Computing Queries from Derived Relations*

P.-Å. Larson and *H.Z. Yang*[†]

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1 Canada

Abstract. Assume that a set of derived relations are available in stored form. Given a query (or subquery), can it be computed from the derived relations and, if so, how? Variants of this problem arise in several areas of query processing. Relation fragments stored at a site in a distributed database system, database snapshots, and intermediate results obtained during the processing of a query are all examples of stored, derived relations. In this paper we give necessary and sufficient conditions for when a query is computable from a single derived relation. It is assumed that both the query and the derived relation are defined by *PSJ*-expressions, that is, relational algebra expressions involving only projections, selections, and joins, in any combination. The solution is constructive: not only does it tell whether the query is computable or not, but it also shows how to compute it.

1. INTRODUCTION

Consider a database consisting of a number of (conceptual) relations $R_1, R_2, \ldots, R_m$, and assume that the extensions of the conceptual relations are not available in stored form. Instead we have available, in stored form, a set of n derived relations, defined by $E_1, E_2, \ldots, E_n$ where each E_i is some expression in relational algebra. We are given a query E_0, that is, a relational algebra expression over $R_1, R_2, \ldots, R_m$. The problem is then the following: Can E_0 be computed from the available, stored relations defined by $E_1, E_2, \ldots, E_n$ and, if so, how?

This paper represents a first step towards the solution of this problem. We give necessary and sufficient conditions for deciding whether E_0 can be computed from a single derived relation E_1, for the case when E_0 and E_1 are both *PSJ*-expressions. A relational algebra expression is a *PSJ*-expression if it involves only the operations project, select, join and Cartesian product. This may seem a very restricted class of expressions, but that is not the case. *PSJ*-queries (queries that can be represented by a *PSJ*-expression) are extremely common in relational database systems. Whatever the user query language, almost every query is either a *PSJ*-query or has one or more subqueries which are *PSJ*-queries.

Different variants of this problem occur in several areas of query processing. The differences arise from different interpretations of stored, derived relations. In a distributed database context, derived relations can be interpreted as relation fragments stored at a site (or group of sites). This variant of the problem has been studied extensively, but normally under the assumption that each fragment is derived from a single relation using only selections and projections [CP84]. Database snapshots are another example of stored, derived relations [AL80].

The solution to this problem also has applications in "traditional" query optimization [MA83,UL82]. Here a derived relation can be interpreted as an intermediate result obtained in the process of computing a query. If some other part of the same query can be "easily" computed from intermediate results $E_1, E_2, \ldots, E_n$, query processing may be speeded up. It may even be worthwhile retaining certain intermediate results, if the chance that they can be used in subsequent queries is high enough to warrant the extra storage space [FS82]. In this context the problem has been studied (usually) under the restriction that E_0 must exactly match one of the expressions $E_1, E_2, \ldots, E_n$.

Our main motivation for studying this problem stems from a different area: physical database design for relational databases. In current relational systems

* This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant No. A-2460.

[†] **Electronic mail:**
UUCP:
{decvax,ihnp4,allegra}!watmath!watdaisy!{palarson,hzyang}
CSNET: {palarson,hzyang}%watdaisy@waterloo.csnet

the structure of the stored database is normally required to be in one-to-one correspondence with the conceptual schema. By this we mean that each conceptual relation exists as a separate stored relation (file). To speed up query processing some auxiliary access structures may be added: secondary indexes, join indexes, record linking, etc. [AS76,BL81,SW76]. This way of structuring the stored database is a simple and straightforward solution. However, it has the effect that the processing of a query almost always requires data to be collected from several physical files.

Instead of directly storing each conceptual relation, we suggest a more flexible approach where the physical database is structured as a set of stored, derived relations. The choice of stored relations should be guided by the (actual or anticipated) query load, so that frequently occurring queries can be processed rapidly. If it is advantageous to do so, some data may be redundantly stored in several physical relations. The structure of the stored database, and changes to the structure, should be completely transparent at the user level, and user queries and updates expressed solely in terms of conceptual relations. The system must be capable of automatically transforming a user query into an equivalent query against stored relations, and a user update to updates of (one or more) stored relations. To make the suggested approach viable the two fundamental problems of query transformation and update transformation must be solved. This paper is a first contribution towards the solution of the query transformation problem.

The idea of not necessarily having a one-to-one correspondence between conceptual relations and stored relations has so far received little attention (outside the area of distributed databases). Joining relations is often the most costly operation in query processing. Schkolnick and Sorensen [SS81] studied the effects of storing relations in prejoined form (called denormalization). Motivated by the availability of special hardware capable of rapid selection and projection, Babb proposed a "Joined Normal Form" [BA82]. It amounts to storing the whole database as one single relation, which is the join of all conceptual relations. Roussopoulos [RO82] originated the idea of view indexing. A view index represents a materialized user view, containing pointers to the tuples contained in the view. A view index is essentially an indirect version of a derived relation.

A classical problem related to the one studied here is that of query containment: given two queries defined by relational expressions E_0 and E_1, is the result of E_0 a subset of the result of E_1? This problem, and the closely related problem of query equivalence, have been solved for certain classes of queries: conjunctive queries and tableau queries. For an overview and references, see the chapters on query optimization in [MA83] or [UL82]. Note, however, that the problem studied here is *not* equivalent to that of query containment. In addition to containment, we impose the further requirement that it must be possible to compute the result of E_0 from the result of E_1.

2. NOTATION AND BASIC ASSUMPTIONS

A **relation scheme** is a list of attribute names. A **database scheme** is a set of relation schemes. An instance (or extension) of a relation is a set of mappings which maps a set of attribute names to values or, equivalently, a set of tuples. An instance (extension) of a database is a collection of instances of all relations in the database schema. When there is no risk of confusion we will use the shorter "relation" instead of "instance of a relation", and correspondingly for database.

Derived relations and queries are here expressed in relation algebra. The following notation will be used:

$R \times S$	Cartesian product of relations R and S.
$R\{C\}$	Select all tuples from relation R satisfying condition C (a boolean expression).
$R[A_1, ..., A_m]$	Project relation R onto attributes $A_1, ..., A_m$.
$R \mid C \mid S$	Join relations R and S over condition C, that is, $R \mid C \mid S = (R \times S)\{C\}$.
$\alpha(C)$	The set of all attributes appearing in condition C.

The logical connectives will be denoted by $+$ for OR, juxtaposition (multiplication) for AND and $\neg$ for NOT. A tuple t satisfies a condition C, if C evaluates to true when each attribute name in C is replaced by the corresponding attribute value from t. An instance of a relation satisfies C if all tuples in the relation satisfy C. Projections and selections are assumed to have higher precedence than binary operations.

A relational algebra expression has relation names as operands and relational algebra operators as operators. If each relation name is replaced by a corresponding instance the expression is evaluated and the result is a derived relation. The result of evaluating an expression E over an instance d of database

$D = \{R_1, R_2, ..., R_n\}$ will be denoted by $V(E,d)$. An expression is considered to be defined over the full database scheme, even though the expression may only refer to a subset of the relations in the scheme.

We state the following without proof: Every (valid) expression constructed from an arbitrary number of Cartesian products, joins, selections and projections (*PSJ*-expression) can always be transformed to an equivalent expression in a standard form consisting of a Cartesian product, followed by a selection, followed by a final projection. One can easily see that this holds by considering the query tree corresponding to a *PSJ*-expression. The standard form is obtained by first pushing all projections to the root of the tree and thereafter all selection and join conditions.

From the above it follows that any *PSJ*-expression E can be written in the form $E = \{R_1 \times R_2 \times \cdots \times R_k\} \{C\} [A_1, A_2, ..., A_l]$ where $R_1, R_2, ..., R_k$ are relations, C is a selection condition and $A_1, A_2, ..., A_l$ are the attributes of the final projection. We can therefore represent any *PSJ*-expression by a triple $E = (\mathbf{A}, \mathbf{R}, C)$ where $\mathbf{A} = \{A_1, A_2, ..., A_l\}$ is called the attribute set, $\mathbf{R} = \{R_1, R_2, ..., R_k\}$ the relation set or base, and C the selection condition.

Example: Consider the following expression defined over relations $R(A,B,C,D)$ and $S(E,F)$:

$$((R[A,B]\ \{A>10\})\ |B=F|\ (S\{E>20\}))\ [A,F]$$
$$= (R \times S)\ \{(A>10)\ (B=F)\ (E>20)\}\ [A,F]$$
$$= (\{A,F\}, \{R,S\}, (A>10)\ (B=F)\ (E>20))\ .$$

The triple representation inherits the naming problem of relational algebra: if a relation R occurs more than once in the relation set, then a reference to one of its attributes in C or $\mathbf{A}$ is ambiguous. The ambiguity can be resolved by appropriate renaming of repeated relation. This problem will be discussed later, but until further notice we will assume that no relation occurs more than once in the relation set of an expression.

3. DERIVABILITY AND COVERAGE

A user query is expressed in terms of conceptual relations. Query transformation consists of transforming a user query into an equivalent query against stored relations. This process should be done automatically by the system and be completely transparent at the user level. Once an equivalent query has been constructed, the task of "optimizing" its evaluation still remains. This problem has been studied extensively and all the known techniques apply [MA83,UL82]. We are not at this stage concerned with constructing efficient queries; the only goal at this stage is to be able to correctly transform queries.

Both stored, derived relations and queries are represented by relational algebra expressions. Stated in its most general form, query transformation amounts to finding a relational algebra expression $F(E_1, E_2, ..., E_n)$ which generates the same result as a given query E_0, for every instance of the database. Solving the problem in its general form appears extremely hard. We will reduce it to a more manageable level by imposing a certain restriction on the deriving expression F. The essence of the restriction is that "self-joins" are not allowed in F. A "self-join" is an expression of the type $E_k\,|\,C\,|\,E_k$ or $E_k \times E_k\ \{C\}$. In our opinion this is not a severe restriction; "self-joins" appear to be rare and of limited use.

Definition. (Derivability) Consider a set of relational algebra expressions $\{E_0, E_1, ..., E_n\}$ defined over a database schema $D = \{R_1, R_2, ..., R_m\}$, and let $\{E_{i_1}, E_{i_2}, ..., E_{i_k}\}$, $k \le n$, be a subset of $\{E_1, E_2, ..., E_n\}$. We say that E_0 is **(algebraically) derivable** from $\{E_1, E_2, ..., E_n\}$, over the base $E_{i_1}, E_{i_2}, ..., E_{i_k}$, if there exists a relational algebra expression F, containing no joins or Cartesian products, such that $V(E_0, d) = V(F(E_{i_1} \times E_{i_2} \times \cdots \times E_{i_k}), d)$ for every instance d of the database D. □

Note that the definition does not completely prohibit joins or Cartesian products; it only forbids "self-joins". There are no restrictions on the use of projections, selections and the standard set operations. An output tuple of F may be constructed by combining any tuple from E_j with any tuple from E_l provided that $j \neq l$. However, combining a tuple from E_j with another one from E_j is not allowed. Note also that F may not be unique. If the database contains redundant data, there may be many different ways of transforming the query.

If the relations defined by $E_1, E_2, ..., E_n$ do not contain "sufficient" data to construct E_0, no deriving expression F can exist. If we can show that "sufficient" data exist, then we can proceed to the more difficult problem of constructing a deriving expression F. Saying that $E_1, E_2, ..., E_n$ contains "sufficient" data vaguely means that "all the data values of E_0 can be found in $E_1, E_2, ..., E_n$ ". More precisely it means that any value (combination of values) of an attribute (set of attributes) found in a tuple in E_0 can also be found, under the same attributes, by taking some combination of tuples from $E_1, E_2, ..., E_n$. Consider the following example:

Example: E_0 is defined over R_1 and R_2, and its attributes are $[R_1.A, R_1.B, R_2.C]$. Assume that a tuple (10, 5, -3) exists in E_0 for some instance of R_1 and

R_2. If the value 10 is not contained in any projection of $E_1, E_2, ..., E_n$ over $R_1.A$ for the same instance of R_1 and R_2, E_0 cannot be derived from $E_1, E_2, ..., E_n$. The same holds if the tuple (10, 5) cannot be found in any projection over $[R_1.A, R_1.B]$. □

This idea is formalized in the following two definitions.

Definition. Let $r = (a_1, a_2, ..., a_m)$ and $s = (b_1, b_2, ..., b_n)$, $m \leq n$, be two tuples of relations $R(A_1, A_2, ..., A_m)$ and $S(B_1, B_2, ..., B_n)$, respectively. Tuple r is said to be a **subtuple** of s if for every a_i, $1 \leq i \leq m$, there exists an attribute j, $1 \leq j \leq n$, such that $a_i = b_j$ and $A_i = B_j$, and no two attributes in r map to the same attribute in s. □

Definition. (Coverage) Let $\{E_0, E_1, E_2, ..., E_n\}$ be a set of relational expressions defined over a common database schema $D = \{R_1, R_2, ..., R_m\}$ and let $\{E_{i_1}, E_{i_2}, ..., E_{i_k}\}$ be a subset of $\{E_1, E_2, ..., E_n\}$. We say that E_0, is **covered** by $E_1, E_2, ..., E_n$, over the base $\{E_{i_1}, E_{i_2}, ..., E_{i_k}\}$, if for every instance d of D and every tuple $t_0 \in V(E_0, d)$ there exists a tuple $t \in V(E_{i_1} \times E_{i_2} \times \cdots \times E_{i_k}, d)$ such that t_0 is a subtuple of t. □

It is easy to see that derivability implies coverage. Coverage is a necessary, but not sufficient, condition for derivability. Testing coverage is much easier than testing derivability as we will see. If the coverage test fails, there is no need to consider derivability.

4. TESTING THE VALIDITY OF BOOLEAN EXPRESSIONS

We will in the sequel frequently need an algorithm for testing whether a given boolean expression is valid, that is, always evaluates to true. Such an algorithm is developed in this section, for formulas containing no arithmetic expressions. It is based on the same idea as an algorithm given by Rosenkrantz and Hunt [RH80]. Our algorithm is faster but more restricted than theirs.

We will consider boolean expressions constructed from variables, constants, comparison operators $(<, =, >)$, boolean connectives and parentheses. An **atomic conditions** is an expression of the form $x\,cop\,y$ where x is a variable, cop is one of $'<', '=',$ or $'>'$ and y is either a variable or a constant. If y is a constant, the condition is said to be a **simple condition**, otherwise a **connective condition**. Negation is not needed; any expression containing negation can be reduced to one without negation. Without loss of generality, all variables are assumed to be defined over some finite set of integer values. Variables correspond to attributes and in practice all attributes have a discrete, finite domain. Any discrete, finite domain can be mapped into a finite set of integer values.

Let $C(x_1, x_2, ..., x_n)$ be a boolean expression, as explained above, with variables $x_1, x_2, ..., x_n$, that is, a formula in first order logic. A formula is valid if $\forall x_1 \forall x_2 \cdots \forall x_n (C(x_1, x_2, ..., x_n))$ holds, that is, if it evaluates to true for all possible values of $x_1, x_2, ..., x_n$. Every x_i, $i = 1, 2, ..., n$ is restricted to its domain, of course. When there is no risk of confusion we will use the shorter notation $\forall x_1, x_2, ..., x_n(C)$.

From elementary logic we know that $\forall x_1, x_2 \cdots x_n(C) \Leftrightarrow \nexists x_1, x_2 \cdots x_n(\neg C)$. Now expand $\neg C$ into disjunctive form, that is, let $\neg C = B_1 + B_2 + \cdots + B_m$ where each B_i is of the form $B_i = B_{i1} B_{i2} \cdots B_{ik_i}$ and each B_{ij} is an atomic condition. The quantifiers can be distributed over the B_i's, and we obtain the following equivalences:

$$\forall x_1, x_2, ..., x_n(C) \Leftrightarrow$$
$$\nexists x_1, x_2, ..., x_n(\neg C) \Leftrightarrow$$
$$\nexists x_1, x_2, ..., x_n(B_1 + B_2 + \cdots + B_m) \Leftrightarrow$$
$$(\nexists x_1, x_2 \cdots x_n(B_1))(\nexists x_1, x_2, ..., x_n(B_2))$$
$$...(\nexists x_1, x_2, ..., x_n(B_n)) .$$

To prove the validity of C it is sufficient to prove, for each B_i separately, that B_i is inconsistent (no value exists such that B_i evaluates to true). If, for any one of the B_i's, we can find a value $x'_1, x'_2, ..., x'_2$ such that B_i is true, then it immediately follows that C is not valid. Finding a value that satisfies all conditions in B_i, or proving that no such value exists, can be done by the algorithm explained below.

Let B be a conjunctive condition with variables $x_1, x_2, ..., x_n$. The first step of the algorithm is to define for each variable a permissible range which is consistent with its domain and the simple conditions of B. Consider a variable x_i and let its domain be $\{L_{x_i}, L_{x_i}+1, ..., H_{x_i}\}$. Denote its permissible range by $r_{x_i} = (a_{x_i}, b_{x_i})$.

For each variable x_i of B we initialize its range to $r_{x_i} = (L_{x_i}, H_{x_i})$. Then we adjust these initial ranges by considering each simple condition of B in turn. Let w denote a simple condition involving x_i. The permissible range of x_i is then adjusted as follows:

(i) if $w = (x_i > c)$ then $r_{x_i} := (\max(a_{x_i}, c+1), b_{x_i})$
(ii) if $w = (x_i < c)$ then $r_{x_i} := (a_{x_i}, \min(b_{x_i}, c-1))$
(iii) if $w = (x_i = c)$ then $r_{x_i} = (c, c)$.

At the end of this step we have for each x_i a permissible range which is consistent with its domain and the simple conditions. If the range of any one variable x_i is empty then the condition B is (trivially) inconsistent. A range $r_{x_i} = (a_{x_i}, b_{x_i})$ is empty if $a_{x_i} > b_{x_i}$.

Example: Consider the conjunctive condition

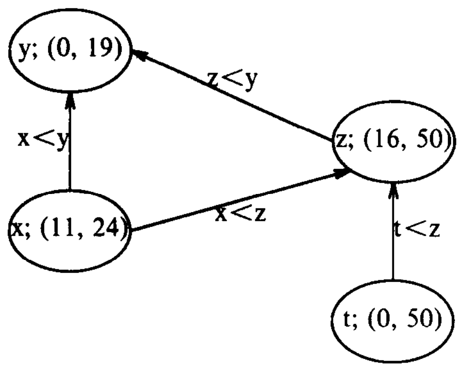
$$B' = (x>10)(y<20)(z>15)(x<25)(x<y)$$
$$(z<y)(x<z)(t<z)$$

where x,y,z are restricted to the domain $\{0,1,...,50\}$. After the first step we have the permissible ranges $r_x = (11,24)$, $r_y = (0,19)$, $r_z = (16,50)$ and $r_t = (0,50)$, so B' is not trivially inconsistent. □

If the condition B is not trivially inconsistent we proceed to the next step, in which the connective conditions are taken into account. A directed graph is constructed where nodes represent variables and edges represent connective conditions of the form $x > y$. A node may represent several variables, as explained below. Each node also has an associated permissible range, which initially is the range obtained from the previous step.

We will denote a node by $N(v; (a,b))$ where v is the set of variables associated with the node and (a,b) is their permissible range. For each variable x in B a node $N(x; (a_x, b_x))$ is created, where (a_x, b_x) is the permissible range obtained from the previous step. Next we merge nodes by using connective conditions of the form $x = y$. Whenever there is a condition $x = y$, we locate the node in whose variable set x occurs, and correspondingly for y. Denote these two nodes by $N_x(v; (a,b))$ and $N_y(u; (c,d))$, respectively. Node N_x is modified to $N_x(v \cup u; (\max(a,c), \min(b,d)))$ and N_y is eliminated. This step terminates when all connective conditions with equality have been processed. If any node now has an empty range, condition B is inconsistent and the processing of B stops. Otherwise, there is still at least one value that satisfies all simple conditions and all connective conditions with equality: the value obtained by setting each variable equal to the lower bound of the permissible range of the node in which the variable occurs.

In the final step we take into account connective conditions of the form $x > y$. For each condition $x > y$ we add an edge from the node in whose variable set y occurs to the node in whose variable set x occurs. For our example we obtain the following graph.



For a variable to have a value that satisfies all connective conditions, the value must be consistent with the values of all its predecessors in the graph. From the graph above we see that the lowest possible values are $t = 0$, $x = 11$, $z = \max(16, 11+1, 0+1) = 16$ and $y = \max(0, 11+1, 16+1, 11+2, 0+2) = 17$. This value satisfies all the conditions of B' and consequently we have found one value for which B' evaluates to true. However, if there had been an additional condition $t > 20$, then the minimum value for y would be $y = 23 = \max(0, 16+1, 11+2, 21+2)$. This is not a permissible value for y because the upper bound is 19, and consequently B' would be inconsistent.

Assume first that the graph obtained does not contain any cycles. Finding the minimum permissible value for each variable can then be done by processing the nodes in a certain order. Consider a node and its immediate predecessors in the graph. If the minimum permissible value for each immediate predecessor has been determined, then the minimum permissible value for (all the variables of) the node is 1 plus the highest of the minimum permissible values of all its immediate predecessors. Any lower value will violate at least one of the conditions. Whenever there are no cycles in the graph, we can easily find a processing sequence such that no node is processed before all its predecessors have been processed. First find all nodes that have no immediate predecessors and mark them. Then repeatedly select any node having only marked immediate predecessors, adjust the lower bound of its range and mark it. If at any stage during this process the range of a node t becomes empty, condition B is inconsistent (because there is no value for the variables of t that satisfies all the conditions). If, when all nodes have been processed, all permissible ranges are nonempty, then there exists at least one combination of values that satisfies all the conditions: the one obtained by setting each variable equal to the minimum value in its permissible range.

Now consider the situation when there are one or more cycles in the graph. Then there is at least one node t which is its own predecessor. Because all the predecessors of a node must be processed before a node can be processed, node t will never be processed. This will in the algorithm lead to a situation where there are still unmarked nodes in the graph, but they all have at least one unmarked (immediate) predecessor. This situation is easily detected. A cycle in the graph can arise only when there is a subset of connective conditions in B such that $(x_{i_1} > x_{i_2})(x_{i_2} > x_{i_3}) \cdots (x_{i_k} > x_{i_1})$. This set of conditions can never be satisfied and consequently B is inconsistent.

The above process for testing the inconsistency of a conjunctive boolean expression is summarized in the algorithm INCONSISTENT below. It is possible to strictly prove its correctness but we will not include the proof here. However, we think that its correctness is fairly obvious from the way it was constructed.

The validity of a universally quantified boolean expression can then be proved, or disproved, by the simple algorithm VALID-C which follows.

```
procedure VALID-$C$($C$ : boolean expression):
          boolean

 {$C$ is a boolean expression as explained
 in the beginning of this section}

  convert $\neg C$ into disjunctive form,
  that is, $\neg C = B_1 + B_2 + \cdots + B_n$

  for each $B_i$ do
    if not INCONSISTENT ($B_i$)
    then return (false)
  od;
  return (true);
end {VALID-$C$}
```

```
procedure INCONSISTENT ($B$: boolean
          expression): boolean

{$B$ must be of the form $B = C_1 C_2 \cdots C_m$ where
each $C_i$ is an atomic condition. Each variable $x$ in $B$
must have a finite range $(L_x, H_x)$.}

G: directed graph with nodes having the format
   $N(v;(a,b))$ where $v$ is a set of variable names and
   $(a,b)$ is the permissible range of the variables;

begin

 for each variable $x$ in $B$ do
    insert a new node $N_x(\{x\};(L_x,H_x))$ into G;
 od;

 for each simple condition $(x\ op\ c)$ in $B$ do
    find node $N_x(\{x\};(a_x,b_x))$;
    case $op$ of
      $<$: $(a_x,b_x) := (a_x, \min(b_x, c-1))$;
      $=$ : $(a_x,b_x) := (c,c)$:
      $>$ : $(a_x,b_x) := (\max(a_x, c+1), b_x)$;
    end;
    if $a_x > b_x$ then return (true);
 od;

 for each connective condition $(x = y)$ in $B$ do
    find nodes $N_x(v;(a,b))$ and $N_y(u;(c,d))$
    such that $x \in v$ and $y \in u$;
    $v := v \cup u$;
    $(a,b) := (\max(a,c), \min(b,d))$;
    delete $N_y$ from G ;
    if $a > b$ then return (true);
 od;

 for each connective condition $(x > y)$ in $B$ do
    find nodes $N_x(v;(a,b))$ and $N_y(u;(c,d))$
    such that $x \in v$ and $y \in u$;
    insert into G an edge from $N_y$ to $N_x$;
 od;

 while G is not empty do
    find any node $N_p(v;(a_p,b_p))$
    without incoming edges;
    if none exists then return (true) ;
    {detected a cycle}
    for each edge $s$ starting from $N_p$ do
     let $N_s(u,(a_s,b_s))$ be the end node of $s$;
     $(a_s,b_s) := (\max(a_s, a_p+1), b_s)$
     if $a_s > b_s$ then return (true);
    od
    delete node $N_p$ from G;
 od;
 return (false);
end {INCONSISTENT};
```

Testing the validity of a boolean expression is equivalent to the satisfiability problem for boolean expressions. It is well-known that testing satisfiability may, in the worst case, take exponential time. However, we do not consider this to be a very serious problem. Selection expressions are normally short, and the exponential explosion occurs only for expressions involving several "not equals" [RH80].

5. TESTING COVERAGE

We are now ready to consider the question whether a *PSJ*-expression E_0 is covered by another *PSJ*-expression E_1. Coverage means that every tuple obtained by evaluating E_0 must occur (as subtuples) in the relation obtained by evaluating E_1, and that this must hold for every instance of the database. Before stating the main theorem we need the notion of an extended attribute set.

Definition. Let C be a boolean expression with variables $x_1,x_2,...,x_n,y_1,y_2,...,y_m$. The variable y_i, $1 \le i \le m$, is said to be **uniquely determined** by $x_1,x_2,...,x_n$, with respect to C, if the following holds:

$$\forall x_1,...,x_n,y_1,...,y_m,y_1',...,y_m'(C(x_1,...,x_n,y_1,...,y_m)$$
$$C(x_1,...,x_n,y_1',...,y_m') \Rightarrow y_i = y_i') . \quad \square$$

Let C' stand for $C(x_1,...,x_n,y_1',...,y_m')$.

The above condition is equivalent to $\neg((y_i > y'_i + y_i < y'_i)CC')$ which can be tested using the algorithm VALID-*C*. The attribute set of a derived relation can be extended to include all variables uniquely determined by those already in the attribute set.

Definition. Let $E = (\mathbf{A},\mathbf{R},C)$ be a *PSJ*-expression and let $\mathbf{B}$ be the set of all attributes uniquely determined by the attributes of $\mathbf{A}$. Then $\mathbf{A} \cup \mathbf{B}$, denoted by $\mathbf{A}+$, is called the **extended attribute set** of E. $\square$

The set $\mathbf{B}$ in the definition above must obviously be a subset of $\alpha(C)$. An attribute not mentioned in C cannot be uniquely determined by the attributes of $\mathbf{A}$. The importance of the notion of an extended attribute set stems from the fact that given a tuple from E we can correctly derive the missing values of all the variables in the set $\mathbf{B}$. The procedure is explained below.

Let $\mathbf{A} = \{x_1,x_2,...,x_n\}$ and $\alpha(C) - \mathbf{A} = \{y_1,y_2,...,y_m\}$. Assume that $y_1,y_2,...,y_k$, $k \le m$, have been shown to be uniquely determined by $x_1,x_2,...,x_n$. We are given a tuple $t = (x_1^0,x_2^0,...,x_n^0)$ from the relation defined by E. The values of the variables $y_1,...,y_m$ are not known, but they must have been such that the tuple t, before the projection, satisfied the condition C. In other words, the following must hold: $\exists y_1,y_2,...,y_m(C(x_1^0,...,x_n^0,y_1,y_2,...,y_m)$.

This is equivalent to $\neg(\forall y_1,y_2,...,y_m(\neg C))$. The algorithm VALID-*C* is now applied to $\forall y_1,...,y_m(\neg C)$, modifying it to return the values found. Denote the values returned by $y_1^0,y_2^0,...,y_m^0$. Such a value combination always exists, because otherwise t would not have satisfied C in the first place. The values for the variables $y_1,y_2,...,y_k$ are guaranteed to be unique, because each variable $y_1,y_2,...,y_k$ is uniquely determined by $x_1,x_2,...,x_n$. In other words, we have correctly reconstructed the missing values of $y_1,y_2,...,y_k$. The following examples illustrates the discussion.

Example: Let $E = (\{A\},R,(B=A)((A>7)(C=5)+(A<5)(C=10))$ where $R = (A,B,C,D)$. One can easily show that B and C (but not D) are uniquely determined by A. Now consider the following instance of R and the result when E is applied to that instance:

R:	A	B	C	D	E:	A
	10	10	5	4		10
	10	11	5	6		4
	11	11	7	4		
	4	4	10	6		
	3	3	14	6		

Each tuple appearing in the result of E must have satisfied the selection condition. For the first tuple in E we can conclude that $B = 10$ (because of the condition $B = A$) and that $C = 5$ (because $A > 7$). In the same way, we find for the second tuple that $B = 4$ and $C = 10$. $\square$

Lemma. *Let $E = (\mathbf{A},\mathbf{R},C)$ be a PSJ-expression. Given a tuple from E, the value of an attribute y, is guaranteed to be uniquely reconstructible if and only if $y \in \mathbf{A}+$.* $\square$

Sketch of proof: The procedure above for reconstructing missing values shows that the condition $y \in \mathbf{A}+$ is sufficient. To show that it is also necessary, assume that $y \notin \mathbf{A}+$. Then we can find two tuples t_0 and t_1 over the base of E (the Cartesian product of all relations in $\mathbf{R}$) such that they agree on all attributes in $\mathbf{A}$ but differ at least on the attribute y. This follows directly from the definition of "uniquely determined" above. The database instance obtained by projecting t_0 and t_1 onto the attributes of each relation in $\mathbf{R}$ is a valid instance. When t_0 and t_1 are projected onto the attributes of $\mathbf{A}$ they will map to the same tuple, because they agree on all attributes of $\mathbf{A}$. Given only this tuple from E (and the condition C) the value of y cannot be uniquely determined. There are at least two different values of y, which when combined with the given tuple satisfy C. $\square$

Given a tuple in E, we know the values of all attributes in $\mathbf{A}$ and we can correctly determine the values of all other attributes in $\mathbf{A}+$. The above lemma shows that $\mathbf{A}+$ is the maximal set of attributes whose values can be determined from a tuple in E. The following theorem gives necessary and sufficient conditions on coverage. Note the requirement that all instances of the relations involved must be non-empty.

Theorem 1. *Consider two PSJ-expressions* $E_0 = (\mathbf{A}_0, \mathbf{R}_0, C_0)$ *and* $E_1 = (\mathbf{A}_1, \mathbf{R}_1, C_1)$ *defined over a database where every relation in* $\mathbf{R}_0 \cup \mathbf{R}_1$ *is non-empty. Let* $\alpha(C_0) \cup \alpha(C_1) = \{x_1, x_2, \ldots, x_n\}$. *Then* E_0 *is covered by* E_1 *if and only if* $\mathbf{A}_0 \subseteq \mathbf{A}_1+$ *and* $\forall x_1, x_2, \ldots, x_n (C_0(x_1, x_2, \ldots, x_n) \Rightarrow C_1(x_1, x_2, \ldots, x_n))$. □

Sketch of Proof: First extend the relation set of E_0 and E_1 to $\mathbf{R}_0 \cup \mathbf{R}_1$ so that they are defined over a common base (Cartesian product). Adding a new relation S to the relation set of an expression does not change the result of the expression (as long as the new relation always contains at least one tuple). A tuple t in the base before adding S will give rise to a number of "copies" in the extended base, where the copies differ only in the values of the attributes of S. If t satisfied a given selection condition, then all its "copies" will satisfy the condition as well. If t did not satisfy the condition, then none of the "copies" will satisfy it. The added attributes have no effect on the selection. The final projection will reduce all "copies" of t to a single tuple, exactly the same one as the projection of t.

Assume that $\mathbf{A}_0 \subseteq \mathbf{A}_1+$ and that the implication holds. The fact that the implication holds, guarantees that each tuple t in the common base satisfying C_0, will also satisfy C_1. Because $\mathbf{A}_0 \subseteq \mathbf{A}_1+$, the projection of t onto $\mathbf{A}_0$ is a subtuple of the projection of t onto $\mathbf{A}_1+$. Consequently E_0 is covered by E_1.

It is obvious that the condition $\mathbf{A}_0 \subseteq \mathbf{A}_1+$ is necessary for coverage. Now assume that E_0 is covered by E_1 and that $\mathbf{A}_0 \subseteq \mathbf{A}_1+$, but that the implication does not hold. Then there must exist a tuple t such that it satisfies C_0 but not C_1. Now consider the database instance where each relation $S_i \in \mathbf{R}_0 \cup \mathbf{R}_1$ consist of the single tuple obtained by projecting t onto the attributes of S_i. When E_0 is evaluated over this instance, the result will be one tuple: the projection of t onto $\mathbf{A}_0$. However, when evaluating E_1 over the same database instance the result will be the empty set. Hence E_0 is not covered by E_1 for this instance, contradicting the assumption that E_0 is covered by E_1 for every instance of the database. □

By the above theorem testing coverage is reduced to two simpler tests. Note that it is not always necessary to construct $\mathbf{A}_1+$. If $\mathbf{A}_0 \subseteq \mathbf{A}_1$ the first condition is trivially satisfied. If this is not the case, then we must test whether every variable in $\mathbf{A}_0 - \mathbf{A}_1$ is uniquely determined by the attributes in $\mathbf{A}_1$. Let x be a variable in $\mathbf{A}_0 - \mathbf{A}_1$. A first simple test is to check whether x is in $\alpha(C_1)$. If x is not mentioned in C_1, then it cannot be uniquely determined by those in $\mathbf{A}_1$.

Note that Theorem 1 can be used to test equivalence of relational expressions for a wider class of expressions than those considered in [AS79] and [SY81].

6. TESTING DERIVABILITY

The fact that an expression E_0 is covered by a derived relation E_1 does not automatically guarantee derivability. All the necessary tuples are guaranteed to exist in E_1, but it may be impossible to select out the ones belonging to E_0, as illustrated by the following example.

Example: Consider a database consisting of the single relation $S(A,B,C)$. Let $E_0 = (\{A\},\{S\},(B>10)(C<20))$ and $E_1 = (\{A,C\},\{S\},B>5)$. It is easy to see that E_0 is covered by E_1 because every tuple satisfying $(B>10)(C<20)$ will automatically satisfy $B > 5$. However, to compute E_0 from the derived relation E_1, attribute B is needed (to further qualify the tuples) and B is not present in E_1 after the projection. Hence E_0 cannot be derived from E_1. □

Consider $E_0 = (\mathbf{A}_0, \mathbf{R}_0, C_0)$ and $E_1 = (\mathbf{A}_1, \mathbf{R}_1, C_1)$ and assume that E_0 is covered by E_1. If $\alpha(C_0) \subseteq \mathbf{A}_1+$, that is, if all the attributes needed to evaluate the condition C_0 are present in $\mathbf{A}_1+$, it is obvious that E_0 can be derived from E_1. However, even when some attributes in $\alpha(C_0)$ are missing from $\mathbf{A}_1+$ it may still be possible to derive E_0 from E_1. The missing attributes may not be needed. In the example above that would be the case if the condition in E_0 were $B > 5$ instead of $B > 10$. To handle this type of situation we introduce the notion of essential and nonessential variables.

Definition. Consider a boolean expression C with variables $x_1, x_2, \ldots, x_n$. The variable x_k is said to be **nonessential** if the following holds

$$\forall x_1, \ldots, x_k, \ldots, x_n, x_k' (C(x_1, \ldots, x_k, \ldots, k_n) \\ = C(x_1, \ldots, x_k', \ldots, x_n)) .$$

Otherwise x_k is an **essential** variable of C. □

A nonessential variable can be eliminated from the condition simply by replacing it with any value from its domain. This will in no way change the value of the expression. However, for derivability we need to know whether a variable in a condition is essential given that another condition holds. This is covered by the next definition.

Definition. Let C_0 and C_1 be boolean expressions with variables $x_1, x_2, \ldots, x_n$. The variable x_k is said to be **computationally nonessential** in C_0 with respect to C_1 if the following condition holds:

$$\forall x_1, \ldots, x_k, \ldots, x_n, x_k' \\ (C_1(x_1, \ldots, x_k, \ldots, x_n) C_1(x_1, \ldots, x_k', \ldots, x_n) \\ \Rightarrow C_0(x_1, \ldots, x_k, \ldots, x_n) = C_0(x_1, \ldots, x_k', \ldots, x_n)) . \ \square$$

The basic idea of this definition is that the exact value of x_k does not matter when evaluating C_0. It can be replaced by any value x_k' as long as the value x_k' is such that C_1 is satisfied. The algorithm VALID-C in section 3 can be used to test whether the condition holds. Let C_1' stand for $C_1(x_1, \ldots, x_k', \ldots, x_n)$ and similarly for C_0'. Then the condition is equivalent to

$$\forall x_1, \ldots, x_k, \ldots, x_n, x_k' \\ (\neg(C_1 \ C_1'(C_0 + C_0')(\neg C_0 + \neg C_0'))) ,$$

to which the algorithm can be applied directly.

Both definitions above can trivially be extended to the case of several nonessential variables by interpreting x_k (and x_k') as sets of variables. Note also that a variable not occurring in C_0 is (trivially) nonessential.

If a variable (or set of variables) y of C_0 is computationally nonessential with respect to C_1, we can correctly evaluate C_0 without knowing the exact value of y. However, y cannot be assigned an arbitrary value in its domain. Consider two *PSJ*-expressions $E_0 = (\mathbf{A}_0, \mathbf{R}_0, C_0)$ and $E_1 = (\mathbf{A}_1, \mathbf{R}_1, C_1)$ where E_0 is covered by E_1. Let $\alpha(C_0) = \{x_1, x_2, \ldots, x_n, y\}$ where $\{x_1, \ldots, x_n\} \subseteq \mathbf{A}_1 +$ and $y \notin \mathbf{A}_1 +$. Assume that we have shown that y is computationally nonessential with respect to C_1. Then y must be one of the variables in C_1. Now consider an arbitrary but fixed tuple $t_1 = (x_1', x_2', \ldots, x_n', y', \ldots)$ from E_1 before the projection onto $\mathbf{A}_1$. After the projection y disappears and its exact value in t_1 is not known. However, we know that it was such that t_1 satisfied C_1. If we somehow can find a surrogate value y'' such that the tuple $t_2 = (x_1', x_2', \ldots, x_n', y'', \ldots)$ satisfies C_1, then from the fact that y is computationally nonessential it follows that $C_0(x_1', x_2', \ldots, x_n', y') = C_0(x_1', x_2', \ldots, x_n', y'')$. In other words, using the surrogate value y'' when evaluating C_0 gives exactly the same result as would have been obtained using the correct, but unknown, value y'.

Example: Let $R(A,B,C,D)$, $E_0 = (\{A,B\}, \{R\}, (A<50)(B>10)(C<40)(D<C)(D>B))$ and $E_1 = (\{A,B,C\}, \{R\}, (B>10)(C<40)(D<C)(D>B) + (B<5))$. E_1 covers E_0 and it is easy to show that D is computationally nonessential in C_0. Now consider the following instance of the relation defined by E_1:

E_1:	A	B	C
	100	30	35
	20	15	25
	20	4	0

For the first tuple we can deduce that the missing value of D must have been in the range $30 < D < 35$. We can set D to any value in this range, $D = 31$, for example. Using $D = 31$ and the given values for the other attributes, we find that the first tuple does not satisfy the condition in E_0. For the second tuple the range is $15 < D < 25$. Setting $D = 16$ we find that the second tuple satisfies the selection condition of E_0. For the third tuple, D can be assigned any value in its range, $D = 0$, for example. (The tuple satisfies the condition $B < 5$ independently of the value of D.) With $D = 0$ (or any other value) the tuple does not satisfy the condition in E_0. The final result of E_0 is hence the single tuple (20, 15). $\square$

Surrogate values for computationally nonessential variables can easily be found using the algorithm given in section 3. Let $\mathbf{A}_1 = \{x_1 x_2, \ldots, x_n\}$ and let $s = (x_1', \ldots, x_n')$ be a tuple in the relation defined by E_1. This tuple must then have satisfied C_1. Let $\alpha(C_1) - \mathbf{A}_1 = \{y_1, y_2, \ldots, y_k\}$, that is, the variables of C_1 not retained after the projection. All computationally nonessential variables of C_0 are guaranteed to be in the set $\{y_1, \ldots, y_k\}$. Finding surrogate values for $y_1, y_2, \ldots, y_k$ to be used with the given tuple $s = (x_1', x_2', \ldots, x_n')$ amounts to proving $\exists y_1, y_2, \ldots, y_k (C_1(x_1', x_2', \ldots, x_n', y_1, y_2, \ldots, y_k))$. Proving this is equivalent to disproving $\forall y_1, y_2, \ldots, y_k (\neg\, C_1(x_1', \ldots, x_n', y_1, \ldots, y_k))$ which can be done by applying the algorithm of section 3 (modifying it to return the value found). Surrogate values $y_1', y_2', \ldots, y_k'$ will always exist, because otherwise s could not have satisfied C_1. It may be necessary to compute new surrogate values for every tuple, as we saw in the example above. However, when given a new tuple, it may be worthwhile trying to use previously computed surrogate values. If the attribute values of the new tuple combined with the 'old' surrogate values satisfy C_1, then there is no need to recompute the surrogates.

The following theorem summarizes the main result of this paper:

Theorem 2. *Let* $E_0 = (\mathbf{A}_0, \mathbf{R}_0, C_0)$ *and* $E_1 = (\mathbf{A}_1, \mathbf{R}_1, C_1)$ *be PSJ-expressions and assume that* E_0 *is covered by* E_1. *Then* E_0 *is derivable from* E_1 *if and only if all variables in* $\alpha(C_0) - \mathbf{A}_1+$ *are computationally nonessential with respect to* C_1. □

Sketch of Proof: If the condition holds then E_0 can be computed in the way discussed above, so the condition is clearly sufficient. To show that it is necessary, assume that $\alpha(C_0) - \mathbf{A}_1+$ is nonempty, that $y \in \alpha(C_0) - \mathbf{A}_1+$ is a computationally essential variable, and that E_0 is derivable from E_1. Then there must exist a relational algebra expression F, not involving the missing attribute y, which correctly computes E_0 from E_1 for every instance of the database. Denote by S the base of E_1, that is, the Cartesian product of all relations in $\mathbf{R}_1$. Because y is computationally essential, we can construct two tuples t_1 and t_2 over S which differ only in the value of y and which both satisfy C_1 but only one, t_1 say, satisfies C_0. Now consider the following two database instances: d_1 obtained by projecting t_1 onto the relations in $\mathbf{R}_1$ and d_2 by similarly projecting t_2. E_0 evaluated over t_1 must then result in one tuple, while E_0 evaluated over t_2 must be empty. However, evaluating F over t_1 gives exactly the same result as evaluating F over t_2, because they differ only in the value of y and the expression F does not involve the attribute y. Hence, F gives an incorrect result for one of the two database instances, contradicting the assumption that E_0 is derivable from E_1. □

Note that the proof does not impose any other restrictions on the deriving expression F than that it does not involve attribute y. This implies that the theorem holds also for expressions that involve "self-joins". In other words, if E_0 cannot be derived from E_1 then it cannot be derived from $E_1 \times E_1$ or $E_1 \times E_1 \times E_1, \ldots$ either.

7. DISCUSSION

In this section we summarize the main assumptions made and briefly discuss the effect of relaxing them.

One of the main restrictions was not to allow "self-joins". A "self-join" in a *PSJ*-expression is indicated by the same relation name occurring more than once in the relation set when the expression is converted into triple format. This restriction can be slightly relaxed. Assume that a relation R occurs k, $k > 1$, times in the relation set of E_0 and that the k occurrences have been renamed $R^1, R^2, \ldots, R^k$. If the same relation occurs in E_1 with at least the same multiplicity, E_0 may be derivable from E_1. Assume that there are l, $l \geq k$, occurrences of R in E_1 and that they have been renamed $S^1, S^2, \ldots, S^l$. If we can find a mapping of the names $R^1, R^2, \ldots, R^k$ onto a subset of $S^1, S^2, \ldots, S^l$ such that E_0 is derivable from E_1 under the mapping, the problem is solved. (A mapping associates each name $R^1, R^2, \ldots, R^k$ with some name $S^1, S^2, \ldots, S^l$.) There does not seem to be any better way of solving this than trying all $l(l-1)\ldots(l-k+1)$ possible mappings.

The algorithm in section 4 works only for a restricted class of boolean expressions: only those where atomic condition are restricted to a comparison of two variables or a comparison of a variable with a constant. Note, however, that none of the results in sections 5 and 6 depend directly on this algorithm. Any class of boolean expressions can be handled provided that we have an algorithm for testing the validity (or satisfiability) of expressions of that class. The core of such an algorithm is an algorithm for testing whether a set of inequalities and/or equalities can all be simultaneously satisfied. Whether such an algorithm exists or not, and its complexity, depend completely on the type of expressions (functions) allowed in the (in)equalities and the domain of the variables. If linear expressions with variables ranging over the real numbers are allowed, the problem is equivalent to findingng a feasible solution to a linear programming problem.

The reader may have noticed that so far we have not mentioned keys, functional dependencies, inclusion dependencies and so on. What effect will they have on derivability? The only constraints taken into account in this paper are domain constraints: any combination of attribute values drawn from their respective domains represents a tuple that potentially may occur in an instance of the relation in question. The only effect of functional dependencies, multivalued dependencies, inclusion dependencies, etc. is to impose constraints on what tuples can occur simultaneously in the database, thereby reducing the set of permitted database instances. Clearly the conditions given for derivability are still sufficient, but they may not be necessary. The proof that the stated conditions are necessary for derivability relies on a certain database instance, consisting of two tuples, to produce a contradiction. In the presence of additional constraints, this may not be a permissible database instance.

References

AL80 Adiba, M.E. and Lindsay, B.G.,
Database Snapshots.
Proc. of 6th Intl. Symp. on VLDB, ACM, New

York, N.Y., (1980), 86-91.

AS79 Aho, A.V., Sagiv, Y. and Ullman, J.D., Equivalence of Relational Expressions. SIAM J. of Computing, 8, 2 (1979), 218-246.

AS76 Astrahan, M.M. et al, System R: A Relational Approach to Database Management. ACM TODS, 1,2, (1976), 97-137.

BAB82 Babb, E., Joined Normal Form: A storage encoding for relational databases. ACM TODS 7, 4 (1982), 588-614.

BL81 Blasgen, M.V. et al., Systems R: An architectural overview. IBM Systems J., 20, 1, (1981), 41-62.

CP84 Ceri, S. and Pelagatti, G., Distributed Databases - Principles & Systems. McGraw-Hill, New York, N.Y., 1984.

FS82 Finkelstein, S., Common Expression Analysis in Database Applications. Proc. 1982 ACM SIGMOD Intl. Conf. on Management of Data. ACM, New York, N.Y., (1982), 235-245.

MA83 Maier, D., The Theory of Relational Database. Computer Science Press, Rockville, Maryland, (1983).

RH80 Rosenkrantz,D.J. and Hunt, H.B., Processing Conjunctive Predicates and Queries. Proc. 6th Intl. Symp. on VLDB, ACM, New York, N.Y., (1980), 64-72.

RO82 Roussopoulos, N., View Indexing in Relational Database. ACM TODS, 7,2, (1982), 258-290.

SY81 Sagiv, Y. and Yannakakis, M., Equivalence among Relational Expression with Union and Difference Operators. JACM, 27, 4 (1981), 633-655.

SS81 Schkolnick, M. and Sorensen, P., The Effects of Denormalization on Database Performance. IBM Research Rep. RJ3082, (1981).

SW76 Stonebraker, M., Wong, E., Kreps, P. and Held, G., The Design and Implementation of INGRES. ACM TODS, 1,3 (1976), 189-222.

UL82 Ullman, J.D., Principles of Database Systems. Computer Science Press, Rockville, Maryland, (1982).