# Query Transformation for PSJ-queries

*H.Z. Yang and P.-Å. Larson*

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

**Abstract**

Consider a database containing not only base relations but also derived relations (also called materialized or concrete views). Relation fragments in a distributed database, view indexes, and intermediate results obtained during query procsssing are all examples of derived relations. The query transformation problem is then the following: Given a query (or a subquery), can it be computed from the available set of derived relations, and, if so, how? We have solved the query transformation problem for the case when both the query and the derived relations are defined by *PSJ*-expressions, that is, relational algebra expressions containing only projects, selects and joins. This paper gives an overview of the underlying theory, shows how to reduce the number of attribute mappings to be considered, and presents a prototype system for query transformation.

## 1. Introduction

A derived relation is a relation resulting from the evaluation of a query over some database instance. Assume that we have available in stored form a set $E_1, E_2, \cdots, E_n$ of derived relations, where each derived relation is defined by a relational algebra expression over the conceptual relations $R_1, R_2, \cdots, R_m$. We are given a query $Q$, that is, a relational algebra expression over $R_1, R_2, \cdots, R_m$. The general *query transformation problem* is then the following: Can $Q$ be computed from the data available in the derived relations $E_1, E_2, \cdots, E_n$ and, if so, how? There are two different versions of this problem. Here we consider only the version which requires that the query be computable from the derived relations for *every* possible instance of the database (the *intensional* version). A more limited version of the problem is to restrict the question to the *current* instance of the database (the *extensional* version).

This work was supported by Cognos Inc. under contract WRI 502-12 and by the Natural Sciences and Engineering Research Council of Canada under grant No. A2460.
Electronic mail: {hzyang, palarson}@waterloo.csnet



We are investigating the query transformation problem under the assumption that both queries and derived relations are defined by *PSJ*-expressions. A *PSJ*-expression is a relational algebra expression constructed from an arbitrary number of projects, selects, and joins. This work has progressed to the point where the theoretical issues of the query transformation problem for *PSJ*-expressions are essentially solved. Note that any query can be transformed by isolating the subqueries consisting of *PSJ*-expressions and then transforming each subquery separately. One of the key concepts in query transformation is that of an *attribute mapping*. An attribute mapping uniquely defines a subquery over the derived relations $E_1, E_2, \cdots, E_n$. When this subquery is evaluated, it produces tuples containing the correct attributes and satisfying the conditions of the query $Q$. The problem of query transformation then boils down to selecting a set $\mathbf{M} = \{M_1, M_2, ..., M_s\}$ of attribute mappings, such that the union of the corresponding subqueries is equivalent to $Q$.

The first part of this paper gives a brief overview of the theory of query transformation. The underlying theory is presented in detail in [LY87]. Some early results were reported in [LY85]. A main part of query transformation is the generation of a sufficient set of attribute mappings. However, there may be many such sets and the number of possible attribute mappings may be large. In the second part of the paper we show that a only limited set of mappings need be considered. In the last part of the paper we briefly present a prototype system for query transformation and show the transformation of a few example queries.

The query transformation problem arises, in various forms, in several different areas of query processing for relational databases. In the context of distributed databases, derived relations can be interpreted as relation fragments stored at various sites. This variant of the problem has been studied extensively, normally under the assumption that each fragment is derived from a single relation using only selections and projections [CP84]. The problem also arises in traditional query optimization [MA83]. In this context, a derived relation can be interpreted as an intermediate result obtained in the process of computing a query. If some other part of the query can easily be computed from available intermediate results, the cost of processing the query may be reduced. The problem of recognizing common subexpressions and the equivalence problem for relational expressions [AS79, SY81] are special cases of the query transformation problem where the query $Q$ is required to exactly match one of the derived relations $E_1, E_2, \cdots, E_n$. It may be worthwhile saving intermediate results and using them for processing other queries. View indexing is a variant of this approach which has been implemented in the ADMS system [RO82, RO86]. In multiple query optimization, the potential for cost reductions by sharing of intermediate results is even greater [FS82, SE86].

Our main motivation for studying this problem stems from a different area: the problem of structuring the stored database in relational systems. In current systems, there is a one-to-one correspondence between conceptual relations and stored relations, that is, each conceptual relation exists as a separate stored relation. We are investigating a more flexible approach where the stored database consists of a set of derived relations and conceptual relations do not necessarily exist in stored form. The choice of stored relations should be guided by the query load so that frequently occuring queries can be answered rapidly. The structure of the stored database should be completely transparent at the user level, and user queries and updates should be expressed solely in terms of conceptual relations. The system must then be able to automatically transform user queries and updates into equivalent queries and updates against stored relations. To make this approach viable the two fundamental problems of query transformation and update transformation must be solved. Some early work based on similar ideas but restricted to prejoining relations is reported in [SS81, BA82]. The update transformation problem is discussed in [BC86].

## 2. Notation and basic results

We consider only derived relations and queries defined by *PSJ*-expressions, that is, relational algebra expressions containing only the operations project, select and join. Cartesian product is seen as a special case of a join. Any valid *PSJ*-expression $E$ can be transformed into a standard form consisting of a Cartesian product, followed by a selection, followed by a projection:

$$E = \pi_{\mathbf{A}}\sigma_C(R_1 \times R_2 \times \cdots \times R_k)$$

We can therefore represent any *PSJ*-expression by a triple $E = (\mathbf{A}, \mathbf{R}, C)$ where $\mathbf{A} = \{A_1, A_2, \cdots, A_l\}$ is called the ***attribute set***, $\mathbf{R} = \{R_1, R_2, \cdots, R_k\}$ the ***relation set*** or ***base***, and $C$ the ***selection condition*** of the expression. It is easy to see that this is correct by considering the operator tree representation of 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.

The notation $\alpha(C)$ and $\alpha(R)$ is used to denote the set of attributes mentioned in a condition $C$ and the attributes of a relation $R$, respectively. The logical connectives are denoted by $\vee$ for OR, juxtaposition or $\wedge$ for AND, $\neg$ for NOT, and $\Rightarrow$ for logical implication. To indicate that all variables of a condition $C$ are universally quantified we write $\forall C$, and similarly for existential quantification. If we need to explicitly indicate which variables are quantified we write $\forall X(C)$ where $X$ is a set of variables. A ***partial evaluation*** of a condition $C$ is obtained by replacing some of its variables by values from the corresponding domains. Let $t$ be a tuple over some set of attributes. The partial evaluation of $C$ with respect to $t$ is denoted by $C[t]$. The result is a new condition with fewer variables. The result of evaluating a relational algebra expression $E$ over an instance $d = \{r_1, r_2, \cdots, r_m\}$ of a database $\mathbf{D} = \{R_1, R_2, \cdots, R_m\}$ is denoted by $V(E, d)$. The result is a set of tuples, that is, a relation instance.

The concept of the ***extended attribute set*** of a derived relation was introduced in [LY85]. Consider a derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$. Given a tuple from $E$ we may be able to correctly reconstruct the value of an attribute not in $\mathbf{A}$. The extended attribute set, denoted by $\mathbf{A}^+$, consists of $\mathbf{A}$ and all attributes whose values can be correctly reconstructed from the values of the attributes in $\mathbf{A}$ for any tuple that satisfies the condition $C$. The exact definition and a general reconstruction procedure are given in [LY85, LY87]. A simple example clarifies the basic idea. Consider the derived relation defined by $E = (\{A, C\}, \{R_1, R_2\}, (B=C)(D=5))$ over $R_1(A, B)$ and $R_2(C, D)$. The extended attribute set is $\mathbf{A}^+ = \{A, B, C, D\}$ because for any tuple satisfying $(B=C)(D=5)$, we know that the value of $B$ must be equal to the value of $C$ and the value of $D$ must be 5. We often refer to the attributes in the extended attribute set as ***visible attributes***.

Transforming a query involves testing whether or not certain Boolean expressions are valid or equivalently, whether their complements are unsatisfiable. Let $C(x_1, x_2, \cdots, x_n)$ be a Boolean expression over variables $x_1, x_2, \cdots, x_n$. $C$ is valid if it evaluates to ***true***, and unsatisfiable if it evaluates to ***false*** for all possible values of its variables. It is satisfiable if it evaluates to ***true*** for some value. Proving the validity of a Boolean expression is equivalent to disproving the satisfiability of its complement. Proving the satisfiability of a Boolean expression is, in general, NP-complete. However, for a restricted class of expressions polynomial algorithms exist. Rosenkrantz and Hunt [RH80] developed such an algorithm for conjunctive Boolean expression. The expression must be in the form $B = B_1 \wedge B_2 \wedge \cdots \wedge B_m$ where each $B_i$ is an atomic condition. An atomic condition must be of the form $x\ op\ y+c$ or $x\ op\ c$ where $op \in \{=, >, \geq, <, \leq\}$, $x$ and $y$ are integer variables, and $c$ is a constant. The running time of the algorithm is $O(n^3)$ where $n$ is the number of distinct variables. We [LY85, LY87] designed a similar algorithm with running time $O(n^2)$ for the case when all variables range over some finite (integer) interval. However, it does not handle atomic conditions of the type $x\ op\ y+c$ where $c \neq 0$. A modified version of the algorithm by Rosenkrantz and Hunt can be found in [BC86].

An expression not in conjunctive form can be tested by first converting it into disjunctive normal form and then testing each conjunct separately. In the worst case, this may cause the length of the expression to grow exponentially.

## 3. Theoretical background

To gain some understanding of what is involved in computing a query from derived relations, we take a look at an example.

*Example:* Consider the following query and derived relations defined over relations $R(\underline{A}, B, C)$ and $S(D, \underline{E}, F)$. We assume that $A$ is the key of $R$,and $E$ the key of $S$.

$$Q = (\{R.B, R.C, S.E\}, \{R, S\}, (R.A = S.D)(R.C < 20))$$

$$E_1 = (\{R.A, R.C, S.E\}, \{R, S\}, (R.A = S.D))$$

$$E_2 = (\{R.A, R.B, R.C\}, \{R\}, (R.B > 10)(R.C < 20))$$

$$E_3 = (\{R.A, R.B, S.F\}, \{R, S\}, (R.A = S.D)(R.B < 20))$$

The extended attribute sets of the derived relations are $\mathbf{A}_1^+ = \{R.A, R.C, S.D, S.E\}$, $\mathbf{A}_2^+ = \{R.A, R.B, R.C\}$, $\mathbf{A}_3^+ = \{R.A, R.B, S.D, S.F\}$, respectively. $E_1$ contains all the tuples required to answer the query, but attribute $R.B$ is missing. $R.B$ can be obtained from $E_2$ and $E_3$ by a "back-join". Both $E_2$ and $E_3$ are needed. Neither $E_2$ nor $E_3$ can alone contribute all the necessary $R$-tuples because of the conditions $(R.B>10)(R.C<20)$ and $(R.B<20)$. The following transformed query will give the desired result:

$$Q = F_1 \cup F_2 \text{ where}$$

$$F_1 = (\{E_2.R.B, E_1.R.C, E_1.S.E\}, \{E_1, E_2\}, (E_1.R.A = E_2.R.A))$$

$$F_2 = (\{E_3.R.B, E_1.R.C, E_1.S.E\}, \{E_1, E_3\}, (E_1.R.A = E_3.R.A)(E_1.R.C < 20))$$

The back-join is expressed in $F_1$ by the condition $E_1.R.A = E_2.R.A$. The joined tuples will automatically satisfy the selection condition of $Q$ and no further qualification is necessary. The back-join in $F_2$ is expressed by $E_1.R.A = E_3.R.A$. The joined tuples must be further qualified by $E_1.R.C < 20$ because the second part of the condition of $Q$ is not automatically satisfied. The two back-joins are "safe" (lossless) because the join is over the key of $R$. □

Given a query $Q$ and a set $\{E_1, E_2, ..., E_m\}$ of derived relations, we attempt to construct an equivalent query of the form $F_1 \cup F_2 \cup \cdots \cup F_s$, where each $F_i$ is a (generalized) *PSJ*-query expressed in terms of a subset of the derived relations $\{E_1, E_2, ..., E_m\}$. For every attribute mentioned in $Q$, there is only a limited number of "value sources" in $\{E_1, E_2, ..., E_m\}$. In the example above, the possible sources for $R.B$ were $\{E_2, E_3\}$ and for $R.A$ the possible sources were $\{E_1, E_2, E_3\}$. The concept of ***attribute mappings*** defined further below formalizes the idea of value sources.

Unless otherwise stated, we will in the sequel consider every attribute name to be prefixed with the name of the derived relation or query from which it is taken. A complete attribute name then consist of three parts: the name of the derived relation/query, the relation name, and the attribute name. This makes it possible to uniquely identify attributes mentioned in several derived relations. Let $attr(E_i) = \mathbf{A}_i \cup \alpha(C_i)$, denote the set of attributes (using complete attribute names) mentioned in $E_i$, and similarly for $attr(Q)$.

We have to introduce some additional notation at this point. Let $T$ be a set of complete (three-part) attribute names. Then $proj(R, T)$ will denote the set of attributes in $T$ that originate from (conceptual) relation $R$ (using three-part names). $exp(T)$ will denote the set of derived relations and/or queries from which the attributes are taken. $noexp(T)$ will denote the set of attribute names in $T$ but without the name of the derived relation or query from which they originate (that is, using two-part names).

*Example:*
For $T = \{Q.R.A, E_1.R.A, E_1.S.B, E_2.R.A\}$ we have

$$proj(R,T) = \{Q.R.A, E_1.R.A, E_2.R.A\}$$

$$exp(T) = \{Q, E_1, E_2\}$$

$$noexp(T) = \{R.A, S.B\}$$

**Definition:** An ***attribute mapping*** $M$ is a mapping (function) from $attr(Q)$ to $\bigcup_{1 \le i \le m} attr(E_i)$ with the following properties:

1. $M$ is one-to-one (injective).
2. For every attribute $Q.R_k.A_j \in attr(Q)$, $M(Q.R_k.A_j) = E_i.R_k.A_j$ for some $i$, $1 \le i \le m$, that is, an attribute mentioned in $Q$ can only be mapped to an attribute having the same relation name and attribute name.

The need for these requirements is obvious: each attribute mentioned in $Q$ must be associated with one, and only one, corresponding attribute in one of the derived relations. The set of derived relations in the image of $attr(Q)$ is given by $exp(M(attr(Q))) = \{E_{i_1}, E_{i_2}, ..., E_{i_k}\}$. We call this the *base* of the mapping $M$ and denote it by $\mathbf{B}_M$.

A mapping $M$ identifies a value source for each attribute in $Q$. Assume, for the moment, that all attributes in $\mathbf{A}_q$ are mapped to visible attributes. The expression $\pi_{M(\mathbf{A}_q)}(E_{i_1} \times E_{i_2} \times \cdots \times E_{i_k})$ would then generate tuples of the correct form, that is, containing all the required attributes. The problem is, of course, that not all tuples generated are valid "response tuples". The task is to define a function $F_M$ that extracts as many valid tuples as possible from the Cartesian product of the derived relations in the base $\mathbf{B}_M$. In order to accept a tuple $t$ from the Cartesian product into the response set, we must guarantee that it has the following three properties:

1. $t$ is not a spurious tuple
2. $t$ satisfies the query condition $C_q$
3. the values of all attributes in $\mathbf{A}_q$ are either visible in $t$ or can be reconstructed from the values visible.

Necessary and sufficient conditions for a tuple to satisfy these requirements are given (without proofs) in the next three sections. The conditions can all be tested at run time and thus define the required function $F_M$. A tuple is rejected by $F_M$ either because it is not a valid response tuple or because it is unsafe. Unsafe means that the tuple *may* be valid, but it cannot be guaranteed. $F_M$ thus extracts from the base the maximal sset of tuples that $M$ can safely contribute to the response set.

The function $F_M$ can be viewed as an extended *PSJ*-expression, which can be written in the following form

$$F_M = \pi_{M(\mathbf{A}_q)}\sigma_{C_M^W}\sigma_{C_M^B}(E_{i_1}\times E_{i_2}\times\cdots\times E_{i_k})$$

The operators are generalizations of the corresponding relational algebra operators. When all the attributes required by the operators are available in the base of $M$, they are regular select and project operators. The operator $\sigma_{C_M^B}$ selects from the Cartesian product all tuples which are guaranteed not to be spurious. The condition $C_M^B$ is called the *back-join condition* of $M$. The operator $\sigma_{C_M^W}$ then extracts all tuples that can be shown to satisfy the query condition $C_q$. The condition $C_M^W$, called the *weakest safe selection condition* of $M$, accepts the maximal set of tuples that can be safely accepted. Finally, the operator $\pi_{M(\mathbf{A}_q)}$ projects the tuples onto the desired set of attributes, in the process reconstructing values for required but missing attributes whenever possible, and discarding tuples for which this cannot be done.

### 3.1. Back-joins

We already saw in the example above the need for back-joins. In the example, $E_1$ provided all the required tuples, but attribute $R.B$ was missing. The missing attribute could be obtained from $E_2$ and $E_3$ (in fact, both were required). Let $t_1 = (a_1, c_1, e_1)$ be a tuple from $E_1$ and $t_2 = (a_2, b_2, c_2)$ a tuple from $E_2$. How do we guarantee that $t_1$ and $t_2$ originate from the same tuple $t$ in $R$? If they do not, and we accept the tuple $(b_2, c_1, e_1)$ formed from $t_1$ and $t_2$ into the result set, then we have a spurious tuple in the result. The tuple is spurious if no tuple of the form $(x, a_2, c_1)$, for some value of $x$, exists in the current instance of $R$. We cannot guarantee that a result set containing spurious tuples is correct because the original query $Q$ does not generate spurious tuples. Not surprisingly, we show in [LY87] that spurious tuples can be avoided if all back-joins are over keys. (There are a few other cases but they are of minor importance in practice).

Let $key(R)$ denote the attributes of the key of a relation $R$. Now define the following set of derived relations:

$U_M(R) = \{ E_k : E_k$ is in the base of $M$ and an attribute of $R$ in $attr(Q)$ is mapped by $M$ into an attribute in $attr(E_k)\ \}$

$M$ specifies a back-join over $R$ if $U_M(R)$ contains more than one derived relation. The back-joins over $R$ are *safe* (lossless) if $key(R)\subseteq \mathbf{A}_k^+$ for every derived relation $E_k\in U_M(R)$. The mapping $M$ is safe if this holds for every relation $R\in\mathbf{R}_q$. To guarantee that a tuple from the base of $M$ is not spurious it must satisfy the following *back-join condition*:

$$C_M^B = \bigwedge_{R_j\in\mathbf{R}_q}\ \bigwedge_{\substack{E_s,E_t\in U_M(R_j)\\ E_s\neq E_t}}\ \bigwedge_{A_k\in key(R_j)}(E_s.R_j.A_k = E_t.R_j.A_k)$$

This condition involves only visible attributes and is hence easy to test. If two tuples from $R$ agree on the key of $R$, they also automatically agree on all other attributes. Hence every tuple that satisfies the back-join condition above also satisfies the following *extended back-join condition*:

$$C_M^{EB} = \bigwedge_{R_j\in\mathbf{R}_q}\ \bigwedge_{\substack{E_s,E_t\in U_M(R_j)\\ E_s\neq E_t}}\ \bigwedge_{A_k\in\alpha(R_j)}(E_s.R_j.A_k = E_t.R_j.A_k)$$

*Example*: The example in the beginning of this section used two mappings, which both specified a back-join over $R$. The key of $R$ is the single attribute $A$. The back-join condition and the extended back-join condition for the first mapping are then

$$C_M^B = (E_1.R.A = E_2.R.A)$$
$$C_M^{EB} = (E_1.R.A = E_2.R.A)\wedge(E_1.R.B = E_2.R.B)\wedge(E_1.R.C = E_2.R.C)$$

and for the second mapping

$$C_M^B = (E_1.R.A = E_3.R.A)$$
$$C_M^{EB} = (E_1.R.A = E_3.R.A)\wedge(E_1.R.B = E_3.R.B)\wedge(E_1.R.C = E_3.R.C)$$

### 3.2. Tuple selection

Once a tuple from the base of a mapping $M$ has been shown to satisfy the back-join condition we know that it is not spurious. The next step is to determine whether it satisfies the query condition $C_q$. If all the attributes mentioned in $C_q$ are mapped to visible attributes, we can simply substitute in the corresponding values and evaluate the condition. However, even when some attributes mentioned in $C_q$ are not mapped to visible attributes, we may still be able to determine that the tuple satisfies $C_q$. This is illustrated by the following example.

*Example*: Consider the following query and derived relation defined over the relation $R(A, B, C)$:

$$Q = (\{A, B\}, \{R\}, (C>10)(B<5)\ )$$
$$E = (\{A, B\}, \{R\}, (C>A)\ )$$

There are no other attribute mappings than the obvious one which maps all attributes of $Q$ into the corresponding attributes of $E$. We cannot guarantee that the query can be computed from $E$ alone, but we may be able to extract some tuples from $E$ towards the result of the query. The condition $(C>10)$ in the query cannot be tested directly because attribute $C$ is not visible in $E$. However, it is easy to see that any tuple in $E$ with an $A$-value greater than or equal to 10 must be the projection of a tuple with a $C$-value greater than 10. Hence we can safely accept any tuple where $A \geq 10$, provided that it satisfies the additional condition $(B<5)$. On the other hand, it is unsafe to accept any tuple where $A<10$. We simply cannot decide whether or not such a tuple satisfies the query condition.

Let $t = (a, b)$ be a tuple from $E$. To accept $t$ into the query result we must prove that the following condition holds

$$\forall E.R.C\ (E.R.C>a \Rightarrow (E.R.C>10)(b<5))$$

This condition can be paraphrased as follows. Whatever the missing C-value of $t$ was, it must have been such that $t$ satisfied the condition of $E$. Otherwise, the tuple would not be in $E$ at all. If, for every such $C$-value, it follows that $t$ must also satisfy the query condition, then we can safely accept $t$ into the result,

To further clarify the idea, consider the following instance of $E$.

$E$:

| A | B |
|---|---|
| 15 | 5 |
| 12 | 3 |
| 6 | 0 |

For the tuple (15,5) we get the condition

$$\forall E.R.C\ (E.R.C>15 \Rightarrow (E.R.C>10)(5<5))$$

The implication does not hold because the consequent is always *false*. Hence the tuple is rejected, which is obviously the correct decision because it does not satisfy the condition $(B<5)$ of the query.

For the tuple (12,3) we get

$$\forall E.R.C\ (E.R.C>12 \Rightarrow (E.R.C>10)(3<5))$$

It is easy to see that the implication holds and hence the tuple is accepted. The tuple satisfies the condition $(B<5)$ and the missing $C$-value must have been greater than 12.

For the tuple (6,0) we have

$$\forall E.R.C\ (E.R.C>6 \Rightarrow (E.R.C>10)(0<5))$$

The implication does not hold and the tuple is rejected. Here we have an unsafe tuple. The $C$-value may or may not have been greater than 10. To be on the safe side we must reject the tuple. □

Let us now return to the general case. Let $t$ be a tuple from the base of a mapping $M$, $\mathbf{B}_M = \{E_{i_1}, E_{i_2}, \cdots, E_{i_k}\}$, and assume that $t$ satisfies the back-join conditions of $M$. To guarantee that $t$ is the projection of a tuple satisfying the query condition $C_q$, $t$ must satisfy the following condition, called the *weakest safe selection condition*:

$$C_M^W = \forall((C_{i_1} \wedge C_{i_2} \wedge \cdots \wedge C_{i_k} \wedge C_M^{EB})[t] \Rightarrow M(C_q)[t])$$

The notation $M(C_q)$ means the condition $C_q$ where every attribute name has been substituted by its image under $M$. The above condition extracts the maximal set of tuples that can be safely extracted. If a tuple does not satisfy this condition we cannot guarantee that it satisfies the query condition.

When all attributes in $C_q$ are mapped to visible attributes, $M(C_q)[t]$ contains no variables and can be evaluated directly. The implication then holds if it evaluates to ***true***, otherwise not. In other words, all we have to do is to test whether the tuple satisfies the query condition or not. This is exactly what one would expect, of course.

### 3.3. Attribute reconstruction

Consider a mapping $M$ that maps some of the attributes in $\mathbf{A}_q$ to attributes not visible in the derived relations of the base of $M$. At first glance, it appears that such a mapping would be useless because we would not know the exact values for some of the attributes in $\mathbf{A}_q$. However, there are situations when we are able to correctly reconstruct the missing values. This is illustrated by the following example. Note that $B$ is not in the extended attribute set.

*Example*: Consider the following query and derived relation over $R(A, B)$.

$$Q = (\{A, B\}, \{R\}, (A>10)(A=B))$$

$$E = (\{A\}, \{R\}, (A>15)(A=B) \vee (A \leq 15))$$

As in the previous example, there are no other mappings than the obvious one. The query cannot be computed from $E$ alone, but we can extract a subset of the tuples needed. Tuples satisfying the condition $(A>10)$ can easily be extracted from $E$. For a subset of those, namely all tuples where $A>15$, we can reconstruct the missing value of attribute $B$ because they must have satisfied the condition $(A=B)$. Hence, we can obtain from $E$ all tuples satisfying $(A>15)(A=B)$. The remaining tuples, that is, tuples satisfying $(A>10)(A \leq 15)(A=B)$ must be found somewhere else.

Let $t = (a)$ be a tuple from $E$. To guarantee that the value of attribute $B$ is reconstuctible, $t$ must satisfy the following condition:

$$\forall E.R.B, E.R.B'$$
$$(\ \{(a>15)(a=E.R.B) \vee (a \leq 15)\} \wedge (a>10)(a=E.R.B)$$
$$\wedge \{(a>15)(a=E.R.B') \vee (a \leq 15)\} \wedge (a>10)(a=E.R.B')$$
$$\Rightarrow (E.R.B=E.R.B')\ )$$

Consider the following instance of $E$ where all tuples have been chosen so that they satisfy the condition $(A>10)$ of the query.

E:

| A |
|---|
| 20 |
| 12 |

For the first tuple we get the condition

$$\forall E.R.B, E.R.B'$$
$$(\ \{(20>15)(20=E.R.B) \vee (20 \leq 15)\} \wedge (20>10)(20=E.R.B)$$
$$\wedge \{(20>15)(20=E.R.B') \vee (20 \leq 15)\} \wedge (20>10)(20=E.R.B')$$
$$\Rightarrow (E.R.B=E.R.B')\ )$$

which can be simplified to

$\forall E.R.B, E.R.B'$

$$((20 = E.R.B)(20 = E.R.B') \Rightarrow (E.R.B = E.R.B'))$$

It is now easy to see that the implication holds and that the only possible value for $B$ is 20. Hence we add the tuple (20, 20) to the result.

The second tuple cannot be accepted because we cannot guarantee that it satisfies the condition $(A = B)$ of the query. Hence we need not test whether the value of $B$ is reconstructible. □

For the general case, the condition that a tuple must satisfy to guarantee reconstructability of missing attributes, is somewhat complex. Let $\mathbf{B}_M = \{E_{i_1}, E_{i_2}, \cdots, E_{i_k}\}$. The set of attributes in $\mathbf{A}_q$ mapped by $M$ into non-visible attributes is given by $I = M(\mathbf{A}_q) - (\mathbf{A}_{i_1} \cup \mathbf{A}_{i_2} \cup \cdots \cup \mathbf{A}_{i_k})$. Let $C_P$ denote the following condition

$$C_P = C_{i_1} \wedge C_{i_2} \wedge \cdots \wedge C_{i_k} \wedge C_M^{EB} \wedge M(C_q)$$

Now consider a tuple $t$ over the visible attributes in the base of $M$, that is, over the attributes in $\mathbf{A}_{i_1} \cup \mathbf{A}_{i_2} \cup \cdots \cup \mathbf{A}_{i_k}$. The values of the attributes in $I$ can be reconstructed if $t$ satisfies the condition

$$\forall Y \forall Y' ((C_P[t])(Y)(C_P[t])(Y') \Rightarrow \bigwedge_{y_j \in I} (y_j = y'_j))$$

where $Y = \alpha(C_P[t]) \cup I = \{y_1, y_2, \cdots\}$ and $Y' = \alpha(C_P[t]) \cup I = \{y'_1, y'_2, \cdots\}$. The expression $(C_P[t])(Y)$ denotes the partial evaluation of $C_P$ with respect to $t$, evaluated over the variables in $Y$, and similarly for $(C_P[t])(Y')$. A general reconstruction procedure is outlined in [LY85] and given in more detail in [LY87]. Note that the condition trivially holds and need not be tested when $I = \emptyset$, that is, when every attribute in $\mathbf{A}_q$ is mapped to an attribute in $\mathbf{A}_{i_1}^+ \cup \mathbf{A}_{i_2}^+ \cup \cdots \cup \mathbf{A}_{i_k}^+$.

### 3.4. Sufficient sets of mappings

Given a query $Q$ and a set of derived relations $\{E_1, E_2, ..., E_m\}$, there may be many ways of mapping the attributes of the query into the corresponding attributes of the derived relations. As shown above, each mapping defines a function over the derived relations in its base and contributes some tuples towards the result of the query. The problem then is to find a ***sufficient set*** of mappings, that is, a set that is guaranteed to generate all the required tuples. We therefore need some criterion for deciding whether a set of mappings is sufficient.

Consider a set $\mathbf{M} = \{M_1, M_2, ..., M_s\}$ of attribute mappings, mapping the attributes of $Q$ into attributes of the derived relations $E_1, E_2, ..., E_m$. The conceptual relations of interest are $\mathbf{B} = \mathbf{R}_q \cup \mathbf{R}_1 \cup \cdots \cup \mathbf{R}_m$. Assume that $\mathbf{B} = \{R_1, R_2, ..., R_n\}$. Let $t_1, t_2, ..., t_n$ be tuples over $R_1, R_2, ..., R_n$, respectively, $t = (t_1 \times t_2 \times \cdots \times t_n)$ and assume that $C_q(t) = \mathit{true}$. (In fact, we only need to consider a tuple defined over the attributes in $\mathbf{A}_q \cup \alpha(C_1) \cup \cdots \cup \alpha(C_m)$.) If the query were computed directly from the original query expression, $t[\mathbf{A}_q]$ (the projection of $t$ onto $\mathbf{A}_q$) would occur in the result and we must show that it will also be contributed by one of the mappings.

For mapping $M_k$ to contribute $t[\mathbf{A}_q]$ to the response set, $t$ must satisfy all the conditions associated with the function $F_{M_k}$. Let $\mathbf{B}_{M_k} = \{E_{i_1}, E_{i_2}, ..., E_{i_{m_k}}\}$ denote the base of mapping $M_k$, $C_{M_k}^P \equiv C_{i_1} \wedge C_{i_2} \wedge \cdots \wedge C_{i_{m_k}} \wedge C_{M_k}^{EB}$, and $t' = t[\mathbf{A}_{i_1} \cup \mathbf{A}_{i_2} \cup \cdots \cup \mathbf{A}_{i_{m_k}}]$. The conditions associated with $F_{M_k}$ that $t$ must satisfy are:

1. $t$ must exist in the base of $M_k$, that is, satisfy
$$C_{M_k}^s \equiv (\bigwedge_{E_j \in \mathbf{B}_{M_k}} C_j)[t]$$

2. $t$ must satisfy the weakest safe selection condition associated with $M_k$, that is,
$$C_{M_k}^W \equiv \forall X (C_{M_k}^P[t'] \Rightarrow (M_k(C_q))[t'])$$
where all variables in the set $X = \bigcup_{E_j \in \mathbf{B}_{M_k}} (attr(E_j) - \mathbf{A}_j)$ are universally quantified. Note that the conditions $C_{M_k}^P$ and $M_k(C_q)$ are over attributes having three-part names. However, after the partial evaluation with respect to $t'$ and quantification, the resulting condition is just a function of $t$, that is, a function of the attribute values in $t$.

3. $t$ must satisfy the condition guaranteeing reconstructability of all the attributes in $\mathbf{A}_q$ mapped to non-visible attributes, that is,
$$C_{M_k}^R \equiv \forall Y \forall Y' (((C_{M_k}^P M_k(C_q))[t'])(Y) \wedge ((C_{M_k}^P M_k(C_q))[t'])(Y') \Rightarrow \bigwedge_{y_j \in I} (y_j = y'_j))$$
where $I = M_k(\mathbf{A}_q) - \bigcup_{E_j \in A_k} \mathbf{A}_j$ and all variables in $Y = Y' = \alpha((C_{M_k}^P M_k(C_q))[t']) \cup I$ are universally quantified. Note again that the resulting condition is just a function of the attribute values in $t$.

If a set of mappings is suffient, then for any tuple $t$ over the attributes in $\mathbf{B} = \mathbf{R}_q \cup \mathbf{R}_1 \cup \cdots \cup \mathbf{R}_m$ that satisfies the query condition, the tuple must also be generated by one of the mappings in the set. We show in [LY87] that $\mathbf{M} = \{M_1, M_2, \cdots, M_s\}$ is sufficient if and only if the following condition holds:

$$\forall (C_q \Rightarrow \bigvee_{k=1}^{s} C_{M_k}^s \wedge C_{M_k}^W \wedge C_{M_k}^R)$$

Once we have a set of mappings we can use this condition to test sufficiency. However, finding a sufficient set of mapping, or showing that no such set exists, may be expensive. This problem is addressed in the next section.

## 4. Reducing the set of mappings

For a given query $Q$ and a set of derived relations, there may be many possible ways of mapping the attributes of $Q$ into the corresponding attributes of the derived relations. In this section we show how to reduce the number of mappings that need be considered. We give necessary and sufficient conditions for detecting when a mapping is subsumed by another mapping, and for detecting when a mapping does not generate any tuples at all.

### 4.1. Partial ordering of mappings

Let $M$ be an attribute mapping, relating the attributes of $Q$ to attributes of derived relations in the set $\{E_1, E_2, \ldots, E_n\}$. Denote the base of $M$ by $\mathbf{B}_M$ and assume that $\mathbf{B}_M = \{E_{i_1}, E_{i_1}, \ldots, E_{i_k}\}$. $M$ defines a function over $E_{i_1} \times E_{i_2} \times \ldots \times E_{i_k}$ which we denote by $F_M$. The relation resulting from evaluating this function over a database instance $d$ is denoted by $V(F_M, d)$. Based on the set of tuples generated by the associated function, we can define a partial ordering on the set of mappings between $Q$ and $\{E_{i_1}, E_{i_1}, \ldots, E_{i_k}\}$. We consider only safe mappings.

**Definition:** Let $M_1$ and $M_2$ be attribute mappings associated with a query $Q$.

(i) $M_1$ is **inferior** to $M_2$, denoted by $M_1 \leq M_2$, if $V(F_{M_1}, d) \subseteq V(F_{M_2}, d)$ for every database instance $d$.

(ii) $M_1$ is **superior** to $M_2$, denoted by $M_1 \geq M_2$, if $V(F_{M_1}, d) \supseteq V(F_{M_2}, d)$ for every database instance $d$.

(iii) $M_1$ is **equivalent** to $M_2$, denoted by $M_1 \equiv M_2$, if $V(F_{M_1}, d) = V(F_{M_2}, d)$ for every database instance $d$.

(iv) $M_1$ is a **null-mapping**, denoted by $M_1 \equiv \emptyset$, if $V(F_{M_1}, d) = \emptyset$ for every database instance $d$.

If we can show that a mapping $M_1$ is inferior to another mapping $M_2$, then $M_1$ can be discarded immediately. If two mappings are equivalent, either one of them can be used. The following theorem enables us to compare two mappings. Note that in the conditions of the theorems and corollaries of this section, two-part variable names (without the name of the derived relation) must be used for the variables quantified by the outermost quantifier. However, for all variables quantified by the quantifier which is part of $C_M^W$ or $C_M^R$, three-part names are still required.

**Theorem 1:** Consider two safe attribute mappings $M_1$ and $M_2$. Then $M_1 \leq M_2$ if and only if

$$\forall (C_{M_1}^{*} \wedge C_{M_1}^{W} \wedge C_{M_1}^{R} \Rightarrow C_{M_2}^{*} \wedge C_{M_2}^{W} \wedge C_{M_2}^{R}).$$

**Proof sketch:** Let $t = (t_1 \times t_2 \times \ldots \times t_l)$ be a tuple from the Cartesian product of the underlying base relation $R_1, R_2, \ldots, R_l$. Then the projection of $t$ onto $\mathbf{A}_q$ is contributed to the result set by $M_1$ if and only if $t$ satisfies $C_{M_1}^{*} \wedge C_{M_1}^{W} \wedge C_{M_1}^{R}$, and similarly for $M_2$. If the condition holds, every tuple contributed by $M_1$ will also be contributed by $M_2$. If the condition does not hold, we can easily construct a database instance (containing one tuple for each of $R_1, R_2, \ldots, R_l$) such that $M_1$ contributes one tuple to the response set of $Q$ but $M_2$ contributes none. This then proves the theorem. □

**Corollary 1.1:** Consider two safe attribute mappings $M_1$ and $M_2$. Then $M_1 \equiv M_2$ if and only if

$$\forall (C_{M_1}^{*} \wedge C_{M_1}^{W} \wedge C_{M_1}^{R} \Leftrightarrow C_{M_2}^{*} \wedge C_{M_2}^{W} \wedge C_{M_2}^{R}).$$

**Proof:** Follows by observing that $M_1$ and $M_2$ are equivalent if and only $(M_1 \leq M_2) \wedge (M_2 \leq M_1)$, and applying Theorem 1. □

**Corollary 1.2:** $M_1 < M_2$ if the following conditions all hold:

$$\forall (C_{M_1}^{*} \Rightarrow C_{M_2}^{*}),\ \forall (C_{M_1}^{W} \Rightarrow C_{M_2}^{W}),\ \forall (C_{M_1}^{R} \Rightarrow C_{M_2}^{R})$$

**Proof**: Follows by applying the rule $((a \rightarrow b) \wedge (c \rightarrow d)) \Rightarrow (ac \rightarrow bd)$.

**Corollary 1.3:** $M_1 \equiv M_2$ if the following conditions all hold:

$$\forall (C_{M_1}^{*} \Leftrightarrow C_{M_2}^{*}),\ \forall (C_{M_1}^{W} \Leftrightarrow C_{M_2}^{W}),\ \forall (C_{M_1}^{R} \Leftrightarrow C_{M_2}^{R})$$

**Proof**: Follows directly from corollary 1.2.

**Theorem 2:** If two mappings $M_1$ and $M_2$ have the same base and specify the same back-joins, then $M_1 \equiv M_2$.

**Proof idea**: Proved by showing that the three conditions of corollary 1.3 are satisfied.

Using this theorem we can significantly reduce the number of mappings to be considered. Let $R$ be a relation that occurs in several derived relations in the base of a mapping. A subset of those will be back-joined over $R$. Then it does not matter to which derived relation in the subset we map the attributes of $R$. The resulting mappings are all equivalent, and hence we need consider only one of them. To simplify the weakest safe selection condition and the condition for attribute reconstructability, we always use one that maps as many attributes as possible to visible attributes.

**Theorem 3**: Let $M$ be a mapping which maps all attributes of the query into visible attributes. Then any mapping obtained from $M$ by adding another derived relation to the base of $M$ is inferior to $M$.

**Proof**: $C_M^R = true$ and $C_M^W = C_q$ because all attributes in the query are mapped to visible attributes. Let $M_1$ be a mapping with base $\mathbf{B}_M \cup \{E_i\}$. Then $C_{M_1}^{*} = C_M^{*} \wedge C_i$. and the condition $\forall (C_{M_1}^{*} \Rightarrow C_M^{*})$ trivially holds. The condition $\forall (C_{M_1}^{W} \Rightarrow C_M^{W})$ holds because $C_M^W = C_q$. The condition $\forall (C_{M_1}^{R} \Rightarrow C_M^{R})$ holds because $C_M^R = true$. The theorem then follows from corollary 1.2. □

This theorem is useful as a stopping condition when generating attribute mappings. Once all attributes have been mapped to visible attributes, there is no point in further augmenting the base by additional derived relations. Note that the theorem does not (necessarily) hold when some attributes of the query are mapped to non-visible attributes.

### 4.2. Detecting null-mappings

A null-mapping is a mapping that never generates any tuples to the result. The following theorem states necessary and sufficient conditions for a mapping to be a null-mapping.

**Theorem 4:** A safe attribute mapping $M$ is a null-mapping if and only if the condition $C_M^* \wedge C_M^W \wedge C_M^R$ is unsatisfiable.

**Proof sketch:** From the discussion in section 3.4 it is clear that a tuple $t$ must satisfy the three conditions above in order to qualify for the response set of $Q$. If the condition above is unsatisfiable no tuple satisfies it, and hence $M$ is a null-mapping. On the other hand, if the condition above is satisfiable, we can construct a database instance such that $M$ contributes one tuple to the response set. □

Provided that the satisfiability of the condition $C_M^* \wedge C_M^W \wedge C_M^R$ can be tested at run-time, we can use this theorem to detect null-mappings. However, this is not always possible because $C_M^W$ and $C_M^R$ may contain universally quantified variables. Let $C(x, y)$ be a Boolean expression over variables $x$ and $y$, and assume that $y$ is universally quantified. The condition $\forall y C(x, y)$ is then unsatisfiable if $\nexists x(\forall y C(x, y))$. We have the following equivalences:

$$\nexists x(\forall y C(x, y)) \equiv \forall x \neg (\forall y C(x, y)) \equiv \forall x \exists y \neg C(x, y).$$

We know of no efficient, general algorithm for testing conditions with mixed universal and existential quantifiers. When the quantifiers are all either universal or existential the algorithm in [LY85, LY87] can be used. Hence, the condition can be tested when $C_M^W$ and $C_M^R$ contain no quantified variables or when the quantified variables can be eliminated.

**Corollary 4.1**: A mapping $M$ is a null-mapping if any one of the conditions $C_M^*$, $C_M^W$, or $C_M^R$ is unsatisfiable.

This is an extremely useful corollary. The condition $C_M^*$ contains no quantified variables and can be tested using the algorithm in [LY85, LY87]. Note that $C_M^*$ is just the conjunction of all the selection conditions of the derived relations in the base of $M$. We could speed up detection of null-mappings even further by keeping track of which derived relations have contradictory selection conditions. If the base contains two derived relations with contradictory conditions, then the mapping is a null-mapping.

The conditions $C_M^W$ and $C_M^R$ are more difficult to test for satisfiability. As discussed above, the problem is that they may include universally quantified variables and we have no general algorithm for testing the satisfiability of a universally quantified expression. The following corollaries identify a number of special cases that can be handled successfully. Both $C_M^W$ and $C_M^R$ are of the form

$$\forall Y (C_a(X, Y) \Rightarrow C_c(X, Y)) \qquad (1)$$

where $Y$ is the set of universally quantified variables and $X$ is the set of variables not quantified. We therefore consider only conditions of the above type.

**Corollary 4.2:** Assume that $C_a(X, Y) = C_a^1(X) \wedge C_a^2(Y)$ and $C_c(X, Y) = C_c^1(X) \wedge C_c^2(Y)$. If $\exists Y \neg (C_a^2(Y) \Rightarrow C_c^2(Y))$, then condition (1) is equivalent to $\neg C_a^1(X)$.

This corollary enables us to eliminate the universal quantification in certain cases. The condition $\exists Y \neg (C_a^2(Y) \Rightarrow C_c^2(Y))$ is equivalent to $\neg ((\forall_Y (C_a^2(Y) \Rightarrow C_c^2(Y)))$, which can be tested efficiently.

**Corollary 4.3:** Assume that $C_a(X, Y) = C_a^1(X) C_a^2(Y)$ and $C_c(X, Y) = C_c^{11}(X, Y) C_c^{12}(Y) \vee C_c^{21}(X, Y) C_c^{22}(Y)$. If $\exists Y \neg (C_a^2(Y) \Rightarrow C_c^{12}(Y) \vee C_c^{22}(Y))$, then condition (1) is equivalent to $\neg C_a^1(X)$.

## 5. A prototype implementation

We have implemented a prototype system for query transformation based on the concepts discussed in the previous sections. It runs on a VAX 11/780 and consist of approximately 8000 lines of C code, divided into four major parts.

*User interface*: The user interface was deliberately kept simple in this first prototype. Conceptual relations are defined by listing the relation name, attribute names, and candidate keys. Stored relations are defined using the triple representation. Queries are defined by relational algebra expressions, using a syntax similar to that in [DA86]. A query can be issued as a sequence of simpler queries using intermediate variables. The output from this stage is an operator tree representation of the query.

*Handling of Boolean expressions*: This part handles storage, conversion, full and partial evaluation, and validity testing of Boolean expressions. The most crucial operation is validity testing, which is done using the algorithm presented in [LY85, LY87]. The current version restricts atomic conditions to a comparison between two variables or a comparison between a variable and a constant. All the normal comparison operators are handled.

*Generation of candidate mappings*: For each relation in the base of the query, the set of source relations is identified. A derived relation is a potential source for a conceptual relation in the query if the conceptual relation occurs in the base of the derived relation. For each source relation we have a partial mapping of the attributes in the query. Every combination of partial mappings covering all the relations in the base of the query is then a candidate mapping. Candidate mappings which are complete, that is, contain all the attributes in the query are kept. Candidate mappings which are not complete, but which can potentially be augmented by safe back-joins are also kept (deficient mappings). Incomplete mappings that cannot be augmented by safe back-joins are rejected immediately.

Candidate mappings are also tested to determine whether the join conditions of the query are realizable, either by join conditions inherited from the derived relation or by explicitly forming the needed join conditions. Candidate mappings satisfying this requirement are said to be join compatible with the query. Mappings that are not join compatible with the query are discarded. (This is a simplification. In practice, such mappings are of limited use, but they are not always null-mappings or subsumed by other mappings.)

*Final query transformation*: The candidate mappings generated in the previous stage are sorted according to the number of derived relations involved and the number of join conditions satisfied. Before any mapping is added to this list, it is tested to determine whether it is a null-mapping, and, if so, it is discarded. The idea is to consider the least expensive mappings first. The more derived relations in the base, the more joins will be required and the more expensive the execution of the corresponding query is likely to be. Deficient mappings are augmented at this stage to make them complete. This involves introducing additional derived relations into the base of the mapping and adding back-join conditions.

To find a sufficient set of mappings, the next mapping from the sorted list is extracted and added to the set of mappings considered so far. At each step, the current set of extracted mappings is then tested to determine whether the set is sufficient. If the total set of candidate mappings is not sufficient, the query cannot be computed from the given derived relations.

The current implementation is intended merely as a "proof-of-concept" prototype. Its main role is as a learning tool used for experimental purposes. We need to find out which steps in the transformation process are the most expensive ones and what types of queries are difficult to transform. Based on the experience gained from the prototype we plan to implement a more comprehensive and more efficient version later on.

To facilitate implementation several simplifications were made. The most important ones are listed below.

- All attributes are restricted to integer domains.
- Only two cases of uniquely determined attributes are handled: an attribute equal to another attribute and an attribute equal to a constant.
- Every visible attribute in a query must be mapped to a visible attribute in a derived relation. For such mappings the condition for attribute reconstructability is trivially satisfied and need not be tested.
- Detection of null-mappings is restricted to testing whether the condition $C_M^* \wedge C_M^W$ is satisfiable.
- No attempt is made to eliminate superfluous mappings from the final set of mappings. A mapping is superfluous if it can be discarded and the remaining set still is sufficient.

We illustrate the performance of the prototype by a few example queries. All queries are against the following database.

*Conceptual relations*:
R1( x1, y1, z1, t1 ), key x1
R2( x2, y2, z2 ), key x2
R3( x3, y3, z3, t3 ), key x3

*Derived relations*:
$E_1$ = ( {x1, y1, z1, t1 }, {R1}, y1>10 )
$E_2$ = ( {x1, y1, z1 }, {R1}, y1<50 )
$E_3$ = ( {x1, y1, t1 }, {R1}, y1<50 )

$E_4$ = ( {x2, y2, y3, z3, t3 }, {R2, R3}, (x2=x3)∧(y3>0) )
$E_5$ = ( {x2, y2, y3, z3, t3 }, {R2, R3}, (x2=x3)∧(y3<30) )

$E_6$ = ( {x2, y2, z2 }, {R2}, z2<0 )
$E_7$ = ( {x2, y2, z2 }, {R2}, z2≥0 )

Relations $E_1$ to $E_3$ are horizontal and vertical partitionings of R1. Note that the conditions of $E_2$ and $E_3$ overlap the condition of $E_1$, and that $E_2$ and $E_3$ do not contain all the attributes of R1. Relations $E_4$ and $E_5$ are essentially the join of R2 with two horizontal partitionings of R3. Again, note the overlap of the condition on y3 and that attribute z2 is missing from both $E_4$ and $E_5$. $E_6$ and $E_7$ are straight horizontal partitionings of R2. It is assumed that the conceptual relations are not available in stored form.

*Query 1:* $Q = \sigma_{y2>20}(R2)$

Mappings generated:
M$_1$: $Q \rightarrow E_6$
M$_2$: $Q \rightarrow E_7$
Cpu-time: 0.2 s

The notation $Q \rightarrow E_6$ means that all attributes in $Q$ are mapped to the corresponding attributes in $E_6$. For this query the system generated two mappings, which is clearly a minimal set. The transformed query is $Q = \sigma_{E_6.R2.y2>20}(E_6) \cup \sigma_{E_7.R2.y2>20}(E_7)$

*Query 2:* $Q = \pi_{x1,\,z1,\,t1}\sigma_{z1>0}(R1)$

Mappings generated:
M$_1$: $Q \rightarrow E_1$
M$_2$: $Q \rightarrow E_3( (E_1)R1)$
M$_3$: $Q \rightarrow E_3( (E_2)R1)$
Cpu-time: 0.4 s

The notation $E_3( (E_1)R1)$ means the back-join of $E_3$ and $E_1$ over R1, and similarly for $E_3( (E_2)R1)$. The back-joins in $M_2$ and $M_3$ are necessary bacause z1 is missing from $E_3$. Mapping $M_2$, even though not a null-mapping, is not required. In fact, it is inferior to both $M_1$ and $M_3$. This example shows the need for detection of superfluous mappings. However, this has not been implemented in the current version of the prototype.

*Query 3:* $Q = \pi_{x2,\,z2,\,z3,\,t3}\sigma_{y3>50}(R2 \bowtie_{x2=x3} R3)$

Mappings generated:
M$_1$: $Q \rightarrow E_4( (E_6)R2, R3)$
M$_2$: $Q \rightarrow E_4( (E_7)R2, R3)$
Cpu-time: 1.0 s

The two mappings generated are both necessary to transform the query. The back-joins are required because attribute z2 is missing from $E_4$. In the process of transforming the query two null-mappings were detected and discarded: $Q \rightarrow E_5( (E_6)R2, R3)$ and

$Q \rightarrow E_5( (E_7)R2, R3)$.

*Query 4:* $Q = \pi_{x2,\, z2,\, z3,\, t3}\sigma_{y3>0}(R2 \times_{x2=x3} R3)$

Mappings generated:

$M_1$: $Q \rightarrow E_5( (E_6)R2, R3)$
$M_2$: $Q \rightarrow E_5( (E_7)R2, R3)$
$M_3$: $Q \rightarrow E_4( (E_6)R2, R3)$
$M_4$: $Q \rightarrow E_4( (E_7)R2, R3)$

Cpu-time: 28 s

This query is similar to query 3; the only change is in the condition on y3. However, the transformation time increased dramatically. Mappings $M_1$ and $M_2$ are superfluous but not null-mappings. Again the need for detecting superfluous mappings is seen. It was found that virtually all of the additional time was spent on proving the sufficiency of the final set of mappings. This and other examples clearly indicate that the sufficency test is the most expensive step. The cost increases dramatically with the number of mappings in the set. We are currently investigating ways of reducing the cost of this step.

*Query 5:*

$Q = \pi_{y1,\, y2,\, z2,\, y3}\sigma_{(y1>60)\wedge(z2>10)}(R1 \times_{x1=x2} R2 \times_{x2=x3} R3)$

Mappings generated:

$M_1$: $Q \rightarrow E_1(R1) \times E_4( (E_7)R2, R3)$
$M_2$: $Q \rightarrow E_1(R1) \times E_5( (E_7)R2, R3)$

Cpu-time: 2.8 s

This is a fairly complex query where the two mappings generated both involve three derived relations. To get attribute z2 both $E_4$ and $E_5$ must be augmented by a back-join of $E_7$. The tuples required from R1 can all be obtained from $E_1$ but the join represented by x1=x2 has to be explicitly performed. In the process of transforming this query, six null-mappings were detected and discarded.

## 6. Concluding remarks

For queries and derived relations defined by *PSJ*-expressions the query transformation problem essentially boils down to finding a sufficient set of attribute mappings. Given an attribute mapping, we showed how to construct a function that extracts the maximal set of tuples that can safely be extracted towards the result of the query. This function correponds to a generalized select-project expression. A set of attribute mappings is sufficient if the union of the corresponding functions is guaranteed to generate all the tuples required to answer the query, and we showed how to test a set of mappings for sufficiency.

The number of attribute mappings corresponding to a query may be large. In the second part of the paper we showed that many of the mappings can be eliminated, either because they are subsumed by other mappings or because they do not generate any tuples at all.

Finally we gave a brief overview of a prototype system for query transformation, including a few examples. Even from these few examples, it was clear that the most expensive part of transforming a query is proving the sufficiency of a set of mappings. The current version of the prototype does not attempt to eliminate superfluous mappings from the final set. We are currently working on eliminating these deficiencies of the prototype.

## 7. References


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

BA82 Babb, E., Joined Normal Form: A storage encoding for relational databases, ACM TODS 7, 4 (1982), 588-614.

BC86 Blakeley, J.A., Coburn, N., and Larson, P.-Å, Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates, Technical Report CS-86-17, University of Waterloo, 1986.

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

DA86 Date, C.J., An Introduction to Database Systems, Volume I, 4th ed., Addison-Wesley, Reading, Massachusetts, 1986.

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

LY85 Larson, P.-Å and Yang, H.Z., Computing Queries from Derived Relations, Proc. of 11th Intl. Conf. on Very Large Data Bases, (1985), 259-269.

LY87 Larson, P.-Å and Yang, H.Z., Computing Queries from Derived Relations: Theoretical Foundation, Technical Report CS-87-35, University of Waterloo, 1987.

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 Int. Conf. on Very Large Data Bases, ACM, New York, N.Y., (1980), 64-72.

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

RO86 Roussopoulos, N. and Kang H., Preliminary Design of ADMS±, Proc. of 12th Intl. Conf. on Very Large Data Bases, (1986), 355-364.

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).

SE86 Sellis, T.K., Global Query Optimization, Proc. 1986 SIGMOD Conf. on Management of Data, 1986, 191-205.