ADDING RELATIONAL DATABASES TO EXISTING SOFTWARE SYSTEMS: IMPLICIT RELATIONS AND A NEW RELATIONAL QUERY EVALUATION METHOD

by

Susan Horwitz

Computer Sciences Technical Report #674

November 1986

Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method

Susan Horwitz
University of Wisconsin–Madison

Abstract

Interactive software systems should include query handlers. Query handlers based on the relational database model are attractive because the model provides a uniform, non-procedural approach to query writing. Standard relational database systems require that all information be stored in relations; however, the data structures used by existing software systems are generally non-relational, and it is impractical to replace them with relations.

A new kind of relations, *implicit relations*, and a new approach to query evaluation based on the use of access functions allow software systems to include relational query facilities without giving up existing non-relational data structures. The new query-evaluation method can also be used in traditional relational databases, and may be more efficient than traditional evaluation methods when applied to queries that use set operations.

1. Introduction

It is easy to see the benefits of including query facilities in interactive software systems such as language-based editors, debuggers, and version-control managers. Existing systems that provide query facilities, however, generally do so in a limited way; the user is restricted to a pre-defined set of queries, and query answers cannot be used as inputs to further queries. This is because the query facilities are implemented in an *ad hoc* manner. A better approach would be to take advantage of current database technology, basing the query facility on the *relational database model* [Codd 1970]. Under this model, arbitrary queries can be written by applying a standard set of operators to a set of relations. Query answers are themselves relations, thus are available as inputs to further queries. An advantage of the relational model over the hierarchical and network database models is that the relational model provides a *uniform, non-procedural* approach to query writing.

In relational database systems, this uniform approach to query writing is possible because all information is *represented* uniformly, as sets of tuples called relations. By contrast, most existing software

This work was supported in part by the National Science Foundation under grant DCR-8603356.

systems use non-relational data structures to store information. Given that we wish to access this information through relational queries, it would seem that there are three possible approaches:

(1) All information is stored both in the original data structures and in relations.

(2) The information in the data structures is translated to relational form as needed for query processing.

(3) The original data structures are replaced with relations.

It is clear that for large systems, solution (1) has unacceptable space requirements, while solution (2) may necessitate long delays, unacceptable in interactive systems. Solution (3) may work in some but not all cases. The problem is that operations previously performed on the non-relational data structures would have to be performed on the relations. While relations are designed to allow efficient query evaluation, they may not be so well-suited to these other operations.

Consider a specific example: language-based editing environments. Most language-based editors represent programs as syntax trees; the data structure used is one in which it is easy to get from a tree node to its children or to its parent. Linton [Linton 1984] designed an editor in which the tree data structures were replaced with relations. This representation has the advantage of allowing queries about program structure to be written; unfortunately, some traditional editing operations became unacceptably slow when performed using the relational representation. For example, using the standard INGRES relational database management system [Stonebraker et al 1976], the display of a ten-line procedure body required forty seconds of elapsed time. Even using a version of INGRES specially tuned for this task, the display required seven seconds. By contrast, display of procedures by language-based editors that use the more traditional tree data structures is virtually instantaneous.

Rather than abandoning the goal of adding a relational query facility to existing software systems, we abandon the goal of storing *all* information in relations. When it is practical to do so, non-relational data structures are replaced with relations, or information is stored in both the original structures and in relations. When such replacement or duplication is *not* practical, the information in the non-relational data structures is accessed through the use of a new kind of relations called *implicit relations*. From the query-writer's point of view, an implicit relation is indistinguishable from any other relation; however, implicit

relations are not stored as sets of tuples, instead the informational content of an implicit relation is extracted as needed from arbitrary non-relational data structures during query evaluation.

Section 2 uses the example of language-based editing environments to clarify the concept of implicit relations. Implicit relations that can be derived from the syntax-tree representation of a program are defined, and are used as examples throughout the rest of the paper.

It is desirable that implicit relations and "normal", or *explicit* relations be indistinguishable from the query-*evaluator*'s point of view as well as the query-*writer*'s point of view. Section 3 presents a new query-evaluation method based on the use of three *access functions*. These access functions are provided for *all* relations, thus allowing a uniform approach to query evaluation.

While the new query-evaluation method is vital to the efficient evaluation of queries that include implicit relations, it may be of some interest in the context of "standard" query evaluation as well. This is because the new method is well-suited to the evaluation of queries defined using the *full* set of relational operators, set operators as well as selection, projection, and join. By contrast, previous methods have concentrated on evaluating queries expressed using select, project, and join only. Section 4 introduces a new example query using only explicit relations to illustrate the potential advantages of our query-evaluation method in this context. Section 5 compares the ideas presented in this paper with previous work.

2. Implicit relations: Examples from language-based editing environments

Providing a query facility as part of a language-based editor allows the programmer to ask questions about the program under development. A very important class of questions are those that deal with program *structure*, for example: "Where, outside of procedure P, is variable x used?". Answering this question requires information about the *hierarchical structure* of the program as well as information about points in the program at which variables are used.

As discussed above in the introduction, it is impractical to replace the traditional syntax-tree representation of a program with a relational representation. While it may be reasonable to include information about variable usage both in the program tree and in a relation, it is probably *not* reasonable to

duplicate structural information. Instead, the following *implicit* relation is defined*:

ANCESTOR(ancestor: *program point*, descendant: *program point*)

The information that is *conceptually* stored in the ANCESTOR relation is *actually* implicit in the structure of the syntax tree, and will be extracted as needed during query evaluation.

The question, ''Where, outside of procedure P, is variable x used?'' can be formulated as a relational query using the implicit ANCESTOR relation and an explicit relation containing information about variable usage: USE(where: *program point*, variable name: *string*). Figure 1 shows an example program, the corresponding syntax tree, and the corresponding USE relation.

```
program main;
begin
  procedure P;
    y:=x+1;
  procedure Q;
    x:=x+1;
  procedure R;
    y:=y+1;
end
```

Figure 1a
Example Program

*Throughout the paper, example relations are defined according to the following syntax:
relation–name(field-name$_1$: *type* $_1$, field-name$_2$: *type* $_2$, ..., field-name$_n$: *type*$_n$)

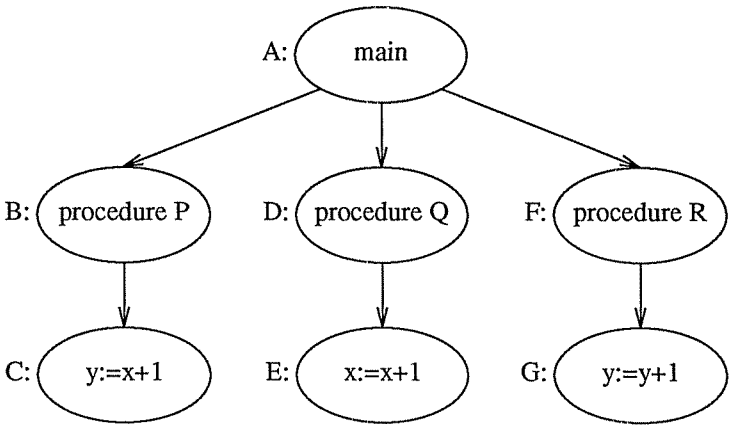


Figure 1b
Corresponding Syntax Tree

USE	where	variable name
	C	y
	C	x
	E	x
	G	y

Figure 1c
Corresponding USE Relation

Figure 2 shows the tree form of the example query, and the relation produced by evaluating the query for the example program of figure 1a. To compute the set of program points outside procedure P at which variable x is used, the set of program points inside procedure P is subtracted from the set of program points at which variable x is used. Note that program points correspond to syntax-tree nodes; the program point that represents procedure P is node "B", the root node of the subtree for procedure P.

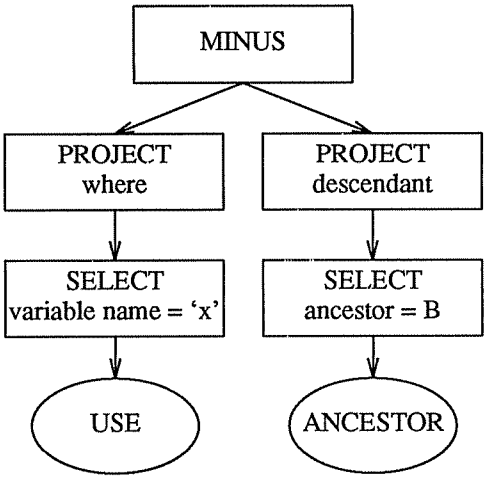


Figure 2a
Example Query Tree:
"Where, outside of procedure P, is variable x used?"

RESULT	where
	E

Figure 2b
Result Relation
Uses of x outside procedure P in the example program

Another example of a relation implicitly available in the syntax tree is the relation of program points and the language construct at that point:

CONSTRUCT(where: *program point*, construct: *language construct*)

The CONSTRUCT relation could be used (in conjunction with the ANCESTOR relation) to formulate questions like: "Which WHILE LOOPS contain PROCEDURE CALLS?".

Naturally, the distinction between implicit and explicit relations must be invisible to the query writer. The method used to evaluate queries must thus be able to handle occurrences of implicit as well as explicit relations. In section 3 below we present a new method for query evaluation. Although motivated by the need to handle implicit relations, the method can be applied to general relational queries; because the

method is designed to handle set operations as well as select, project, and join, it may be more efficient than traditional query-evaluation methods when applied to queries that use set intersection, union, and difference.

3. A new query-evaluation method

The query-evaluation method presented in this section relies on the use of three access functions: *membership test*, *selective retrieval*, and *relation producing* functions.

Definition:

(1) The membership-test function for relation R, given tuple t, returns 'true' if t is in R, and otherwise returns 'false'.

(2) The selective-retrieval function for relation R, given list of fields $f_1, f_2, \ldots f_n$, and list of values $v_1, v_2, \ldots v_n$, returns the set of tuples in R that have value v_i in field f_i for all i.

(3) The relation-producing function for relation R returns the set of tuples in relation R.

The basic idea behind the new query-evaluation method is to use membership-test and selective-retrieval functions to avoid materializing both implicit and intermediate relations. Section 3.1 below discusses how some relational operators can be implemented using their operands' membership-test or selective-retrieval functions in place of the actual operand relations.

Section 3.2 gives an outline of our query-evaluation method. The method assumes the existence of membership-test, selective-retrieval, and relation-producing functions for all relations used in the query, both implicit and explicit. Section 3.3 discusses how these functions can be constructed, using the ANCESTOR relation introduced above as an example. Given these functions to build on, membership-test, selective-retrieval, and relation-producing functions are constructed for the relations represented by the internal nodes of the query tree. The time and space savings of the query-evaluation method come from using the membership-test and selective-retrieval functions associated with implicit and intermediate relations, rather than materializing these relations. Constructing membership-test and selective-retrieval functions for internal nodes is discussed in section 3.4.

3.1. Using membership-test and selective-retrieval functions

Some relational operators can use an operand relation's membership-test or selective-retrieval function in place of an explicit representation of the operand relation. MINUS is one such operator. Given an explicit representation of R1, and a membership-test function for relation R2, one can implement (R1 – R2) as follows: a single scan is made through R1 considering each tuple t in turn; t is in the result relation if it is not a member of R2.

```
let
    R1 and ANSWER be relations
    R2.membership be a membership-test function for relation R2
in
begin
   ANSWER := {};
   for each tuple t in R1 do
       if (¬R2.membership(t))
          then ANSWER := ANSWER ∪ { t };
   return( ANSWER );
end
```

Figure 3
Implementing MINUS using a membership-test function

Similarly, EQUI-JOIN can be implemented using one explicit operand relation and the other operand relation's selective-retrieval function. A single scan is made through the explicit operand relation; for each tuple t, the selective-retrieval function of the other operand relation is called with the number of the join field and the appropriate value from t. All returned tuples are joined with t and added to the result relation. The code given below in figure 4 is for the operation: JOIN (R1, R2) WHEN (field #3 of R1) = (field #1 of R2).

```
let
   R1 and ANSWER be relations
   R2.selective-retrieval be a selective-retrieval function for relation R2
in
begin
  ANSWER := {};
  for each tuple t in R1 do
      for each tuple u in R2.selective-retrieval( (1), (t.field#3) ) do
          ANSWER := ANSWER ∪ JOIN( t, u );
  return( ANSWER );
end
```

Figure 4
Implementing EQUI-JOIN using a selective-retrieval function

The evaluation of queries that include implicit relations is acceptably efficient *only* when calls to the implicit relations' membership-test or selective-retrieval functions can replace ''normal'' access to the relations. The evaluation of queries that include only *explicit* relations can benefit from this technique as well. The code given in figure 4 may appear at first to be no more than a nested-loops join in which an index on the join field is used to look up values in the inner relation [Wong and Youssefi 1976][Selinger et al 1979]. The important innovation introduced by our technique is that R2 need not be a materialized relation; instead, it can represent an arbitrary relational computation, *including set operations.* As discussed below in section 3.4, R2.membership and R2.selective-retrieval can often be implemented so that neither R2 nor any of the intermediate relations involved in the computation of R2 need to be built. Instead, the membership-test or selective-retrieval functions associated with the nodes of the subtree rooted at R2 are called. By contrast, most existing database management systems implement set operations in such a way that one or both operands and/or the result of applying a set operator are usually built as temporary relations. This issue is discussed further in section 4.

Because operand relation R2 need not be a materialized relation, we refer to the technique illustrated in figure 4 as an *extended* nested-loops join. An example query for which an extended nested-loops join is the evaluation method of choice is presented in section 4.

3.2. Algorithm outline

Given a query tree, and given membership-test, selective-retrieval, and relation-producing functions for the relations named at the leaves of the tree (construction of these functions is discussed below in section 3.3), the algorithm for query evaluation using membership-test and selective-retrieval functions is the following:

Step 1: *Function construction*

(a) Make one post-order traversal of the query tree. At each internal node *n*:

(1) Build functions *n*.membership, *n*.selective-retrieval, and *n*.relation-producing.

(2) Compute an estimate of the cost of invoking each function.

(3) Compute an estimate of the sizes of the relations returned by the selective-retrieval and relation-producing functions.

(b) Build a relation-producing function for the root node of the query tree.

Step 2: *Evaluation*

Call the relation-producing function at the root node of the query tree.

Functions at node *n* can call any function at a child of *n* in the query tree. For example, the relation-producing function at the root of the query tree of figure 2 might call both children's relation-producing functions, or it might call its left child's relation-producing function and its right child's membership-test function.

The cost estimate for a function at node *n* is computed using the cost estimates for the functions it calls and the appropriate size estimates. When there are several possible ways to implement a function, cost estimates are used to choose the most efficient implementation. For example, consider the costs of the two possible implementations for the relation-producing function at the root of the query tree of figure 2:

Implementation method 1: Compute the set difference of the relations returned by both children's relation-producing functions.

Total Cost = cost(*left child's relation-producing function*) + cost(*right child's relation-producing function*) + a cost proportional to the size of the smaller of the two relations returned by the children's relation-producing functions to perform the subtraction.

Implementation method 2: Call the right child's membership-test function once for every tuple returned by the left child's relation-producing function.

Total Cost = cost(*left child's relation-producing function*) + size(*relation returned by left child's relation-producing function*) * cost(*right child's membership-test function*).

Estimating the sizes of intermediate relations is a very difficult problem. It follows that accurate cost estimates for a node's membership-test, selective-retrieval, and relation-producing functions may be difficult to compute. By assigning a cost of 'infinity' to all implicit relations' relation-producing functions we can avoid the selection of a disastrous evaluation strategy, one that instantiates implicit relations. Beyond that, empirical evidence is needed to determine how size estimates should be made, and how well the selection process is carried out in practice.

3.3. Membership-test and selective-retrieval functions for leaf relations

Membership-test and selective-retrieval functions must be provided for all leaf relations, both *implicit* and *explicit*. The functions for implicit relations defined on non-relational data structures are implemented as operations on the data structures. For example, the functions for the implicit ANCESTOR relation defined above in section 2 operate on the program's syntax tree. The most straightforward way to implement membership testing for the ANCESTOR relation is the following: given a potential ancestor-descendant pair $<a, d>$, start at node d and walk up the program tree to the root; if node a is found on this path return 'true', otherwise return 'false'. The time cost of a membership-test function implemented in this way is, in the worst case, proportional to the height of the program tree. Amortized cost $O(log(size\ of\ program\ tree))$ can be achieved using the *link-cut tree* data structure described in [Tarjan 1983]. This method requires that the program's syntax tree be represented in a different form than is usual, and has higher storage requirements.

The selective-retrieval function for the ANCESTOR relation, given a value d for the descendant field, walks up the program tree to the root, building a tuple $<n, d>$ for each node n along the path. Given a value a for the ancestor field, the function traverses the subtree rooted at a, building a tuple $<a, n>$ for each node n in the subtree. Both the time and space costs of selective retrieval for the ANCESTOR relation are thus proportional to the size of the answer set.

Membership-test and selective-retrieval functions must also be written for explicit relations, relations stored as sets of tuples. The methods used for membership testing and for selective retrieval, and the efficiency of the functions, depend on the storage and access methods provided for the relations. The most efficient functions can be provided for indexed relations. Less efficient functions can be written for rela-

tions stored in sorted order. Functions for a relation stored unsorted with no auxiliary access method are the least efficient; a complete scan of the relation may be required on every invocation.

3.4. Membership-test and selective-retrieval functions for internal nodes

One could implement the membership-test and selective-retrieval functions for an internal node of a query tree by building the intermediate relation represented by the node and searching for the appropriate tuples. Since the point of using membership-test and selective-retrieval functions is to *avoid* building intermediate relations, this strategy will rarely be cost-effective. In general, membership-test and selective-retrieval functions make use of similar functions found at the next level down in the query tree, which in turn make use of functions at the *next* level down, and so on, until we reach the leaves of the tree and the functions associated with the leaf relations.

The particular membership-test and selective-retrieval functions built for an internal node of a query tree depend on which relational operator is found at that node. Sometimes there are several ways to implement the functions. The selective-retrieval function for an INTERSECTION node n, with operands R1 and R2, is a good example; one could implement n.selective-retrieval in any of the following ways:

(1) Given parameters f (a list of fields) and v (a list of values), n.selective-retrieval returns the intersection of R1.selective-retrieval(f, v) and R2.selective-retrieval(f, v).

(2) For every tuple t returned by R1.selective-retrieval(f, v), n.selective-retrieval calls R2.membership(t); t is in the set returned by n.selective-retrieval iff R2.membership returns 'true'.

(3) Same as (2), reversing the roles of R1 and R2.

For each instance of an INTERSECTION node, the implementation with the lowest cost estimate is chosen.

In the absence of PROJECT nodes, the design of membership-test functions for the internal nodes of a query tree is quite straightforward. For example, the membership-test function for an INTERSECTION node n with operands R1 and R2, given parameter t, returns the conjunction of R1.membership(t) and R2.membership(t):

```
n.membership(t) {
   if (R1.membership(t))
     then return(R2.membership(t));
     else return(false);
   }
```

PROJECT nodes complicate membership-test functions because the arity of the relation represented by a PROJECT node is smaller than the arity of its operand relation. We want to build a membership-test function f for the PROJECT node that invokes its child's membership-test function f'; however, f cannot pass to f' values for fields that are not included in the PROJECT node's list of projected fields. We allow f to pass a special ''wildcard'' value, denoted by '$*$', to f' for every field not included in the list of projected fields. Membership-test functions must thus be prepared to handle tuples that contain wildcard values.

In some cases, SELECT nodes can replace wildcard values with constant values. This situation can be illustrated by considering the example query of figure 2. The root node of this query tree is a MINUS node. The relation-producing function for this node can be implemented by calling the relation-producing function of its left child, and the membership-test function of its right child.

The membership-test function of the right-hand-side PROJECT node is called by the relation-producing function of the MINUS node with a tuple of the form <d>, because the arity of the relation represented by both the MINUS node and the PROJECT node is 1.

The PROJECT node's membership-test function calls the membership-test function of the right-hand-side SELECT node with a tuple of the form <$*$, d>, because the arity of the relation represented by the SELECT node is 2. The wildcard value is in the first field of the tuple because the projected field, 'descendant', is the second field of the relation represented by the SELECT node.

The SELECT node's membership-test function calls the membership-test function of the ANCESTOR relation with a tuple of the form <B, d>. The SELECT node's membership-test function is able to replace the wildcard value in the first field with a constant value because a tuple cannot be in the relation represented by the SELECT node unless it has the value B in the first field.

The above example illustrates a particularly fortuitous situation; the wildcard value passed down by the PROJECT node is immediately replaced with a constant value by the SELECT node. This will not be the case in general; thus, membership-test functions must be written to handle arguments containing wildcard values. For example, in the presence of wildcard values, the membership-test function for an INTERSECTION node cannot simply return the conjunction of its children's membership-test functions. Figure 5 illustrates a situation in which doing so would lead to erroneous results. If the INTERSECTION node's membership-test function is called with a tuple of the form <$*$, 1> and calls its children's membership-test

functions with the same tuple, they will each return 'true'. R1.membership returns 'true' because it includes tuple <0, 1>, and R2.membership returns true because it includes tuple <1, 1>, yet there is no tuple in the relation represented by the INTERSECTION node with a 1 in the second field.

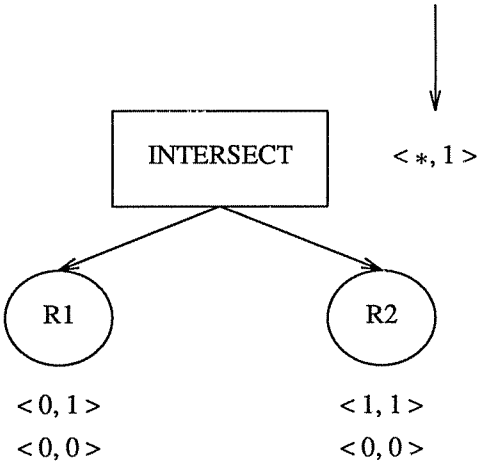


Figure 5
Why an INTERSECTION node's membership-test function must change
in the presence of wildcard values

When there are wildcard values in the given tuple *t*, the INTERSECTION node's membership-test function must be implemented in one of the following ways:

Implementation method 1

Step 1:
Call one child's selective-retrieval function, passing as arguments the list of fields in *t* that do not have wildcard values and the corresponding list of values.

Step 2:
For every tuple *u* returned by Step 1, call the other child's membership-test function with argument *u*. As soon as the membership-test function returns 'true', the membership-test function for the INTERSECTION node returns 'true'. If the membership-test function never returns 'true', the membership-test function for the INTERSECTION node returns 'false'.

Implementation method 2

Step 1:
Call both children's selective-retrieval functions, passing as arguments the list of fields in *t* that do not have wildcard values and the corresponding list of values.

Step 2:
Compute the intersection of the two sets returned by step 1. If the intersection is non-empty, the INTERSECTION node's membership-test function returns 'true', otherwise it returns 'false'.

The relative costs of the node's children's membership-test and selective-retrieval functions, and the size estimates for the relations returned by the selective-retrieval functions determine which of the above methods is used.

To summarize:

- Membership-test, selective-retrieval, and relation-producing functions are built for all internal nodes of the query tree. The functions associated with node n can call functions associated with any child of n.
- A cost estimate is computed for each function; when there is more than one possible implementation for a particular function the implementation with the lowest cost is selected.
- A naive approach to writing membership-test functions can fail when the query contains PROJECT nodes. Example problems and solutions were outlined above; a complete treatment of this issue is found in [Horwitz 1985].

4. Generality of the method

In this section we introduce a new example query to illustrate the potential advantages of our query-evaluation method over existing methods in the context of queries that use only explicit relations. We expect our method to be most advantageous in the evaluation of non-trivial queries involving set operators. In general, the larger the query tree and the greater the number of set operators, the more beneficial our method will be. This is because our method treats all of the relational operators, including the set operators, in a uniform fashion, thus allowing the optimization of the *entire* query, rather than optimizing portions of the query independently of each other.

By contrast, the SQL query optimizer considers the operands of a UNION separately [Lohman 1986], and arranges for subqueries to be fully evaluated before the top-level query [Selinger et al 1979]. Users of the QUEL language can implement set union and set difference by building one operand relation and inserting or deleting the tuples of the other operand relation. This separation of a query into two parts clearly leads to lost opportunities for optimization. Set union can also be implemented using "or", and set difference can be implemented using aggregates. However, it is unclear how likely naive users are to avail themselves of these possibilities (especially the latter), or how well the QUEL optimizer handles such queries.

The example query discussed below uses the following (explicit) relations:

PARTS(Part#: *integer*, ComponentOf: *string*)

SUPPLIERS(SName: *string* , Part#: *integer*)

We wish to retrieve the numbers of all parts that are components of ''tractor'', and are supplied by either ''Smith'' or ''Jones'' but not both of them. A straightforward translation of this query to tree form yields the query tree shown below in figure 6a. Figure 6b gives a higher-level view of the same query, substituting English explanations for some sub-queries.

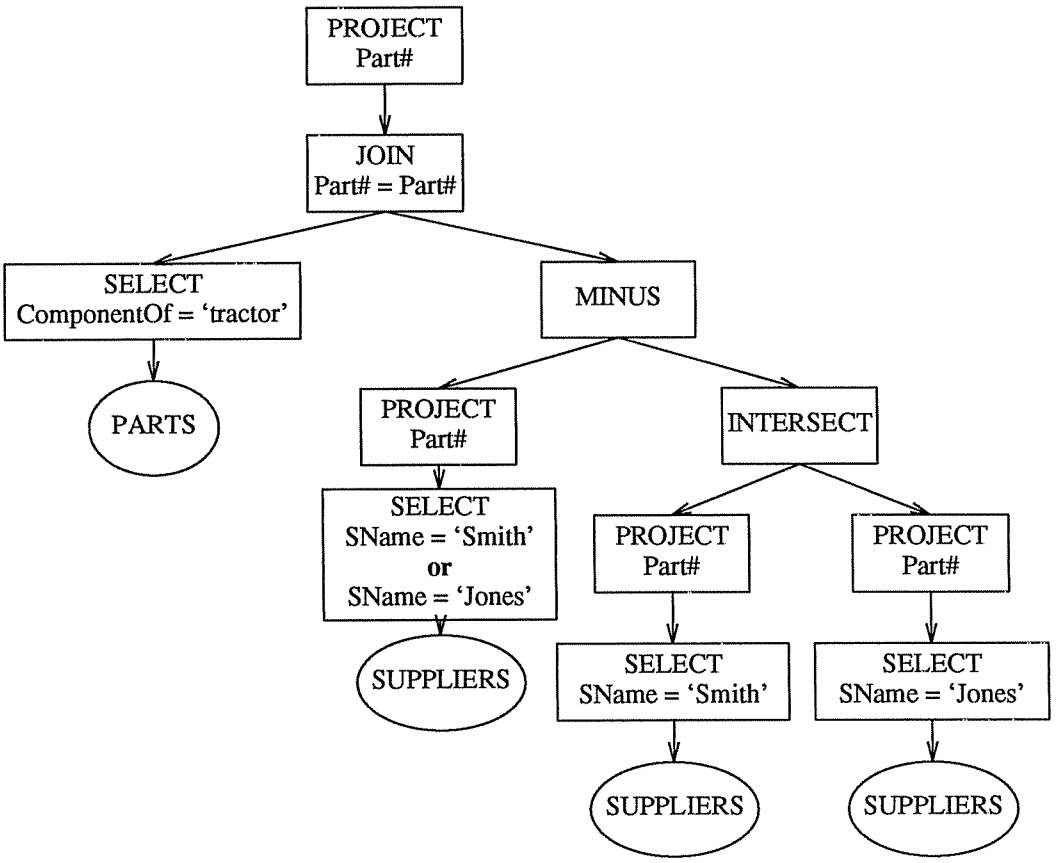


Figure 6a
Example query using only explicit relations
All tractor components supplied by Smith or Jones but not both

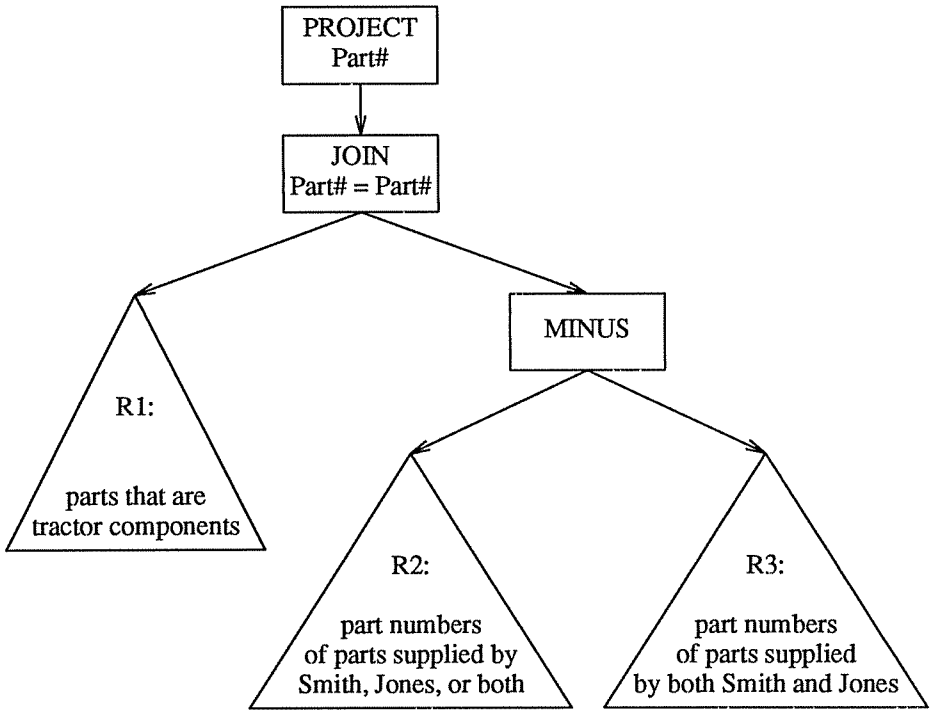


Figure 6b
Simplified, "English" version of example query of figure 6a

Assuming that the number of tractor components is small compared to the number of parts supplied by either Smith or Jones, and that there are indexes on the "ComponentOf" field of the PARTS relation and the "Part#" field of the SUPPLIERS relation, the best way to evaluate this query is to use an extended nested-loops join (introduced in section 3.1):

(1) Tractor components are extracted from the PARTS relation using the index on the ComponentOf field.

(2) For each tuple *t* extracted in step (1), a lookup is done on the relation represented by the MINUS node. If a tuple with the appropriate part number is found, it is joined with *t*.

(3) The Part# field of each tuple created in step (2) is stored in the result relation.

The lookup in step (2) is performed on a non-materialized relation; this is the *extended* part of the extended nested-loops join. Because the non-materialized relation is defined using *minus*, standard query evaluators would probably not choose extended nested-loops join to evaluate this query. By contrast, the evaluation strategy that would be selected by applying our method to the query tree of figure 6 is exactly the extended nested-loops join described above.

Using our query-evaluation method, the lookup of step (2) is implemented using a call to the MINUS node's selective-retrieval function, asking for all tuples that have the given part number in field number one. Because the relation represented by the MINUS node has only one field, its selective-retrieval function is equivalent to its membership-test function; a call to MINUS.membership leads to calls to the MINUS node's children's membership-test functions, eventually leading to calls on SUPPLIERS.membership, which are implemented using the index on the "Part#" field.

The cost of step (1) is proportional to the number of tractor components; the cost of step (2) is proportional to the number of tractor components times the cost of a lookup on the SUPPLIERS relation. A lookup on the SUPPLIERS relation (using the Part# index) is proportional to the number of suppliers that supply the given part. In the worst case, when all suppliers supply all parts, the cost of a lookup is proportional to the number of suppliers. The worst-case overall cost of the extended nested-loops join is thus O((# of tractor components)* (# of suppliers)).

Given a multi-attribute or multi-dimensional index on the SUPPLIERS relation, the cost of SUPPLIERS.membership is independent of the number of suppliers supplying each part, and the overall cost of the extended nested-loops join is just O(# of tractor components). Note also that by using pipelining, the extended nested-loops join can be performed without materializing *any* intermediate relations.

How does the performance of extended nested-loops join compare to a more traditional evaluation method in which the relation represented by the MINUS node is materialized and then joined with the set of tractor components? To answer this question without getting bogged down in the details of exactly how the relation represented by the MINUS node would be computed, note that the computation of the MINUS relation requires the computation of the relation labeled "R2" in figure 6b. Computing relation R2 without an index on the SName field of the SUPPLIERS relation requires at least one complete scan of the SUPPLIERS relation, making this a far more expensive approach than the extended nested-loops join. Given an index on the SName field of the SUPPLIERS relation, relation R2 can be computed more efficiently, and a more detailed analysis would be needed to determine the relative merits of the extended nested-loops join approach, and the approach in which the MINUS relation is materialized. Note however, that materializing the MINUS relation has two inherent disadvantages as compared to the extended nested-loops join:

(1) A multi-attribute or multi-dimensional index on the SUPPLIERS relation cannot be fully utilized since information about the part numbers of tractor components is not available within the MINUS sub-query.

(2) Pipelining cannot be fully utilized to prevent the materialization of intermediate relations since the approach forces the materialization of at least one intermediate value, namely the MINUS relation itself.

5. Relation to previous work

The major contributions of this paper are the definition of implicit relations and the design of a new query-evaluation method. In this section we compare these two aspects of our work with relevant previous work.

The concept of implicit relations has some similarities with the concept of *views* [Ullman 1980 pp. 5-9]. In both cases, the user of a database is given access to relations that are not actually stored as sets of tuples. View relations and implicit relations differ in how they can be defined, and in their intended use.

A view relation is defined by a query; conceptually, the query is re-evaluated after every modification to the database, and the view relation is the result of the most recent evaluation. View definitions are limited to relational operators applied to explicit or view relations.

Implicit relations are defined by three access functions: membership-test, selective-retrieval, and relation-producing, which can be defined by applying *arbitrary operators* to *arbitrary data structures*. While the example implicit relations presented in this paper have been defined in terms of non-relational operations on non-relational data structures, it is equally reasonable to define implicit relations in terms of operations (relational or not) on explicit relations. Thus, implicit relations can be viewed as a generalization of view relations.

We turn next to a comparison of our approach to query evaluation with relevant previous work in that area. The goals of our query-evaluation method are:

(1) Avoid materializing implicit relations whenever possible.

(2) Avoid materializing intermediate relations whenever possible.

(3) Achieve goals (1) and (2) in the presence of set operators as well as select, project, and join.

Having just considered the similarities between view relations and implicit relations, one is led to ask whether there is an analog to our first goal, avoiding the materialization of implicit relations, in the context

of queries that use view relations. The answer is that query-evaluators do try to avoid the materialization of view relations used in queries. Because view relations are defined as relational operations on explicit relations, the mechanism for avoiding the materialization of view relations is straightforward: the view definition is inserted into the query in place of the view relation, and the entire query is optimized [Stonebraker 1975].

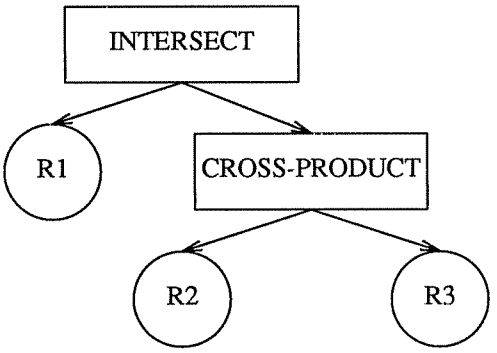
Of course, this approach won't work when an implicit relation is used in a query because implicit relations may not be defined in terms of relational operations on explicit relations. Thus, while the goal of avoiding the materialization of implicit relations is similar to the goal of avoiding the materialization of view relations, the methods used to achieve that goal must differ. Our approach to avoiding the materialization of implicit relations is to use the membership-test and selective-retrieval functions provided for these relations in place of the relations themselves.

Membership-test and selective-retrieval functions are also used to achieve our second goal, avoiding the materialization of *intermediate* relations. This can be viewed as a combination and generalization of the methods of [Wong and Youssefi 1976] and [Liu 1979].

Our use of selective-retrieval functions to evaluate joins corresponds to "tuple substitution" as presented in [Wong and Youssefi 1976]. Our method can be considered to be a generalization of theirs because we are able to do lookups on non-materialized operand relations and because we allow the operands of joins to be defined using set operators, while they only consider queries defined using selection, projection, and join. Further, our method is more flexible in that the use of selective-retrieval is only one possible way to implement a JOIN node's relation-producing function. Other possibilities, for example, calling both operands' relation-producing functions followed by a sort-merge join [Selinger et al 1979], can be considered as well.

[Liu 1979] proposes the use of membership tests for the evaluation of some set operators. However, because Liu was concerned with optimizing the execution of set-oriented programming languages, he considers *only* the set operators, and not selection, projection, or join. While it is rather trivial to build membership-test functions for all nodes of a query tree that contains only set operators, it is not obvious how to do so for a query tree that includes selection, projection, and join. Some of the subtleties that arise in the latter case were mentioned briefly in section 3.4; a complete treatment appears in [Horwitz 1985].

Other approaches to query optimization that seek to avoid building intermediate relations are the use of pipelining [Smith and Chang 1975] [Yao 1979] [Lu and Carey 1985] and of tree transformations [Smith and Chang 1975] [Hall 1976]. When pipelining is used, tuples are passed up the tree as they are computed rather than waiting until an entire intermediate relation is formed. Pipelining is thus similar to our query-evaluation method in that intermediate relations are not materialized. It is dissimilar in that, when pipelining is used, the tuples of intermediate relations are all computed, whereas our method seeks to avoid or reduce the computation (as well as the materialization) of intermediate values. To understand this difference, consider the following query:



One way to evaluate this query without materializing the temporary relation represented by the CROSS-PRODUCT node is to use pipelining. The cross product is computed a tuple-at-a-time, and each tuple is passed up the query tree. The tuple is in the result relation if it is in R1. While no temporary relation is materialized using this approach, all tuples of the cross product are computed; thus, the cost of the evaluation is at least proportional to the size of the cross product.

By contrast, using our method the query can be evaluated without materializing the temporary relation, and without computing *any* tuples of the cross product. Instead, a scan is made through relation R1, each tuple t is divided into two tuples, t_1 and t_2, according to the arities of R2 and R3, and lookups are done in relations R2 and R3 on t_1 and t_2 respectively. Tuple t is in the result relation if t_1 is in R2 and t_2 is in R3. The cost of this evaluation is thus independent of the size of the cross product.

Of course, other factors, for example, the existence of indexes, affect the over-all cost of evaluating the query. This example is merely meant to illustrate the philosophical difference between our approach and pipelining, and to indicate the potential advantages of our method. Further, the two approaches are not

incompatible; pipelining can be incorporated into our approach by having selective-retrieval and relation-producing functions use pipelining rather than returning entire relations.

Tree transformations such as combining sequences of projections into a single projection and combining sequences of selections into a single selection, can reduce the number of intermediate relations in the query tree. Moving selection operators ahead of construction operators can reduce the sizes of the intermediate relations represented by the internal nodes of the query tree. These techniques do not, however, address the question of whether one can avoid building some of the intermediate relations of the transformed tree.

As with pipelining, incorporating tree transformations into our method will be important to its success. A query can often be represented using a number of different trees, and a different relation-producing function will be built for the root node of each such tree. It is important that transformations be used to produce trees in which the cost of the root node's relation-producing function is minimized.

Similarly, other optimization techniques could be used to produce more efficient membership-test, selective-retrieval, and relation-producing functions. Exploiting idempotency and unsatisfiability to simplify *conditions* [Hall 1976] [Eswaran et al 1976] can lead to more efficient membership-test and relation-producing functions; using the methods of [Astrahan and Chamberlin 1975] [Blasgen and Eswaran 1977] [Selinger et al 1979] [Yao 1979] for access path selection, the methods of [Smith and Chang 1975] for choosing sort orders, and the methods of [Gotlieb 1975] [Wong and Youssefi 1976] for calculating join orders can all lead to more efficient relation-producing functions.

Our third goal, the efficient evaluation of queries that include set operations, has been all-but ignored in past work. Two exceptions are [Shaw 1985] and [Chu and Hurley 1982]. The work presented in [Shaw 1985] deals with query processing on a non-standard machine designed to support the execution of relational operations, and is thus not comparable with the work presented here. [Chu and Hurley 1982] discuss query processing for distributed databases; their model of query evaluation is one in which intermediate values are computed and passed among a set of processors. Thus, while they share our goal of handling set operators, they do not share our goal of avoiding the materialization of intermediate relations, and the two approaches have little in common.

While we recognize the importance of select-project-join queries, we feel that the set operators deserve consideration, too. We believe that there are queries for which the most natural formulation involves the use of set intersection, union, and difference; therefore the efficient evaluation of queries that use set operators is a reasonable goal.

6. Summary

The desire to allow interactive software systems to include relational query facilities without giving up existing, non-relational data structures led to the design of *implicit relations*. Information contained in the non-relational data structures is *conceptually* stored as sets of tuples, and can be accessed by using implicit relations in queries.

The use of implicit relations in queries requires a new approach to query evaluation. The method described here is a uniform approach based on the use of three access functions: membership-test, selective-retrieval, and relation-producing. Access functions for implicit relations are defined as operations on the appropriate non-relational data structures; access functions for normal or *explicit* relations are defined according to the access methods provided for the relations.

While the new query-evaluation method *must* be used on queries that include implicit relations, it may prove superior to existing methods when applied to queries that include only explicit relations as well. In general, we expect our method to be most advantageous in the evaluation of queries that make heavy use of set operations. Further investigation is needed to define the class of applications to which our method is well-suited, and to determine how well it performs in practice.

References

[Astrahan and Chamberlin 1975]
Astrahan, M.M. and Chamberlin, D.D. Implementation of a structured English query language. *Communications of the ACM Vol. 18 No. 10* (Oct 1975) 580-588

[Blasgen and Eswaran 1977]
Blasgen, M.W. and Eswaran, K.P. Storage and access in relational databases. *IBM Systems Journal 16* (1977) 363-377

[Chu and Hurley 1982]
Chu, W.C. and Hurley, P. Optimal query processing for distributed database systems. *IEEE Transactions on Computers Vol. c-31, No. 9* (Sept. 1982) 835-850

[Codd 1970]
Codd, E.F. A relational model of data for large shared data banks. *Communications of the ACM Vol. 13 No. 6* (June 1970) 377-387

[Eswaran et al 1976]
Eswaran, K.P. Gray, J.N. Lorie, R.A. Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM Vol. 19 No. 11* (1976) 624-633

[Gotlieb 1975]
Gotlieb, L. R. Computing joins of relations. *ACM SIGMOD International Conference on Management of Data* (May 1975) San Jose, CA, 55-63

[Hall 1976]
Hall, P. A. V. Optimization of single expressions in a relational data base system. *IBM J. Res. Develop. Vol. 20* (May 1976) 244-237

[Horwitz 1985]
Horwitz, S. Generating language-based editors: a relationally-attributed approach. *PhD thesis.* Cornell University (August, 1985)

[Linton 1984]
Linton, M. Implementing relational views of programs. *Proc. of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments* (April 1984) Pittsburgh, PA, 132-140

[Liu 1979]
Liu, L. Essential uses of expressions in set-oriented expressions. *Ph.D. Thesis.* Cornell University (May 1979)

[Lohman 1986]
Lohman, G.M. Do semantically equivalent SQL queries perform differently? *IEEE 1986 International Conference on Data Engineering* (Feb. 1986) Los Angeles, CA, 225-226

[Lu and Carey 1985]
Lu, Hongjun and Carey, M.J. Some experimental results on distributed join algorithms in a local network. *Proceedings of VLDB 85* (1985) Stockholm, 292-304

[Selinger et al 1979]
Selinger, P.G. Astrahan, M.M. Chamberlin, D.D. Lorie, R.A. and Price T.G. Access path selection in a relational database management system. *ACM SIGMOD International Conference on Management of Data* (May-June 1979) Boston, MA, 23-34

[Shaw 1985]
Shaw, D.E. Relational query processing on the Non-Von supercomputer. From: *Query Processing in Database Systems* W. Kim, D. Reiner, D. Batory, eds. Springer-Verlag (1985)

[Smith and Chang 1975]
Smith, J.M. and Chang, P. Optimizing the performance of a relational algebra database interface. *Communications of the ACM Vol. 18 No. 10* (Oct. 1975) 568-579

[Stonebraker 1975]
Stonebraker, M. Implementation of integrity constraints and views by query modification. *ACM SIGMOD International Conference on Management of Data* (May 1975) San Jose, CA, 65-78

[Stonebraker et al 1976]
Stonebraker, M., Wong, E., and Kreps, P. The design and implementation of INGRES. *ACM Trans. on Database Systems Vol. 1 No. 3* (Sept. 1976) 189-222

[Tarjan 1983]
Tarjan, R.E. *Data Structures and Network Algorithms.* Society for industrial and applied mathematics Philadelphia, PA (1983)

[Ullman 1980]
Ullman, J. *Principles of Database Systems.* Computer Science Press Potomac, MD. (1980)

[Wong and Youssefi 1976]
Wong, E. and Youssefi, K. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems Vol. 1 No. 3* (Sept. 1976) 223-241

[Yao 1979]
Yao, S.B. Optimization of query evaluation algorithms. *ACM Transactions on Database Systems Vol. 4 No. 2* (1979) 133-155