n-1}$, such that T_i is the type of a component of type T_{i-1} (i.e. T_{i-1} is a tuple type) or T_i is the type of the elements of instances of type T_{i-1} (i.e. T_{i-1} is a set- or list type), i = 2,...,n.

If x_n is an instance of type T_n then there is exactly one instance x_i of type T_i, i = 1,...,n-1, such that x_i is a component of x_{i-1} (i.e. x_{i-1} is a tuple) or x_i is an element of x_{i-1} (i.e. x_{i-1} is a set or a list).

The function f is an identification function w.r.t. T_1 if for all instances x_n, y_n of type T_n the following holds:

$$f(x_n) = f(y_n) \Rightarrow x_n =_{id} y_n \wedge ... \wedge x_1 =_{id} y_1 .$$

The relation $=_{id}$ denotes the identity of objects. This is different from objects having identical values, for example if we deal with lists. The first and the third element of the list < 1,2,1,3 > for example, both have the same value but they are different objects.

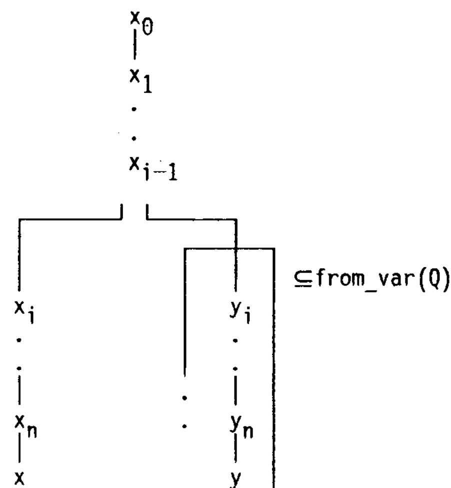
If $a_1,...,a_k$ are defined to be a key for a relation, this combination of attributes may serve as an identification function w.r.t. this relation. Of course, the system has to know about identification functions. These functions are listed in the database catalog or in a data dictionary, as well as some rules to deduct that a given part of a predicate is an identification function, like the rules for the deduction of functional dependencies from a given set of dependencies in the relational data model.

Referring to our example query (Q1.3) the attribute e_no serves as an identification function with respect to departments if no e_no appears twice in a departments object.

Now we are ready to define some criteria to decide whether a variable is redundant. Let Q be an sfw-expression which may be nested in an sfw-expression R.

```
R = ...
      (select t_expr
       from   f_list
       where  f(x) = f(y) ∧ pred)
    ....
    ....
```

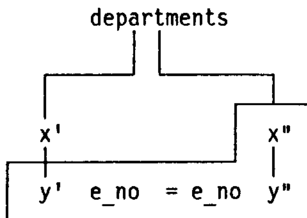
Let's assume that the hierarchical relationships between the variables being valid in Q are as given in the following diagram, where x_j depends on x_k if there is an edge from x_k to x_j



x_0 represents the database object dbo to which x_1 is bound, i.e. x_0 is a dummy variable whose value is always the database object dbo.

If x and y are compatible and if f is an identification function w.r.t. the range of variable x_{i-1}, then $y_i,\ldots, y_n$, y are redundant.

The relationship among the variables of (Q1.3) may be depicted as follows (the box contains the variables which are declared in the nested sfw-expression):



Assuming that the attribute e_no serves as an identification function, the variables x″ and y″ are redundant.

If there are redundant variables there is an equivalent HDBL expression without these variables. The transformation rules to produce this equivalent query are given in the following. In general these transformations are performed by substitution of redundant variables by compatible ones. We use the same notation for the substitution as in chapter 4. If t denotes a term then t[y/x] denotes a term with each occurrence of the symbol y replaced by the symbol x.

We have to distinguish between two different cases, depending on the from-list f_list of Q.

1. f_list contains a non-redundant variable, i.e.

   $\text{from_var}(Q) \setminus \{y_i,\ldots,y_n,y\} \neq \phi$

   The following HDBL-expression R′ is equivalent to R:

```
R'= ...
       (select t_expr[y_i/x_i,...,y_n/x_n,y/x]
        from   red_f_list[y_i/x_i,...,y_n/x_n,y/x]
        where  pred[y_i/x_i,...,y_n/x_n,y/x])
       ....
       ....
```

   red_f_list denotes a from-list where the declarations of all redundant variables have been removed from f_list.

2. f_list contains only declarations of redundant variables, i.e.

   $\text{from_var}(Q) \setminus \{y_i,\ldots,y_n,y\} = \phi$

   Then the following expression R′ is equivalent to R

```
R'= ...
           if     pred[y_i/x_i,...,y_n/x_n,y/x]
           then
              set t_expr[y_i/x_i,...,y_n/x_n,y/x]
           else
              empty
       ....
       ....
```

   empty denotes the empty set and the expression set $t_expr[y_i/x_i,\ldots,y_n/x_n,y/x]$ denotes a constant set with one element.[3]

Removing the redundant variables x″ and y″ from (Q1.3) results in the following query:

```
(Q1.4) select tuple(d: x.dep_no,
                    e:(select tuple(no: y.e_no,
                                    nm: y.name)
                       from y in x.empl))
       from x in departments
```

Substitution of redundant variables y by a compatible variable x in the predicate pred may produce subterms like x.att = x.att or x.att > x.att. These subterms are replaced by true resp. false, i.e all subterms x.att θ x.att are replaced by true if $\theta \in \{=, \leq, \geq\}$ and all subterms x.att θ x.att are replaced by false for $\theta \in \{\neq, >, <\}$. The result of this replacement may be further reduced by applying well known rules of predicate logic [16].

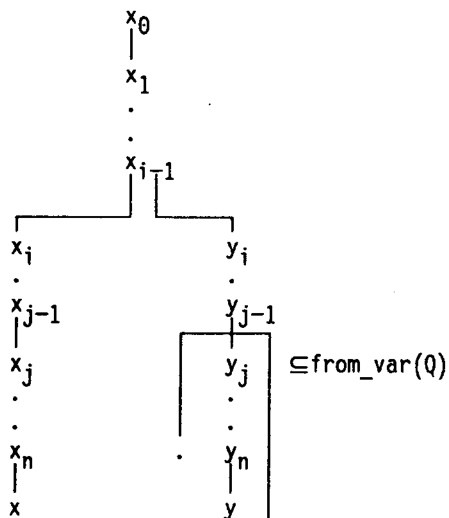
If the predicate $pred[y_i/x_i,\ldots,y_n/x_n,y/x]$ can be reduced to true the sfw-expression Q may be replaced by the constant expression set $t_expr[y_i/x_i,\ldots,y_n/x_n,y/x]$

If the predicate $pred[y_i/x_i,\ldots,y_n/x_n,y/x]$ can be reduced to false the sfw-expression Q may be replaced by the constant expression empty denoting the empty set.

The following definition gives a generalization of the above definition of redundant variables. Up to now we assumed that all variables $y_i,\ldots,y_n$ are declared in f_list, i.e. the local from-list of the sfw-expression Q. Now we allow $y_i,\ldots,y_{j-1}$ to be declared outside of f_list and $y_j,\ldots,y_n$, y are elements of from_var(Q). This situation is given in the following diagram:

---

[3] Here we use an if then else construction which is not available in our present HDBL version. Here it is used to indicate the strategy of the query evaluation program



Besides an identification function we now need a condition which ensures that the restrictions, implied by the predicates over $y_i,...,y_{j-1}$ are not stronger than the restrictions given by the predicates over $x_i,...,x_{j-1}$. Assuming that all predicates are given in a conjunctive normal form we collect all disjunctions containing an occurrence of a variable w appearing in a from-list outside of the sfw-expression Q and denote the conjunction of these disjunctions by $pr_Q(w)$. Using this notation we can express the above condition by

$$pr_Q(y_i) \wedge ... \wedge pr_Q(y_{j-1}) \Rightarrow$$
$$pr_Q(x_i)[x_i/y_i] \wedge .. \wedge pr_Q(x_{j-1})[x_{j-1}/y_{j-1}]$$

In general we cannot decide whether this condition holds or not without evaluating the predicate against the database state. But there are important cases where this condition holds. e.g. if there are no predicates referring to variables y_i at all outside the sfw-expression Q.

If this condition holds and f is an identification function w.r.t. the range of variable x_{i-1}, resp. dbo if i = 1, the variables $y_j,...,y_n$, y are redundant variables and may be substituted by their corresponding compatible variables $x_j,...,x_n$,x using the same transformation rules given above.

Another class of redundant variables not covered so far may occur in connection with existentially quantified variables. For an example see query (Q2.2) given in chapter 4.

Let's again look at our sfw-expression Q, possibly nested in an HDBL-expression R.

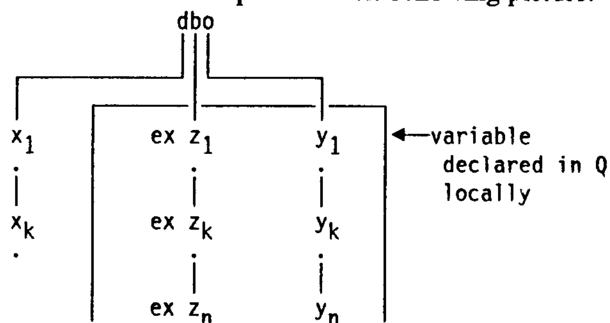
```
R = ...
      (select t_expr
       from   f_list
       where  pred)
    ....
    ....
```

We assume the following:

1. $\{y_1,...,y_n\} \subseteq$ from_var(Q) such that y_{j+1} depends on y_j, j = 1,...,n-1. and y_1 is bound to a database object dbo
2. pred contains existentially quantified variables $z_1,...,z_n$ such that z_j is compatible to y_j, i.e. pred looks like

   exists (z_1 in dbo) ... exists (z_n in z_{n-1} ...): pred′
3. There are variables $\{x_1,...,x_k\} \subset$ var(Q), $k \leq n$, such that x_j is compatible to y_j, j = 1,...,k-1.

This situation corresponds to the following picture:



If pred′ looks like

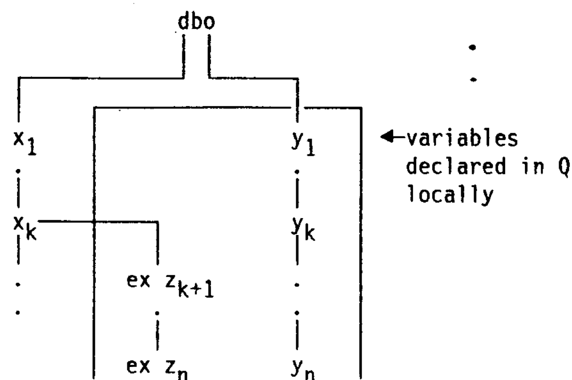
$$f1(x_k) = f1(z_k) \wedge f2(z_n) = f2(y_n) \wedge pred''$$

and pred″ does not contain further restrictions over variables z_j, j = 1,...,n, and if f1 and f2 are both identification functions w.r.t. dbo, then the variables $y_1,...,y_k$ are redundant and may be substituted by compatible variables $x_1,...,x_k$.

A slightly different situation occurs if we change the assumption no. 2 above to

> pred contains existentially quantified variables $z_{i+1},...,z_n$, such that z_j is compatible to y_j and z_{i+1} is declared within x_i, i.e. pred looks like
>
> exists (z_{i+1} in x_i...) ... exists (z_n in z_{n-1} ...): pred′

which can be displayed by the following figure:

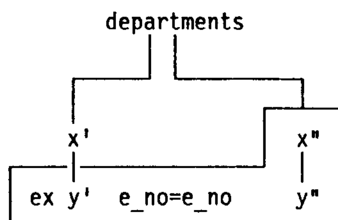


If pred′ looks like

$fl(z_n) = fl(y_n) \wedge pred''$

and pred″ does not contain further restrictions over variables z_j, $j = k+1,\ldots,n$, and if fl is an identification function w.r.t. dbo, then the variables $y_1,\ldots,y_k$ are redundant and may be substituted by compatible variables.

The following figure will present the situation in our example query (Q2.2) where e_no gives the identification function. The variable x″ is redundant.



The substitution of the redundant variables leads to

```
R'= ...
       (select t_expr[y_1/x_1,...,y_k/x_k]
        from   red_f_list[y_1/x_1,...,y_k/x_k]
        where  pred"[y_1/x_1,...,y_k/x_k,
                     z_1/x_1,...,z_k/x_k,
                     z_k+1/y_k+1,...,z_n/y_n])
    ....
    ....
```

respectively to

```
R'= ...
       (select t_expr[y_1/x_1,...,y_k/x_k]
        from   red_f_list[y_1/x_1,...,y_k/x_k]
        where  pred"[y_1/x_1,...,y_k/x_k,
                     z_k+1/y_k+1,...,z_n/y_n])
    ....
    ....
```

Again red_f_list denotes a from-list where the declarations of all redundant variables have been removed from f_list.

Applying this transformation to our example query (Q2.2) we get:

```
(Q2.3) select tuple(d: x.dep_no,
                    e:(select tuple(no: y.e_no,
                                    nm: y.name)
                       from y in x.empl)
       from x in departments
```

which is exactly the same as (Q1.4).

If we look into the database catalog we find that the inner sfw-expression specifies the value of the whole complex attribute empl. Therefore we may replace our query (Q1.4) by the query (Q1′), if the renaming of attributes is neglected.

```
(Q1') select tuple(d: x.dep_no,
                   e: x.empl)
      from   x in departments
```

6 Further Query Transformations in HDBL

The data model of HDBL is structurally object oriented [11] which means that a whole complex structure may be treated in the same way as an atomic structure in the query language, i.e. the term x.attr may denote, for example, an integer value or a set of tuples of lists of integer. Selecting all the employees from the departments database object we would state:

```
select  y
from    x in departments, y in x.empl
```

Here the term y denotes a tuple. An equivalent formulation in HDBL is the following:

```
select tuple(e_no: y.e_no, name: y.name)
from    x in departments, y in x.empl
```

This query breaks up the structure of a tuple into atomic parts and these parts are put together to form the specified result structure which in this case is the original structure.

To speed up the internal processing, the implementation of our prototype system supports the object oriented approach by offering a "complex object at a time" interface at the access system. In addition to fetch a complete complex object it is also able to perform some kind of projections.

Based on these features we extend our language to be able to express such operations directly in the query and offer some transformation rules to rewrite a query using this projection operation which directly maps to a call of the access system.

Let T denote a type with at least one subtype TUPLE(...). The outermost tuple type is

$TUPLE(a_1\!: T_1,\ldots,a_n\!: T_n)$

where a_i are the attribute names and T_i are types, $i = 1,\ldots,n$.

A p-expression with respect to type T is then given by

$[n_1\!: a_{i1}\ p_1,\ldots,\ n_k\!:\ a_{ik}\ p_k\ ]$

where $a_{ij} \in \{\ a_1,\ldots,a_n\ \}$ and no attribute name appears twice, p_j is either the empty string or a p-expression with respect to type T_{ij} and the n_j are new names to be used as attribute names in the result structure.

If exp is an HDBL expression of type T and p is a p-expression $[n_1\!: a_{i1}\ p_1,\ldots,\ n_k\!: a_{ik}\ p_k\ ]$ w.r.t. T then ***exp p*** is a HDBL term. The type of this term is derived from type T, where according to the p-expression all names a_{ij} are renamed to n_j and all remaining attributes are deleted.

We now present some transformation rules to introduce this extended projection operator.

```
Q = select tuple(b_1: y.expr_1,...,b_k: y.expr_k)
    from   y in Range_y
    where  predicate
```

If predicate is equivalent to true and $expr_i$ is either an attribute name, an attribute name followed by a p-expression or a p-expression, and no attribute name appears twice, then Q is equivalent to Q′:

```
Q' = Range_y[ b_1: expr_1,...,b_k: expr_k]
```

There are queries without an explicit tuple constructor in the select-clause, like in

```
select y          select y[...]     select y.attr
from   y in Rg_y  from   y in Rg_y  from   y in Rg_y
```

They are transformed into

```
Rg_y              Rg_y[...]          Rg_y[ attr: attr ]
```

respectively.

Applied to our example (Q1.4) we will get the following expression:

```
(Q1.5) select tuple(d: x.dep_no,
                    e: x.empl[no: e_no, nm: name])
       from x in departments
```

We may apply the same rules one more time and we will finally produce query

```
(Q1.6)
departments[d: dep_no, e: empl[no: e_no, nm:name] ]
```

If we don't care about the renaming of the attributes e_no to no and name to nm we may even write, using information from the database catalog:

```
(Q1.7) departments[d: dep_no, e: empl]
```

The evaluation of this query requires only one call to the storage system.

7 Conclusion and future directions

Based on an extended NF^2 data model we introduced methods and rules for the transformation of queries in HDBL, the SQL-like query language of the AIM prototype. Generated queries may be of arbitrary complexity, for example if views are involved. Based on this observation we presented transformation rules for the elimination of views which means to avoid the explicit construction of intermediate results.

Then we defined some criteria for the detection of redundant variables. A variable is redundant if there is an equivalent query without this variable. Based on a generalized key property we were able to detect redundant variables and we provided rules for the transformation into a query with less variables. This results in a faster query evaluation.

These transformations are based on the substitution of variables by defining terms, which in turn may reveal possibilities for the static evaluation of predicates and the reduction of the where-clauses of a query.

A last step was proposed by introducing some kind of extended projection operator having a direct counterpart at the access system interface, based on the structurally object oriented implementation of the system.

Some of the ideas were taken over from previous work in the relational context, such as [19], where redundant join operations are handled. We adopted the methods to our complex data structures and our data base language. In the relational context this work has been done in an algebraic setting. There are a lot of proposals for an algebra for NF^2-structures [23; 21], but most of them are not powerful enough to cover our intended data structuring capabilities. A recent proposal [1] seems to provide an adequate expressive power and may be a basis for an algebraic treatment of query transformations as in classical relational theory [18] . Another formalism used for query transformation are the so-called tableaux [26] . This formalism does not easily carry over to our complex structures and if it does, it doesn't seem to be an adequate representation of HDBL queries.

Our investigations on query transformations open a scope for the selection of a "good" query. How to find this query and how to decide which is a "good" one is an open problem. In the case of redundant variables there is a criterion for a "better" query. But there are cases where it is advisable not to remove a sfw-expression from a from-list. On the other hand this elimination may reveal further redundancies as shown by our examples.

We are planning to integrate such query transformation rules into our query evaluator of the AIM prototype using concepts and methods developed for rule based systems. To use such methods for the optimization of queries in database systems has been advocated recently in [12; 14] to become more flexible in adding new rules or changing a set of optimization rules.

Acknowledgements

We would like to thank our colleagues working in the AIM project, especially P. Pistor and K. Küspert for carefully reading a prior version of this paper.

References

[1] **S. Abiteboul, C. Beeri:** *On the Power of Languages for the Manipulation of Complex Objects* INRIA Rapports de Recherche No. 846, 1988

[2] **S. Abiteboul, N. Bidoit:** *Non First Normal Form Relations: An Algebra Allowing Restructuring* JCSS, 1986

[3] **D. Batory et al.:** *Genesis: An Extensible Database Management System,* IEEE Trans. on Software Engineering, Vol.14 No.11, Nov. 1988 pp.1711-1730

[4] **M. Carey et al.:** *The Architecture of the EXODUS Extensible Database System,* Proc. 1986 IEEE Int. Workshop on Object Oriented Database Systems, Pacific Grove, pp.52-65

[5] **P. Dadam , K. Küspert et al.:** *A DBMS Prototype to support extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies.* Proc. ACM SIGMOD Conference, Washington D.C., 1986, pp. 356-367

[6] **P. Dadam , K. Küspert et al.:** *Managing Complex Objects in R^2D^2* in G.Krüger, G. Müller(eds.) HECTOR VolumeII: Basic Projects Springer Verlag 1988, pp. 304-331

[7] **P. Dadam , R. Dillmann et al.:** *Object-Oriented Databases for Robot Programming,* in G.Krüger, G. Müller(eds.) HECTOR VolumeII: Basic Projects Springer Verlag 1988, pp. 289-303

[8] **C. Date:** *An Introduction to Database Systems, Vol. I, Fourth Edition,* Addison Wesley Publ. Comp. 1986

[9] **C. Date:** *A Guide to the SQL Standard,* Addison Wesley Publ. Comp. 1987

[10] **U. Dayal et al.:** *Simplifying Complex Objects: The PROBE Approach,* Proc. BTW 1987, Informatik Fachberichte 136, Springer Verlag 1987, pp. 17-37

[11] **K. R. Dittrich:** *Object-Oriented Database Systems: The Notion and the Issues,* Proc. Intl. Workshop on Object Oriented Database Systems, Pacific Grove, Ca., USA, pp. 2-6, 1986

[12] **J. C. Freytag:** *A Rule Based View of Query Optimization,* Proc. ACM SIGMOD-Conference, San Francisco, USA, 1987, pp. 173 - 180

[13] **T. Härder (ed.):** *The PRIMA Project - Design and Implementation of a Non-Standard Database System,* SFB124 Research Report No 26/88, Univ. Kaiserslautern, 1988

[14] **W. Hasan, H. Pirahesh:** *Query Rewrite Optimization in Starburst* IBM Research Report RJ6367, 1988

[15] **Y. Jan:** *Algebraic Optimization for Nested Relations* Proc. 23rd Hawaii Intern. Conf. on System Sciences 1990, Vol. II: Software Track, pp. 278 - 287

[16] **M. Jarke, J. Koch:** *Query Optimization in Database Systems* Computing Surveys, Vol. 16, No. 2, June 1984, pp. 111 - 152

[17] **V. Linnemann et al.:** *Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions,* Proc. 14th Intern. Conference on Very Large Data Bases, Los Angeles, USA, Sept. 1988, pp. 294-305

[18] **D. Maier:** *The Theory of Relational Databases* Pitman Publishing Company, 1983

[19] **N. Ott, K. Horländer** *Removing Redundant Join Operations in Queries Involving Views* Information Systems 10:3, 1985 pp.279-288

[20] **P. Pistor, F. Andersen:** *Designing a Generalized NF^2 Data Model with an SQL-type Language Interface,* Proc. 12th Intern. Conference on Very Large Data Bases, Kyoto, Japan 1986, pp.278-285

[21] **M. Roth, H. Korth, A. Silberschatz:** *Extended Algebra and Calculus for Nested Relational Databases* ACM TODS 13:4, 1988, pp. 389 -417

[22] **M. Stonebraker, L. Rowe:** *The Design of POSTGRES,* Proc. ACM SIGMOD '86, Washington, D.C., pp.340-355

[23] **H.-J. Schek, M. Scholl:** *An Algebra for the Relational Model with Relation-Valued Attributes* Information Systems 11:2, 1986

[24] **M. Scholl:** *Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations* Proceedings ICDT '86 pp. 380-396, Springer Lecture Notes in Computer Science 243 , 1986

[25] **N. Südkamp, V. Linnemann:** *Query Rewriting in an NF^2-like Database Language* Technical Report, IBM Scientific Center Heidelberg, TR 89.10.017, 1989

[26] **J. D. Ullman** *Principles of Database Systems* Pitman Publishing Comp. 2nd. Ed. 1982