Rule-Based Translation of Relational Queries into Iterative Programs

Johann Christoph Freytag[1]
IBM Almaden Research Center, San Jose, CA 95120
Nathan Goodman
Kendall Square Research Corp., Cambridge, MA 02139

Abstract

Over the last decade many techniques for optimizing relational queries have been developed However, the problem of translating these set-oriented query specifications into other forms for efficient execution has received little attention

This paper presents an algorithm that translates algebra-based query specifications into iterative programs for an efficient execution While the source level operates on sets of tuples, the generated programs manipulate tuples as their basic objects The algorithm incorporates techniques which have been developed in the areas of functional programming and program transformation

1. Introduction

With today's computer technology, data are frequently stored and accessed by a database management system (DBMS) Such a system provides a uniform user interface for submitting requests in the form of queries which the DBMS then executes to produce the desired result In a *relational DBMS*, queries are expressed in a data-independent manner A user query describes only the conditions that the final response to the request has to satisfy Therefore, it is the DBMS's responsibility to develop an evaluation strategy to compute the result efficiently The component of the DBMS which decides on the strategy is called the *query optimizer* In a second step the system then has to find a representation of the query which guarantees fast execution We call this problem the *query translation problem*

This paper presents a solution to the query translation problem We develop an algorithm which translates *relational queries into iterative programs* In a more general context, this kind of translation has been studied in the area of *program transformation* We use some of the techniques developed in that area and show that they provide an elegant solution for the query evaluation problem The proposed algorithm is based on sets of transformation rules, thus allowing us to easily extend it to handle more complex queries

The paper is organized as follows In the next two subsections we describe query optimization and evaluation in some detail and outline some aspects of functional programming and program transformation Section 2 defines the source and target levels of the transformation before we describe the algorithm in detail Finally, we extend the proposed algorithm to new operators and discuss some important aspects of our approach

1.1 Query Optimization and Evaluation

Most relational database systems consist of two major components the logical database processor (LDBP) and the physical database processor (PDBP) For example, in SYSTEM R the LDBP is called the Relational Data System (RDS) and the PDBP is called the Relational Storage System (RSS) [ASTR76]

The LDBP translates a user query into an internal representation and optimizes it to guarantee efficient evaluation The query optimizer, a subcomponent of the LDBP, decides on the best evaluation strategy for a user-submitted query Based on information about the internal representation of the accessed data and the evaluation strategies available for each operator in the query, the query optimizer produces a *query evaluation plan (QEP)* For example, the existence of indices or information about the physical order of tuples in the relations influences the optimizer's choice of either an index-join, a merge-join or a nested-loop join as the optimal evaluation strategy for the join operator Jarke and Koch give a comprehensive overview on various optimization techniques for relational queries [JARK84] We assume that algebraic operators, which manipulate sets of tuples, describe the final output of the query optimizer

As a second step, the PDBP must evaluate the generated QEP against the database to compute the requested result The PDBP has to satisfy two conflicting requirements On the one hand, its interface should be set-oriented, thus allowing immediate evaluation of QEPs, on the other hand it must be amiable to an efficient execution To ensure the latter, many database systems implement a PDBP whose operations manipulate tuples, or records, as their basic objects This choice of a PDBP introduces the problem of mapping QEPs into sequences of operations executable by the PDBP We call this mapping problem the *query translation problem* which is the subject of this paper

1 This research was done while the author was a student at Harvard University Cambridge MA 02138 The work was supported by the Office of Naval Research under grant ONR-N00014-83-K-0770

One possible solution to the query translation problem is to provide a set of procedures which independently implement each of the operators in the QEP Each procedure is implemented in terms of operations on the PDBP For example, Buneman et al use this approach for the evaluation of QEPs [BUNE82] Their implementation takes advantage of "lazy evaluation" [FRIE76] to keep intermediate results between operators as small as one tuple Buneman's approach gives an *interpretative solution* to the query translation problem Operators in the QEP are evaluated by calling procedures at execution time

An alternative solution is the *compilation*, or *translation*, of QEPs into programs which directly invoke operations of the PDBP For instance, Lorie et al translate QEPs into assembler programs which embed calls to the RSS in SYSTEM R [LORI79] They design an access specification language (ASL) which precisely defines all valid operators of QEPs in SYSTEM R However, due to the complexity of the compilation step, R* replaces the compilation of ASL statements by their interpretation [DANI82]

Our solution, described in this paper, also compiles queries However, using a two-level translation approach, we avoid the complexity encountered by Lorie First, we compile the set-oriented QEPs into iterative programs expressed in higher level programming languages C or PASCAL, extended by operations of the PDBP, are possible target languages of the translation algorithms we develop In the second step, existing compilers for these languages can then translate the generated programs into machine-executable code

The two-level translation approach has several advantages First and most important, the generation of compact, iterative programs from set-oriented query specifications eliminates unnecessary procedure calls and repeated evaluation of the same function calls, thus *minimizing* the overhead during execution The compiler for the programming language, chosen as the target language for the first translation step, may further optimize the iterative programs during the compilation into machine language Second, the generation of programs in higher level languages guarantees a certain degree of *machine independence* The creation of these programs does not require any knowledge of a particular machine environment Switching compilers achieves portability onto different machines Third, the separation into two independent steps *simplifies* the translation problem High level programming constructs easily express QEPs without considering other aspects of the intended execution environment Additionally, the suggested translation avoids duplicating tasks that compilers already perform We assume that these carry out standard program optimization and optimization for a particular machine environment

1.2. Functional Programming and Program Transformation

Generally speaking, program transformation promises to provide a comprehensive solution to the problem of producing programs which try to solve several incompatible goals simultaneously On the one hand, programs should be correct and clearly structured, thus allowing easy modification On the other hand, one expects them to be executed efficiently Using languages like C or PASCAL for implementation of programs, one is immediately forced to consider aspects of efficiency that are often unrelated to their correctness, natural structure, and clarity

For this reason, the transformational approach tries to separate these two concerns by dividing the programming task into two steps The first step concentrates on producing programs that are written as clearly and understandably as possible without considering efficiency issues If initially the question of efficiency is completely ignored, the resulting program might be written very comprehensibly, but might be highly inefficient or even unexecutable The second step then successively transforms programs into more efficient ones - possibly for a particular machine environment - using methods which preserve the meaning of the original program The kind of improvements during the second step go beyond those achievable during the optimization phase of compilers for conventional programming languages

In many ways, these intentions guided the design of query languages for relational database systems, such as QUEL [STON76] or SQL [ASTR76] Both languages permit the user to express database requests in a clear and understandable form that describes properties of the requested result without considering aspects of an efficient evaluation It is the responsibility of the DBMS to find an efficient execution strategy As already discussed, the query optimizer produces a QEP which determines the major evaluation steps However, the QEP still needs further refinement to guarantee an efficient execution

Many transformation methods consider the manipulation of recursively defined programs or programs defined by recursive equations As Burstall and Darlington state, "the recursive form seems well adapted to manipulation, much more than the usual Algol-style form of programs " [BURS77] Much research has also focused on the translation schemes that replace recursion by iteration [COHE80] Burstall and Darlington developed a set of transformation schemes for the recursion removal together with a systematic approach for the manipulation of recursive programs [DARL76] [BURS77] They introduced transformation steps such as unfolding, folding, abstraction, and the application of simplification rules Using these steps, they successfully implemented a semi-automatic transformation system

Our transformation algorithm reflects this direction in program transformation Without user guidance it generates iterative programs for set-oriented query specifications The algorithm relies exclusively on the definition of recursive programs and their transformation by unfolding, rule-driven simplification, folding, and the replacement of recursion by iteration

2. The Source Level and the Target Level of the Transformation

This section introduces the operators used to describe the source and the target level of the transformation We also introduce a language which is powerful enough to express iterative programs executable on the PDBP

The QEP produced by the query optimizer determines the major computational steps for the evaluation of the query We assume that these actions are specified as algebraic operators which we call *actions* These operate on either tables or sets of tuples Sets

of tuples differ from tables in that we leave unspecified whether they reside in main memory or on secondary storage This aspect becomes irrelevant for performing the transformations

We define the following set of actions which is sufficiently large to demonstrate the proposed transformations

(SCAN table)

is an operator which accesses a table and returns a set of all of its tuples

(FILTER pred? set_of_tuples)

returns the subset of all those tuples satisfying the predicate pred?

(PROJECT project_list set_of_tuples)

projects a set of tuples onto the specified attributes of the projection list

(LJOIN join? set1 set2)

defines a nested loop join of tuple set set1 with set2 Each tuple in set1 is matched with every tuple in set2 to test the join predicate join? If the test evaluates to true, their concatenation is added to the output set

A QEP is said to be *well-formed* if it is either a SCAN operator accessing a table or an action whose input set is produced by a well formed QEP

The current set of actions does not include operations which manipulate indices, create new tables, or store tuples in tables We shall later demonstrate how these actions are easily integrated into the transformation algorithms

Example 1

We use the following database for the examples in this paper

EMP (Emp#, Name, Salary, Dept, Status)

PAPERS (Emp#, Title, Year)

Each tuple in the relation describes an employee by his or her employee number, name, salary, department, and status Relation PAPERS stores the employees who wrote papers recording the title and the year of the publication

For the query

Find the names of all professors who published papers after 1980,

the query optimizer may have decided on the following QEP

```
(PROJECT (Name)
  (LJOIN (Emp# = Emp#)
    (FILTER (Status = Prof) (SCAN EMP))
    (FILTER (Year > 1980) (SCAN PAPERS))))
```

□

While QEPs perform operations on sets of tuples or tables, the PDBP operates on individual tuples Abstracting from a particular implementation, we introduce the notion of streams [FRIE76] Streams are sequences of tuples which [FRIE76] used in the context of lazy evaluation In our context the word connotes images of tuples streaming through the different operators We define a generic set of operators that manipulate streams while hiding details of a specific PDBP This approach treats the access to tables and sets of tuples in a uniform way We define the following five operators which access, create and test streams

(first stream)
returns the first element of the stream

(rest stream)
returns a new stream by removing the first element of the stream

empty
returns the empty stream

(out ele stream)
creates a new stream whose first element *ele* is followed by all elements in *stream*

(empty? stream)
evaluates to true if the stream is empty and to false if stream is an expression of the form (out ele stream)

While actions in QEPs completely specify the source language, a target language has to be defined to express programs resulting from the translation These programs then include calls to the defined stream functions We propose a small functional language in a Lisp-like notation which is well suited for formal manipulation We shall describe the target language informally The language is based on expressions An expression is either a variable, a function expression, or a conditional expression A function expression has the form $(f_1\ t_1\ \ t_n)$, where f_1 is a function symbol and t_i , $i = 1,\ \ ,n$ are expressions called actual parameters A conditional expression has the form $(if\ t_1\ t_2\ t_3)$, where t_1, t_2, t_3 are expressions If t_1 evaluates to true, then the value of t_2 is the value of the expression, otherwise it is the value of t_3 When t_3 is superfluous we use the form $(if\ t_1\ t_2)$

Functions may be defined by the equation $(f_1\ x_1\ \ x_n) = t_1$, where f_1 is a function symbol, the x_i are formal parameters, and t_1 is an expression built from function symbols, possibly including f_1 and variables x_i $(f_1\ x_1\ \ x_n)$ is called the *function header* and t_1 is called the *function body* If f_1 is invoked in a function expression, the value is determined by substituting t_1 for the expressions and by replacing the formal parameters by the actual ones Functions and expressions are sufficient for most steps of the translation process However, to express the final iterative programs we use a PASCAL-like notation for loops and assignment statements

3. A Transformation Algorithm based on Recursive Programs

This section develops the transformation algorithm which is based on the manipulation of recursive programs The algorithm uses five major steps to translate QEPs into iterative programs They are explained in more detail in the following subsections The first step, called *unfolding*, or *substitution*, replaces an action by its corresponding recursive function During the second step, a set of transformation rules *simplifies* the initial program without changing its intended meaning The simplification step produces a minimal form of the program The third step, *folding*, possibly reduces the program generated so far This step is best characterized as the inverse to unfolding If necessary, the fourth step, called *union step*, further manipulates programs by a set of transformation rules to successfully apply the final step which replaces recursion by iteration We present the complete transformation algorithm in the last subsection

3.1 Unfolding

For the first step of the transformation, unfolding, we define programs, called *DB functions*, expressed in the target language that we introduced Each DB function correctly implements a valid action (or part of it) in the QEP A combination of DB functions is called a *DB expression*

While actions are denoted by capital letters, DB functions will be denoted by lower case letters Each DB function calls only itself, other DB functions, and possibly additional functions which are denoted by *pred*?, *join*?, and *prj* These implement the predicates used as the first parameters to actions FILTER and LJOIN, and the projection list for action PROJECT, respectively For the transformation, developed in this section, it is unimportant how these functions implement the predicates and the projection list For our purposes it is sufficient to denote these functions by lower case letters For the implementation of LJOIN we also introduce the function *conc* to concatenate two tuples We define the following DB functions

```
(scan table) =
  (if (empty? table) *empty*
    (out (first table) (scan (rest table))))

(filter stream) =
  (if (empty? stream) *empty*
    (if (pred? (first stream))
      (out (first stream) (filter (rest stream)))
      (filter (rest stream))))

  (project stream) =
    (if (empty? stream) *empty*
      (out (prj (first stream)) (project (rest stream))))

  (ljoin str1 str2) =
    (if (empty? str1) *empty*
      (union (jn (first str1) str2) (ljoin (rest str1) str2)))
```

```
(union str1 str2) =
  (if (empty? str1) str2
    (out (first str1) (union (rest str1) str2)))


(jn ele stream) =
  (if (empty? stream) *empty*
    (if (join? ele (first stream))
      (out (conc ele (first stream)) (jn ele (rest stream)))
      (jn ele (rest stream))))
```

The definition of functions *scan, project, filter, jn*, and *union* all have a common form which is called *linear recursive form* [COHE80] A function f has a linear recursive form if f is the only recursive function called in its definition Function *ljoin* accesses two streams and its definition reflects the functional decomposition Function *jn* joins the first tuple of the first stream with all tuples of the second The *union* concatenates the result of the join with the set of tuples resulting from the recursive call to *ljoin* Since the definition of *ljoin* contains calls to other linear recursive functions, its definition has an *extended linear recursive form* [COHE80] [FREY85]

The reader may have noticed the difference in the number of parameters for actions FILTER, PROJECT, and LJOIN, and their implementing DB functions *filter, project*, and *ljoin*, respectively The first parameter does not occur as a parameter in the corresponding function Not including them in the definition header will simplify the later transformation significantly Instead these parameters are "compiled into" the DB functions We denote the "specialized" functions by *filter1, project1*, and *ljoin1*

Example 2

The QEP of example 1 translates into the DB expression

```
(project1 (ljoin1
  (filter1 (scan EMP))
  (filter2 (scan PAPERS))))
```

where

- *project1* is a specialization of *project* containing projection list prj1 which implements the projection list (Name)
- *filter1* is a specialization of *filter* containing a call to pred1? which implements predicate (Status = Prof)
- *filter2* is a specialization of *filter* containing a call to pred2? which implements predicate (Year > 1980)
- *ljoin1* is a specialization of *ljoin* which calls *jn1* *jn1* is a specialization of *jn* calling the joining function *join1*? which implements the join predicate (Emp# = Emp#)

□

For space reasons we do not show the expression for the above example after unfolding However, the resulting expression allows us to investigate the "interactions" between functions When combined, the same functions with the same parameters may appear more than once Therefore, the next step, called *simplification*, eliminates superfluous function calls by using a set of transformation rules which we introduce in the following subsection

3.2. Simplification

To simplify expressions in our language, we introduce a set of *transformation rules*, or *rewriting rules* These rules generate a canonical form for each expression with a minimal number of conditional expressions and only one output stream For example, the rule

$((if\ true\ t_1\ t_2) \rightarrow t_1)$

specifies to replace any expression whose condition is *true* by the consequent expression t_1, where t_1 and t_2 are arbitrary expressions

For the manipulation of expressions we define the following set of rules

Rule I Function exchange rule
$((f\ (if\ t_1\ t_2\ t_3)) \rightarrow (if\ t_1\ (f\ t_2)\ (f\ t_3)))$

Rules IIa and IIb Deletion rules
$((if\ t_1\ (\ (if\ t_1\ t_2\ t_3)\ )\ t_4) \rightarrow (if\ t_1\ (\ t_2\ )\ t_4))$
$((if\ t_1\ t_2\ (\ (if\ t_1\ t_3\ t_4)\ )) \rightarrow (if\ t_1\ t_2\ (\ t_4\ )))$

Rule III Distribution rule
$((if\ (if\ t_1\ t_2\ t_3)\ t_4\ t_5) \rightarrow (if\ t_1\ (if\ t_2\ t_4\ t_5)\ (if\ t_3\ t_4\ t_5)))$

Rules IVa and IVb True/false rules
$((if\ true\ t_1\ t_2) \rightarrow t_1)$
$((if\ false\ t_1\ t_2) \rightarrow t_2)$

Rules Va, Vb, Vc, and Vd Stream rules
$((first\ (out\ t_1\ t_2)) \rightarrow t_1)$
$((rest\ (out\ t_1\ t_2)) \rightarrow t_2)$
$((empty?\ *empty*) \rightarrow true)$
$((empty?\ (out\ t_1\ t_2)) \rightarrow false)$

The function exchange rule distributes functions with one parameter over conditional expressions Note that we only define the function exchange rule for *one-parameter functions* In [FREY85] we show that this rule is the only one necessary for the transformation of expressions generated from actions The deletion rules replace superfluous conditional expressions by either the consequent or the alternate of the eliminated expression The distribution rule simplifies two nested conditional expressions The true/false rules implement partial evaluation for conditional expressions The remaining rules reflect some semantic knowledge about stream operators They help to reduce the number of intermediate results by performing partial evaluation on streams

This set of transformation rules is certainly not the most general one Other researchers have proposed larger sets of rules which can transform more general programs [GIVL84] [BELL84] [BIRD84] However our set of rules is sufficient to transform all functions and programs derived from QEPs

Our proposed set of rules enjoys two important properties first, the transformation always terminates Second, the final expression form is independent of the order in which we apply the rules to the initial expression When applied exhaustively, the final form is unique This property is called the *Church-Rosser property*, or *CR* for short Huet developed a *theory of term rewriting systems* to test if the *CR* property holds for any set of transformation rules [HUET80] More details to prove the *CR* property for the above set can be found in [FREY85]

3 3 Folding

Consider the DB expression (project1 (filter1 str)) which derives the minimal form

```
(if (empty? str) *empty*
  (if (pred1? (first str))
    (out (prj1 (first str))
      (project1 (filter1 (rest str))))
    (if (empty? (filter1 (rest str))) *empty*
      (out (prj1 (first (filter1 (rest str))))
        (project1 (rest (filter1 (rest str))))))))
```

This expression already contains fewer operations than the one resulting from unfolding[2] However, we would like to derive an expression which refers only to (first str) and (rest str) The function expressions (first (filter1 (rest str))) and (rest (filter1 (rest str))) should be eliminated We recognize that the subexpression

```
(if (empty? (filter1 (rest str))) *empty*
  (out (prj1 (first (filter1 (rest str))))
    (project1 (rest (filter1 (rest str))))))
```

resembles the definition of the project function using the actual parameters (filter1 (rest str)) The third step, called *folding*, replaces a subexpression t' by the function expression $(f_1\ t_1\ \ t_n)$, if t' is equal to the body of f_1 up to renaming of actual parameters $t_1\ \ t_n$ For the above subexpression we substitute the function expressions (project (filter1 (rest str))) into the minimal expression, derived by unfolding and simplification, thus yielding the desired expression

```
(if (empty? str) *empty*
  (if (pred1? (first str))
    (out (prj1 (first str)) (project1 (filter1 (rest str))))
    (project1 (filter1 (rest str)))))
```

Note that (first str) is now the only parameter to functions *pred1?* and *prj1* and that the remaining recursive calls in the expression are performed on the same combination of functions as in the initial DB expression with (rest str) being the only parameter for the recursive calls In the above example the function combination is (project1 (filter1 (rest str))) We say that the DB expression has an *ideal form* In [FREY85] we show that any combination of *project, filter, scan,* and *jn* functions has an ideal form which is derivable by unfolding, simplification, and folding

3 4. The Union Step

Unfortunately, these steps are not sufficient for the transformation of DB expressions which include operations *join* and *union* We need to introduce another set of transformation rules to derive the ideal form for those functions They form a fourth transformation

2 Again for space reasons we do not show the expression resulting from unfolding However the reader may convince himself (herself) by deriving the expression using unfolding

step called the *union step* To correctly define the rules, we need to define *restricted rules* A restricted rule is denoted by

$$(t_1 \xrightarrow{C} t_2)$$

where C is a condition containing variables used in t_1 The rule can be applied if the condition evaluates to *true* when assigning values to variables in t_1 (and therefore in C) For the union step we define the following four rules

$$((\text{union } (\text{union } t_1\ t_2)\ t_3) \rightarrow (\text{union } t_1\ (\text{union } t_2\ t_3)))$$
$$((f_1\ (\text{union } t_1\ t_2)) \rightarrow (\text{union } (f_1\ t_1)\ (f_1\ t_2)))$$
$$((f_2\ (\text{union } t_1\ t_2)\ t_3) \xrightarrow{C} (\text{union } (f_2\ t_1\ t_3)\ (f_2\ t_2\ t_3)))$$
$$((f_2\ (t_1\ (\text{union } t_2\ t_3))) \xrightarrow{C} (\text{union } (f_2\ t_1\ t_2)\ (f_2\ t_1\ t_3)))$$

where $t_i, i = 1 \ldots n$ are arbitrary expressions, f_1 is a function with one parameter, and f_2 is a function with two parameters The last two rules are needed for functions such as *Ijoin* For their application we have to exclude function *union* and impose the restriction $C = (f_2 \neq \text{union})$ Notice that the restriction is important for two reasons First, if $(f_2 = \text{union})$ the transformation result is semantically incorrect Second, the rules can be applied forever, that is the transformation is not finite any more However if we allow conditions on rules as restrictions for their application, then the above set has the *CR*-property [FREY85] In [FREY85] we also prove that *any combination of DB functions yields an ideal form using the four transformation steps unfolding, simplification, folding and the union step*

3.5. The Simplification Algorithm

Based on the definitions of the four transformation steps we define algorithm ID which computes the ideal forms for any DB expression If $t = (f_1\ (f_2\ (f_3\ str)))$ the algorithm first computes the ideal form for $(f_2\ (f_3\ str))$ before the combination with f_1 is encountered Let *SIM* denote the simplification by the set of transformation rules, *FL* denote the folding step, and let *UN* denote the union step If $(f\ str) = t$ is a function definition then $B(f)$ denotes the body of the function, i e $B(f) = t \quad t[str \mid t_1]$ denotes the substitution of all variables *str* in t by expression t_1 Then the algorithm is defined as follows

ALGORITHM ID

INPUT DB expression t ,
OUTPUT ideal form of t

Assume all table names to be unique

if $t = (f_1\ t_1)$ and t_1 is a table name
return $(B\ (f_1)\ [str_1 \mid t_1])$,

if $t = (f_1\ t_1)$ and t_1 is a DB expression
$l_1 = \text{ID}\ (t_1)$,
return $(UN\ (FL\ (SIM\ (B\ (f_1)\ [str_1 \mid l_1]))))$,

if $t = (f_1\ t_1\ t_2)$
$l_1 = \text{ID}\ (t_1)$, $l_2 = \text{ID}\ (t_2)$,
return $(UN\ (FL\ (SIM\ (B\ (f_1)\ [str_1 \mid l_1,\ str_2 \mid l_2]))))$,

end_of_algorithm

In the first case of the algorithm we have reached the bottom of the recursion and do not need any simplification or folding The algorithm simply returns the body of the function with the actual parameters substituted The second and third case take care of the combination with functions which manipulate one or two streams, respectively In both cases, variables t_1 and t_2 represent DB expressions which are transformed by calling ID recursively

3 6. Recursion Removal

The final transformation step that we propose for the improvement of programs replace recursion by iteration While the recursive form supports clear programming style and simplifies program manipulation, it may not be the most efficient form for program execution [COHE80] [BURS77] We use the transformation schemes proposed by Burstall and Darlington for linear recursive functions [BURS77] Consider the DB function project

```
(project str) =
  (if (empty? str) *empty*
    (out (prj (first str)) (project (rest str))))
```

which can be transformed into the iterative program

```
result = *empty*, stream = str,
WHILE not (empty? stream) DO
  result = (out (prj (first stream)) result),
  stream = (rest stream),
END_DO
```

The iterative form introduces the variable *result* which accumulates the intermediate result on each call to function *out* The variable is initialized to **empty**, the "bottom" value of the recursion Notice that this transformation is only possible if we are *not* concerned with the order of elements

The same transformation can be applied to all other DB functions Most important, the ideal form of expressions *guarantees that recursion can be replaced by iteration for the combination of DB functions* [FREY85]

3 7 The Recursive Transfomation Algorithm

Now we are ready to describe the complete transformation algorithm which uses recursive function definitions Let Π denote the mapping from QEPs to DB expressions using the conventions of Section 3 1 Let $RI(t_1, t_2)$ denote the procedure which replaces recursion by iteration in t_2 where t_1 is the combination of DB functions which determines the recursion

ALGORITHM TR_R

```
Input  Query evaluation plan qep,
Output  Iterative program
  t = Π(qep),   t1 = t,
  while t1 contains a DB expression  do
    /* compute the ideal form */
    t2 = ID(t1),
    /* generate iterative expression */
    t3 = RI(t1, t2),
    /* replace recursion by iteration */
    t = t[t1 | t3],
    t1 = More_DB_epxressions?,
  end_do,
  generate_assignment_statements,
end_of_algorithm
```

When given a QEP with n tables, algorithm TR_R produces an iterative programs with n nested loops [FREY85] During each iteration, algorithm it replaces one level of recursion by iteration, possibly leaving a recursive subexpression which accesses $n-1$ tables

Example 3

The transformation algorithm TR_R translates the DB expression of example 2 as follows The ideal form produced in the first iteration is of the form

```
(if (empty? EMP) *empty*
  (if (pred1? (first EMP))
    (union
      (project1 (jn1 (first EMP)
        (filter2 (scan PAPERS))))
      (project1 (join1
        (filter1 (scan (rest EMP)))
        (filter2 (scan PAPERS)))))
```

which yields the iterative expression

```
str1 = EMP,
WHILE not (empty? str1) DO
  IF (pred1? (first str1)) THEN
    (union
      (project1 (jn1 (first str1)
          (filter2 (scan PAPERS))))
      result)
    END_IF,
    str1 = (rest str1),
END_DO
```

The iterative expression still contains another DB expression calling DB functions, namely (union (project1 (jn1 (first str1)

In the next round the algorithm produces an ideal form for this subexpression, replaces recursion by iteration and substitutes the result into the above expression The expression t does not contain any further calls to any recursive function, thus terminating the loop of TR_R Adding the appropriate assignment statements, the final iterative program in PASCAL-like notation is

```
result = *empty*, str1 = EMP,
WHILE not (empty? str1) DO
  IF (pred1? (first str1)) THEN
    str2 = PAPERS,
    WHILE not (empty? str2) DO
      IF (pred2? (first str2)) THEN
        IF (join1? (first str1) (first str2))      THEN
            result =
              (out (conc (first str1)
                   (first str2)) result),
      str2 = (rest str2),
      END_DO,
    str1 = (rest str1),
  END_DO,
```

□

4. Discussion

In this section we show how to extend the current algorithm and briefly discuss implementation related aspects

4 1 Extending the Algorithm

As the reader may have noticed, we introduced a small set of actions for the proposed transformation algorithm How is the algorithm affected when new actions are added to the existing set? Suppose the PDBP provides indexes on tables for fast access to individual tuples As the query optimizer decides whether to use an index or not, we need to introduce two new actions for indexes

(SEARCH pred? index)

is an operator that returns a *table* with all the tuples satisfying the predicate *pred?* Action SCAN may then retrieve the index entries in the "restricted" table

(GET table set_of_index_tuples)

retrieves all tuples in the table according to the index tuples provided in the input set

For the manipulation of indexes by the PDBP we introduce two tuple operations

(access_index pred? index)

returns a table containing all tuples which satisfy the provided predicate *pred?* This operator directly implements action SEARCH for QEPs without specifying further details of its implementation by the PDBP

(get_tuple table i_tuple)

given an index entry i_tuple, returns the corresponding tuple from the specified table

Using these two operators, we implement SEARCH and GET by the DB functions search and get, respectively

```
(search index) =
  (access_index pred? index)

(get stream) =
  (if (empty? stream) *empty*
    (out (get_tuple table (first stream))
      (get (rest stream))))
```

As with DB expressions *filter, project* and *ljoin*, the first parameters of *search* and *get* do not occur in the header of the implementing functions They are "compiled" into the function bodies to simplify the transformation These definitions are sufficient to extend the algorithm to new actions Since the functions which implement both actions are *equivalent in structure* to the already existing ones, we do not have to change the algorithm itself

Example 4

Suppose there exists an index ISTAT on attribute Status for relation EMP For the query in Example 1 the query optimizer may have decided on the QEP

```
(PROJECT (Name)
  (LJOIN (Emp# = Emp#)
    (GET EMP (SCAN (SEARCH (Status = Prof) ISTAT)))
    (FILTER (Year > 1980) (SCAN PAPERS))))
```

□

The current set of actions does not include operators to create new tables or store tuples in tables We assume that these operations are implemented by the PDBP To cope with intermediate results in queries, our approach is easily extended We define *query programs QPs*, i e straight-line sequences of QEPs QPs may express any evaluation requiring intermediate results the query is divided into subqueries Each subquery is evaluated by some QEP storing its intermediate result in a newly created table which may be accessed by subsequent QEPs

A major achievement of the transformation algorithm is the elimination of intermediate streams between calling and called functions This transformation is often called *vertical loop fusion* [GOLD84] We may also be interested in another form of loop elimination called *horizontal loop fusion* two programs which access the same stream are combined into one program with one loop The new program now computes both results in "parallel", thus accessing the stream only once Consider the DB expression [3]

```
(union (filter1 str) (filter2 str))
```

The current transformation algorithm creates a program which accesses the stream *str* twice In [FREY85] we show how to perform horizontal loop fusion by transformation Unfortunately, this kind of transformation needs some user guidance and cannot be easily included in algorithm TR_R

4.2. Implementation

We have implemented algorithm TR_R using the Lisp dialect T [REES82] The implementation is rather straightforward since we can express all transformation steps, unfolding, simplification, folding, and recursion removal, by formal rewriting rules Therefore, the heart of the implementation is a *rule application function* which applies a set of rules once (in the case of unfolding) or exhaustively to a provided expression In general, rules within each transformation step can be applied in any order However, there may exist an optimal order that derives the minimal final form by a minimal number of rule applications For example, consider the simplification step for expression $t = (if\ true\ t_1\ t_2)$ Any transformation of subexpression t_2 is superfluous since the true-rule eventually reduces t to t_1 Our implementation does not attempt to solve this optimization problem

5. Conclusion

We presented a solution to the query translation problem which is based on techniques of program transformation We showed that the recursion based transformation leads to an elegant algorithm which has several desirable properties First, the algorithm creates compact iterative programs from algebraic query specifications Second, the algorithm is extendible to new operators without changing the overall structure of the transformation Finally, we use rules to describe uniformly the different steps of the algorithm, thus leading to an immediate suggestion for implementing our transformation

6. Acknowledgement

We wish to thank Guy Lohman, Laura Haas, and Bill Cody for carefully reading a draft of this paper While the first author was a student at Harvard University, this work was supported by the Office of Naval Research under grant ONR-N00014-83-K-0770

Bibliography

[ASTR76] Astrahan, M et al , *SYSTEM R Relational Approach to Database Management,* ACM Transactions of Database Systems **1,2** (June 1976) pp 97-137

[BELL84] Bellegarde, F , *Rewriting Systems on FP Expressions that Reduce the Number of Sequences they Yield,* ACM Symposium on LISP and Functional Programming (August 1984) pp 63-73

[BIRD84] Bird, R S , *The Promotion and Accumulation Strategies in Transformational Programming,* ACM Transactions on Programming Languages and Systems **6,4** (October 1984) pp 487-504

[BUNE82] Buneman, P and Frankel, R E , *An Implementation Technique for Database Query Languages,* ACM Transactions on Database Systems **7,2** (June 1982) pp 164-186

3 This expression is not derivable from any QEP so far

[BURS77] Burstall, R,M and Darlington, J , *A Transformation System for Developing Recursive Programs,* Journal of the ACM **24,1** (January 1977) pp 44-67

[COHE80] Cohen, N H , *Source-to-Source Improvement of Recursive Programs,* PhD Thesis, Harvard University (May 1980)

[DANI82] Danies, D , *Query Compilation in a Distributed Database System,* Technical Report RJ #3432, IBM Research Laboratoy, San Jose, CA 95193 (March 1982)

[DARL76] Darlington, J and Burstall, R M , *A System which Automatically Improves Programs,* Acta Informatica **6,1** (January 1976) pp 41-60

[FREY85] Freytag, J C , *Translating Relational Queries into Iterative Programs,* PhD Thesis, Harvard University, also Technical Report TR-14-85 (September 1985)

[FRIE76] Friedman, D P and Wiese, D S , *CONS should not Evaluate its Arguments,* in S Michaelson and r Milner, *Automata, Languages, and Programming,* Edinburgh University Press (Edinburgh, 1976) pp 257-284

[GIVL84] Givler, J S and Kieburtz, R B , *Schema Recognition for Program Transformation,* ACM Symposium on LISP and Functional Programming (August 1984) pp 74-84

[GOLD84] Goldberg, A and Paige, R , *Stream Processing,* ACM Symposium on LISP and Functional Programming (August 1984) pp 53-62

[HUET80] Huet, G , *Confluent Reductions Abstract Properties and Applications of Term rewriting Systems,* Journal of the ACM **27,4** (October 1980) pp 797-821

[JARK84] Jarke, M and Koch,J , *Query Optimization in Database Systems,* ACM Computing Surveys **16,2** (June 1984)

[LORI79] Lorie, R A and Wade, B W , *The Compilation of a High Level Data Language,* Technical Report RJ #2598, IBM Research Laboratory, San Jose, CA 95193 (1979)

[REES82] Rees, J A and Adams, N , *T A Dialect of LISP or, LAMDA The ultimate Software Tool,* ACM Symposium on LISP and Functional Programming (August 1982)

[STON76] Stonebraker, M and Wong, E , et al , *The Design and Implementation of INGRES,* ACM Transactions of Database Systems **1,3** (September 1976) pp 189-222