Logical Optimisation of Distributed Knowledge Base Queries

M. C. TAYLOR*
Department of Computer Science, University of Keele, Keele, Staffs.

Query optimisation is a crucial element in providing acceptable performance in a distributed knowledge base system. This paper considers optimisation in a heterogeneous environment where each node provides an interface to a common model. The common model is object-oriented, using unnormalised relations with an extended relational algebra as query language. Both the data structures and the language are formally defined using the Vienna Development Method (VDM). These formal definitions are then used as a basis for deriving conditions under which various optimising transformations can be applied.

Received November 1988, revised July 1989

1. INTRODUCTION

The last few years have seen a rapid growth in knowledge base research. Considerable progress has been made, and the scope of applications has widened. Some of these applications require a very substantial store of data, which has led to the idea of storing the knowledge in a conventional database. Such approaches include Prolog interfaces to relational databases. Alternatively, a tightly-coupled system such as LDL,[14] POSTGRES[11] or DEAL[6] can be used to provide integrated data and knowledge processing. Traditional database approaches to performance issues, such as query optimisation and indexing techniques, can then be used to provide efficient processing.

The integration of data and knowledge processing creates the potential for the development of distributed knowledge bases (DKBs) as an extension of the concept of distributed databases. When the required knowledge is spread over more than one knowledge base, a DKBS can extend the range of knowledge processing.

In its most general form, a DKBS will allow pre-existing knowledge bases to join as nodes, irrespective of their underlying knowledge representation model and other potential sources of incompatibility. Clearly a good deal of progress must be made before these generalised systems become available, but such a generality is an important goal since the linking of pre-existing knowledge bases is the primary motivation for DKBs.

A heterogeneous DKBS requires a common knowledge model in whose terms all internodal messages must be expressed. Each node needs to provide a translator between this common model and its own model. The ideas developed in this paper assume that DEAL will be used as the common model. DEAL (DEductive ALgebra) is an extended relational algebra capable of deductive and recursive processing on unnormalised relations (hence supporting complex objects).[6,7] It is believed that DEAL has the features that are necessary in a common model, in particular combining the power of frame languages and Prolog.

Since the query language is based loosely on relational algebra, it is possible to use some established ideas from distributed database research in developing an approach to query optimisation. In distributed databases, three distinct phases of query optimisation have been identified:

(i) logical optimisation
(ii) execution strategy selection
(iii) local access strategy selection

The first phase refers to the process of determining the order in which operations should be carried out, without regard to which node should execute any particular operation. The second phase allocates each operation to a specific execution node. The final phase is concerned with the way in which a particular execution node will carry out an operation. Usually this phase is considered to be outside the control of the distributed system, and solely the concern of the executing node itself (at least in systems which allow pre-existing databases as nodes). We propose to adopt the same approach for a DKBS.

This paper is concerned solely with the first phase, viz logical optimisation. Generally it is possible to find more than one sequence of operations that will produce the required results. A set of transformations can be found, which can be applied to a query expression to yield logically equivalent expressions. For each transformation, we need to discover the conditions under which it is valid and, of course, whether it can be expected to bring about a more efficient execution of the query. For the latter, we shall adopt the heuristics developed in [5]. In order to check the validity of each transformation, we have formally defined the semantics of DEAL, using the Vienna Development Method (VDM). The meaning of each construct of the language is precisely defined by pre and post-conditions, and thus two sequences of operations can be shown to be logically equivalent (or not). A similar approach has been shown to work well for relational algebra queries to a distributed database,[12] but the extension to a knowledge base environment brings added complexity.

The remainder of this paper is structured as follows. In section 2 we describe an architecture for a distributed knowledge base – the approach to optimisation assumes this architecture to be used for the system. In section 3 we describe DEAL, and indicate how it can handle the basic requirements of Prolog and frame languages. Section 4 describes VDM, which is then used in the following section to define the semantics of DEAL. Section 6 discusses possible optimising transformations

* Present address: Department of Computer Science, University of Houston, Texas, USA.

for DEAL queries, in line with the heuristics adopted. In section 7 we describe the use of VDM to formally derive the conditions under which each of the optimising transformations is valid. Section 8 is the conclusion.

2. DISTRIBUTED KNOWLEDGE BASE ARCHITECTURE

We envisage a DKBS which will support as nodes pre-existing knowledge bases, each potentially using a different knowledge representation model. To enable all these systems to interact, we require a common model to which each node can provide a translator. Similar to the distributed database approach, each node provides a nodal knowledge participation schema in the common model, describing the knowledge that it is making available to the global system. The common model should be high-level, and should be capable of representing any construct of each of the underlying nodal models. In particular we anticipate nodes using production rules, logic programming and frame languages.
Production rules take the form

IF ⟨antecedent⟩ THEN ⟨consequent⟩

Logic programming languages such as Prolog are based on horn clauses. Prolog can represent production rules provided the consequent is a simple one without logical connectives. Frame languages, on the other hand, have little in common with the other approaches. They are based on an object hierarchy with property inheritance. In a sense they are more general, yet often they allow only pre-defined operations.

In[7], DEAL has been proposed as the common language, and that is the approach which we shall follow here. The advantages of a relational framework for an object-oriented language to represent knowledge are that relations can be readily decomposed and recomposed, and that the relational operations are closed, permitting both high-level structures and operations. These advantages are particularly apparent in a distributed environment.

We shall assume all end-user queries to be expressed in DEAL, though in principle other interfaces could be provided. At the user's node, a transaction processor will compile the query into the form of a parse tree. At this point an optimiser will apply optimising transformations to the tree according to a set of heuristics, and will decompose it into subtrees, each representing a subquery. Each subquery must be allocated to an execution node, and sent via a standard communication protocol. Each node must have a means of executing DEAL queries. Since full translation of DEAL into the underlying model will not generally be feasible, some nodes may include a supplementary knowledge module to carry out processing which cannot be handled locally.

3. DEAL LANGUAGE

In this paper we shall not consider the whole of DEAL, but shall concentrate instead on the constructs that must be provided in order to provide interfaces to both Prolog systems and frame-based systems.

DEAL supports complex objects by means of unnormalised relations. It allows a relation to possess attributes which are themselves relations. Alternatively an attribute may be composite, with several subordinate attributes grouped together to form a higher-level attribute. An example is

DEPT (DNO, DNAME, EMP (ENO, ENAME, DOJ (DAY, MONTH, YEAR), SAL))

Here the relation DEPT is defined to have three attributes – DNO, DNAME and EMP. EMP is itself a subrelation, having four attributes ENO, ENAME, DOJ and SAL. Of these, DOJ (representing date of joining) is in turn composite, having components DAY, MONTH and YEAR.

A DEAL expression in general has three components, though sometimes one or more may be omitted. The components are:

- base expression
- attribute specification
- selection predicate

The attribute specification corresponds to the relational algebra 'project' operation, specifying the attributes to appear in the result. These attributes may be high-level 'molecular' attributes (such as EMP and DOJ in the above example) or they may be atomic.

The selection predicate corresponds to the relational algebra 'select' operation, though it also allows tuple predicates.

The base expression may involve various operations on one or more relations, and evaluates to a relation. The allowed operations include:

- cartesian product
- (natural) join
- outer join
- union
- difference
- extend (to add an extra attribute to a relation, the values of the new attribute being defined in terms of existing attributes)
- replace (similar to extend, except that the new attribute replaces an existing one)
- unnormalise (where a number of existing attributes are grouped together under a new higher-level attribute (such as DOJ in the above example)).

Each tuple of a relation is regarded as representing a fact. Deduction may be accomplished by generating new tuples from existing ones, and adding them to a relation by means of a union operation. As an example, consider a relation LIKE (PER, OBJ) where each tuple represents the fact that the named person (PER) likes the named object (OBJ, which may itself be another person). A possible rule involving this relation would be

"John likes anyone who likes wine"

In Prolog such a rule could be expressed as

```
LIKE (john, X) :— LIKE (X, wine)
```

In DEAL, the same rule can be written

```
LIKE := LIKE ++ ANON [PER := "John", OBJ := X]
  WHERE LIKE [PER = X, OBJ = "Wine"]
```

Here '++' indicates the union operation. The expression to the right of the union generates the tuples to be added to LIKE. The part within the first set of square brackets is the attribute specification, and the

part following WHERE is the selection predicate. The word ANON is used to indicate the absence of any base expression from that part of the query. The full query, however, can be viewed as being a union of two base expressions, yielding another base expression. Thus a base expression can be defined recursively – essentially it is any subexpression which evaluates to a relation.

4. THE SPECIFICATION LANGUAGE

The query language semantics will be specified using VDM[10], which originally evolved from work on the definition of programming languages and their compilers but is now used more generally for the specification of software systems. It has also been used quite widely for database specifications[2,3]. VDM is often described as a model-oriented specification language, in that data types are specified by providing a model. The language provides a basic set of models (set, sequence, map and composite object), and more complex models can be constructed by using the basic set as building blocks. We first outline each of the basic models, and then describe the way in which operations can be specified. The language allows specifications to be sufficiently precise to form the basis for proofs of correctness – in our case the correctness of a transformation applied to a query expression.

The *set* is a familiar concept from mathematics, and the notation used in specifications is largely the same as that used in mathematics. The notation set of X is used in preference to P(X) (ie the power set) when only finite subsets are permitted.

A *composite object* has a number of fields and is denoted by, for example,

Datec:: day: {1,...,366}
year: {1583,...,2599}

which defines Datec to be composed of fields day and year, the type of each field defined as a subrange of integers. An instance of a composite type can be constructed by means of a make-function, eg mk-Datec (20, 1987) will construct an instance of type Datec, having day = 20 and year = 1987. To select a particular field from a composite object, we use the notation field (object). Thus day(mk-Datec (20, 1987)) will yield 20.

A *map* is essentially a finite function from a domain set to a range set. Whereas a function is often defined by a fixed expression, a temperature chart could be modelled as a map from the set of cities to the set of possible temperatures. The declaration would be

Tempchart = map City to Temp

An instance of such a map can be defined by, for example,

m = {'London' → 10, 'Keele' → 8, 'Aberdeen' → 11}

The domain of a map is denoted by

Dom(m) = {'London', 'Keele', 'Aberdeen'}

The application of a map to a specific element of its domain is dentoted by

m('London') = 10

There is a map union operator (∪) which is defined only on two operand maps whose domains have empty intersection. The result of the union is a map which contains all the maplets of each of the operands.

{'London' → 10, 'Aberdeen' → 11} ∪
{'Keele' → 15, 'Glasgow' → 6}
= {'London' → 10, 'Keele' → 15, 'Aberdeen' → 11,
'Glasgow' → 6}

To restrict a map to those elements of its domain which belong to a specified set, we use the notation s ◁ m. Thus

{'London', 'Edinburgh', 'Aberdeen'} ◁ m
= {'London' → 10, 'Aberdeen' → 11}

Similarly, to get just those elements of the domain which *do not* belong to the specified set, we use s ⩤ m.
The *sequence* is not used in our model, and is therefore not discussed here.

In specifying a data type, it is often useful to include in the model an invariant, which imposes a restriction on the set of values allowed in the data type. For example, the composite type Datec, defined above, would require an invariant to state that the value 366 is only permissible for the day field when the value of year corresponds to a leap year. Thus

Inv-Datec (mk-Datec (d, y)) △
is-leapyr (y) v d < = 365

In addition to modelling the values of a data type, we need to be able to specify operations. In VDM, an operation specification in general has four components

(i) OPNAME (arg1 : type1, arg2 : type2,..., argn : type n)r : type r
(ii) ext rd g1 : type 1, wr g2 : type 2, . . . , rd gn : type n
(iii) pre predicate
(iv) post predicate

The first component gives the name of the operation, the names and types of its arguments, and the name and type of its result.

The second component gives the names and types of any external variables to which the operation has access. The specifications used in this paper do not include any external variables, hence this component will be omitted.

The two remaining components define the behaviour of the operation, and are called the pre-condition and post-condition. The specification states that, for any initial state which satisfies the pre-condition, the operation yields a final state which satisfies the post-condition.

5. SEMANTICS OF DEAL

Before defining the constructs of DEAL itself, it is necessary to define the data structures on which it operates. The data structures will be defined as VDM models (or abstract data types) and the language will then be defined as a set of operations on these models.

DEAL operates on a database consisting of unnormalised relations. Thus

db = set of relation

Each relation may contain repeating groups (subrelations) and composite attributes, in addition to the

atomic attributes of Codd's relational model. For example,

(A1) DEPT (DNO, DNAME, EMP (ENO, ENAME, SAL, DOJ (DAY, MONTH, YEAR)))

In this case, for each department tuple, there may be many employees. EMP is a subrelation of DEPT, each tuple of DEPT containing one EMP relation as an attribute. Furthermore, EMP contains a composite attribute DOJ which has three components—DAY, MONTH and YEAR. Figure 1(a) shows a possible extension for the DEPT relation. To model this structure, we first normalise the relation thus

(A2) DEPT (DNO, DNAME, ENO, ENAME, SAL, DAY, MONTH, YEAR)

with in general the same (DNO, DNAME) values being repeated many times for the same values of the remaining attributes (see figure 1(b)). Then we preserve the information on subrelations and composite attributes separately, in an attribute hierarchy, which in VDM terms can be modelled as a map from inferior attributes to superior attributes. The hierarchy defines DOJ as consisting of (DAY, MONTH, YEAR) and EMP as consisting of (ENO, ENAME, SAL, DOJ) (see figure 2).

In VDM notation

relation :: structure : map attname to domain
hierarchy : map (inf)attname to (sup)attname
where state : set of tuple

tuple = map attname to attval

As it stands, this definition allows too broad a class of objects to qualify as (unnormalised) relations. We require an invariant to capture the further essential properties of relations:

inv-relation (mk-relation (str, h, sta)) $\triangleq$

$\forall t \in sta \,.\, dom\ t \subseteq dom\ str \wedge$ (1)

$\forall a \in dom\ t \,.\, t(a) \in str(a) \wedge$ (2)

$rng\ h \cap dom\ str = \{\} \wedge$ (3)

$dom\ h \subseteq rng\ h \cup dom\ str$ (4)

Condition (1) states that the attributes whose values appear in any given tuple must be a subset of the attributes of the relation as a whole. In many cases it will be the entire set, but allowing more generally any subset permits us to include null values. Strictly, the definition should incorporate the entity integrity constraint that only non-key attributes may take null values. For simplicity, however, we shall disregard keys here.

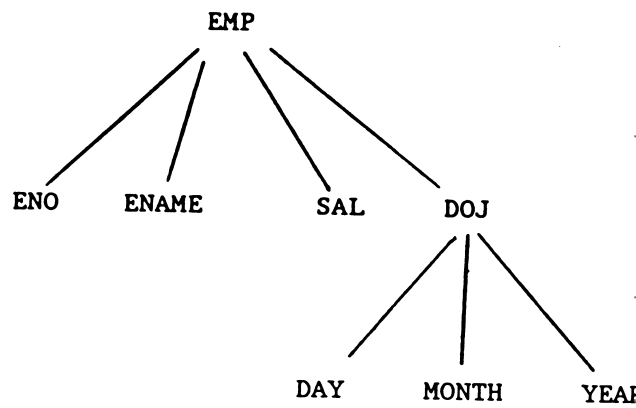

Figure 2. Attribute hierarchy for the DEPT relation (attributes DNO and DNAME are not part of the hierarchy)

DNO	DNAME	EMP	(ENO	ENAME	SAL	DOJ	(DAY	MONTH	YEAR))
1	FINANCE		101	CARTER	10		1	3	86
			106	MAXWELL	15		1	9	87
2	PERSONNEL		108	ROBSON	11		15	2	88
			122	MILLER	8		1	4	89
3	PLANNING		103	HARVEY	20		1	10	78
			104	ATKINSON	16		3	9	87
			115	FERGUSON	25		10	8	83

(a) A possible extension of the DEPT relation

DEPT.

DNO	DNAME	ENO	ENAME	SAL	DAY	MONTH	YEAR
1	FINANCE	101	CARTER	10	1	3	86
1	FINANCE	106	MAXWELL	15	1	9	87
2	PERSONNEL	108	ROBSON	11	15	2	88
2	PERSONNEL	122	MILLER	8	1	4	89
3	PLANNING	103	HARVEY	20	1	10	78
3	PLANNING	104	ATKINSON	16	3	9	87
3	PLANNING	115	FERGUSON	25	10	8	83

(b) A normalised DEPT relation

Figure 1: DEPT relation

Condition (2) states that any attribute value must be a value of the domain over which that attribute has been defined.

Condition (3) states that the range of the hierarchy mapping has an empty intersection with the set of attributes defined in the structure. This is because the structure gives only the attributes in the normalised version of the relation (see (A2) above), whereas the hierarchy maps inferior attributes to superior ones. Thus the range of the hierarchy mapping consists of only the composite attributes and subrelations, i.e. precisely those which are omitted from the structure definition.

Condition (4) states that any inferior attribute must either be also a superior attribute or else one of the attributes in the normalised version of the relation. It is necessary to allow the first of these possibilities because sometimes the hierarchy may extend to more than two levels. Thus in (A1) above, the attribute DOJ is superior to DAY, MONTH and YEAR but inferior to EMP (figure 2).

Before proceeding to define the language, one special function is introduced. This is to give the set of attributes at the lowest level of the hierarchy which correspond to a given set of attributes. This allows for queries to refer to attributes at various levels, and for the system to interpret each query as though the corresponding bottom-level attributes had been referred to. The function is defined by

leaves : set of attname $->$ set of attname
leaves ({}) = {}
and, for any x $\neq$ {},
leaves (x) = (x − rng(hierarchy(rel)))
$\cup$ leaves({y|hierarchy(rel)(y) $\in$ x})
where x contains attributes from the relation rel.

Since rng(hierarchy(rel)) represents all the non-leaf attributes of rel, it can be seen that (x − rng(hierarchy(rel))) is the set of all leaf attributes within x. To this set must be added the leaf attributes which are descendants of attributes in x, which are given by the term on the right of the union. For example, considering the relation (A1) whose attribute hierarchy is shown in figure 2,

leaves({dname,ename,doj}) =
({dname,ename,doj{doj,emp}) $\cup$
leaves({day,month,year})
= {dname,ename,day,month,year}

We now proceed to specify the general form of a DEAL expression, ignoring for the moment the possible forms of the base expression. In the general case, the base expression is simply a relation, which may be derived in a variety of ways from the underlying database. The attribute specification can be viewed as a set of attribute names, each of which is either a bottom-level or a higher-level attribute. The selection predicate could be viewed as a map from tuples to the boolean set. To make the specifications more precise, however, it is useful to state explicitly whether the predicate is dependent on the whole tuple or only on certain attribute values. Thus we define a data type pred (for predicate) by

pred = map (map attname to attval) to B

The predicate takes a projection of a tuple onto certain attributes, and returns a boolean result. It is important to note that it takes the same projection for each tuple of the relation. Thus

inv,pred (p) $\triangleq \forall$ m1, m2 $\in$ dom
p . dom m1 = dom m2

Here dom m1 is the set of attributes which appear in the predicate.

We are now in a position to define the general form of a DEAL query:

EvalQuery (b:relation, as:set of attname, sp:pred)
res : relation
pre as $\subseteq$ dom structure (b) $\cup$ rng hierarchy (b) $\wedge$
$\forall$m $\in$ dom sp. $\forall$t $\in$ state (b) . dom m$\lhd$t
$\in$ dom sp
post structure (res) = leaves (as) $\lhd$ structure (b) $\wedge$
hierarchy (res) = {a $\in$ dom hierarchy (b)
. leaves ({a}) $\cap$ leaves (as) $\neq$ {}} $\lhd$ hierarchy (b) $\wedge$
state (res) = {dom structure (res) $\lhd$ t . t $\in$ state (b)
$\wedge$ $\forall$m $\in$ dom sp . sp (dom m $\lhd$ t)}

The pre-condition states that

(i) each attribute in the attribute specification is either a bottom-level attribute (in dom structure (b)) or a higher-level attribute (in rng hierarchy (b)).
(ii) the selection predicate is defined on all the typles of the relation

The post-condition states that

(i) the structure of the result relation can be formed by restricting the structure of the base expression to those bottom-level attributes which are either in the attribute specification or inferior to some attribute in the attribute specification.
(ii) the attribute hierarchy of the result relation can be formed by restricting the hierarchy of the base expression to those parts where the inferior attribute is either in, or overlaps with, the attribute specification. Such overlaps might occur through the inferior attribute itself being a higher-level attribute, only part of which is retained in the result relation.
(iii) the result relation can be formed by taking those tuples of the base expression which satisfy the selection predicate, and projecting them onto the required attributes as specified in (i).

An alternative form of DEAL expression involves tuple predicates in the WHERE clause, and generates tuples in terms of these predicates rather than selecting them from a base expression. The attribute specification can then be modelled by a type

attspec = map (map attname to attval) to (map attname to attval)

The semantics of this form of query can be specified by

Evalquery2 (as : attspec,sp : pred) res : relation
pre $\forall$d $\in$ dom as . $\forall$m $\in$ dom sp . dom d $\subseteq$ dom m
post structure(res) = {a $->$ d|$\forall$t $\in$ rng as . a
$\in$ dom t $\wedge$ $\forall$s $\in$ rng as . a $\in$ dom s $=>$ s(a) $\in$ d} $\wedge$
hierarchy(res) = {} $\wedge$
state(res) = rng as }

Since this second form of expression does not involve

any base expression, there is little scope for applying optimising transformations to it. Therefore we shall concentrate here on the first form of expression, and shall refer to that as the general form of DEAL query.

Having specified the general form of a DEAL query, it remains to define the possible forms of the base expression. The base expression consists of a sequence of operations of an extended relational algebra, the operations of course being slightly modified to cater for unnormalised relations. The correct sequence of operations can be determined by syntax analysis. If specifying the semantics we need only be concerned with the definition of each individual operation.

The *extend* operation takes a single relation and extends it with a further attribute. The value to be assigned to the new attribute is a function of the other attribute values in the tuple—though sometimes it may be a constant for all tuples. To allow for the fact that the value is not necessarily dependent on every attribute value in the tuple, we define the generating function in similar fashion to the selection predicate:

```
gener = map (map attname to attval) to attval
where
inv-gener (g) △ ∀ m1, m2 ∈ dom g . dom m1
        = dom m2
```

The invariant states that the attributes needed for generating the new value are the same for each tuple.

The extend operation is defined as follows:

```
Extend (rel : relation, att : attname, value : gener)
        res : relation
pre att ∉ dom structure (rel) ∪ rng hierarchy (rel) ∧
  ∀m ∈ dom value . ∀t ∈ state (rel) . dom m ◁ t
∈ dom value
post ∃d . rng value ⊆ d ∧
  structure (res) = structure(rel) ∪ {att − > d} ∧
  hierarchy (res) = hierarchy (rel) ∧
  state (res) = {t ∪ {att → value (dom m ◁ t)}|
                t ∈ state (rel) ∧ m ∈ dom value} ∧
  leaves ({att}) ∩ dom hierarchy (res) = {}
```

The pre-condition states that

(i) the added attribute must have a name that is distinct from those of the existing attributes (bottom-level or higher-level) of the relation
(ii) the generating function which defines the new attribute is defined on all tuples of the relation.

The post-condition states that

(i) the structure of the result can be formed by adding to the original structure a new attribute whose name is specified in the query and whose values belong to a (unspecified) domain d which can be deduced from the nature of the generating function.
(ii) the hierarchy is unchanged by the operation—implying that the new attribute must not form part of the hierarchy, but may possess neither superior nor inferior attributes.
(iii) the result relation may be formed by extending each tuple with an additional attribute, whose value is taken from the generating function.
(iv) as implied by (ii), the new attribute remains separate from the attribute hierarchy.

The *replace* operation is similar to extend except that here the new attribute replaces one which was previously present. It may be viewed as extend followed by a projection to remove the replaced attribute. Formally it is defined thus:

```
Replace(rel : relation, oldatt : attname,
  newatt : attname, newval : gener) res : relation
pre newatt ∉ dom structure (rel) ∪
  rng hierarchy (rel) ∧
  oldatt ∈ dom structure (rel) ∪
    rng hierarchy (rel) ∧
  ∀m ∈ dom newval . ∀t ∈
    state (rel) . dom m ◁ t ∈
    dom newval
post ∃d . rng value ⊆ d ∧
  structure (res) = leaves ({oldatt}) ⩤
    (structure (rel) ∪ {newatt − > d}) ∧
  hierarchy (res) = {a ∈
    dom hierarchy (rel) . leaves ({a}) ∩
                leaves (dom structure (rel) ∪
                {newatt} − leaves      ({oldatt})) ≠
                {}} ◁
                hierarchy (rel) ∧
    state (res) = {leaves ({oldatt}) ⩤ (t ∪
      {newatt − > newval (dom m ◁ t)})
  |t ∈ state (rel) ∧ m ∈ dom newval} ∧
  leaves ({newatt}) ∩ dom hierarchy (res) = {}
```

The pre-condition states that

(i) the new attribute has a name that is distinct from those of the existing attributes (bottom-level or higher-level) of the relation.
(ii) the attribute to be replaced must initially be present either as a bottom-level or as a higher-level attribute
(iii) the generating function, used to derive the values of the new attribute, is defined on each tuple of the relation.

The post-condition states that

(i) the structure of the result is the original structure with the definition of the new attribute added and with appropriate attributes removed. Those removed are either just the replaced attribute (in the case where it is a bottom-level attribute) or all bottom-level inferiors of the replaced attribute (in the case where it is a higher-level attribute).
(ii) the attribute hierarchy of the result is taken from the original hierarchy, eliminating those parts involving the replaced attribute.
(iii) the result relation is formed by extending each tuple of the original relation with a new attribute, whose value is obtained from the generating function, and removing the attribute or attributes which are being replaced.
(iv) as with the extend operation, the new attribute remains separate from the attribute hierarchy.

The remaining operators are mostly more straightforward. For the *join*, we impose the restriction that relations may be joined only on a single attribute, which must be at the bottom level of the attribute hierarchy (or separate from the hierarchy altogether).

Join (rel1 : relation, rel2 : relation, att1 : attname,
att2 : attname)res : relation
pre att1 ∈ dom structure (rel1) ∧ att2 ∈
dom structure (rel2) ∧
structure (rel1) (att1) = structure (rel2) (att2)
post structure (res) = structure (rel1) ∪
att2 ⩤ structure (rel2) ∧
hierarchy (res) = hierarchy (rel1) ∪
att2 ⩤ hierarchy (rel2) ∧
state (res) = {t1 ∪ att2 ⩤ t2 | t1 ∈
state (rel1) ∧ t2 ∈ state (rel2) ∧
t1 (att1) = t2 (att2)}

Cartesian product requires no pre-condition, since it is defined on any pair of relations.

Cartprod (rel1 : relation, rel2 : relation) res : relation
post structure (res) = structure (rel1) ∪
structure (rel2) ∧
hierarchy (res) = hierarchy (rel1) ∪
hierarchy (rel2) ∧
state (res) = {t1 ∪ t2 | t1 ∈ state (rel1) ∧
t2 ∈ state (rel2)}

The post-condition states that

(i) the structure of the result is merely the union of the structures of the operands.
(ii) the hierarchy of the result is the union of the hierarchies of the operands.
(iii) the result relation is formed by concatenating tuples, one from each operand relation, for each possible pair of tuples.

The definition of *outer-join* is similar to that of join, except that there is no information loss with an outer-join. A tuple of one relation which matches no tuple of the other relation is nevertheless included in the result, being filled out with null values for the attributes which come from the other relation.

Outerjoin (rel1 : relation, rel2 : relation,
att1 : attname, att2 : attname) res : relation
pre att1 ∈ dom structure (rel1) ∧ att2 ∈
dom structure (rel2) ∧
structure (rel1) (att1) = structure (rel2) (att2)
post structure (res) = structure (rel1) ∪
att2 ⩤ structure (rel2) ∧
hierarchy (res) = hierarchy (rel1) ∪
att2 ⩤ hierarchy (rel2) ∧
state (res) = {t1 ∪ att2 ⩤ t2 | t1 ∈
state (rel1) ∧ t2 ∈ state (rel2) ∧
t1 (att1) = t2 (att2)} ∪
{t1 | t1 ∈ state (rel1) ∧ ∄ t2 ∈ state (rel2).
t1 (att1) = t2 (att2)} ∪
{t2 | t2 ∈ state (rel2) ∧ ∄ t1 ∈ state (rel1).
t1 (att1) = t2 (att2)}

The set operations (*union* and *difference*) are very straightforward:

Union (rel1 : relation, rel2 : relation) res : relation
pre structure (rel1) = structure (rel2) ∧
hierarchy (rel1) = hierarchy (rel2)
post structure (res) = structure (rel1) ∧
hierarchy (res) = hierarchy (rel1) ∧
state (res) = state (rel1) ∪ state (rel2)

The pre-condition is simply the condition for union-compatibility.

Difference (rel1 : relation, rel2 : relation) res : relation
pre structure (rel1) = structure (rel2) ∧
hierarchy (rel1) = hierarchy (rel2)
post structure (res) = structure (rel1) ∧
hierarchy (res) = hierarchy (rel1 ∧
state (res) = state (rel1) − state (rel2)

The *unnormalise* operation has the effect of grouping together a number of existing attributes and introducing a higher-level attribute as their superior. No new values are added to the relation, only the attribute hierarchy being affected.

Unnorm (rel : relation, sup : attname,
inf : set of attname) res : relation
pre sup ∉ dom structure (rel) ∧ sup ∉
rng hierarchy (rel) ∧
inf ⊆ dom structure (rel) ∪ rng hierarchy (rel) ∧
inf ∩ dom hierarchy (rel) = {}
post structure (res) = structure (rel) ∧
hierarchy (res) = hierarchy (rel) ∪
{a → sup | a ∈ inf} ∧
state (res) = state (rel)

The pre-condition states that

(i) the introduced higher-level attribute must have a name that is distinct from that of any existing (bottom-level or higher-level) attribute.
(ii) each of the grouped attributes must already exist, some may be superior attributes while others may be separate from the attribute hierarchy, but none may be already inferior attributes in the hierarchy.

6. LOGICAL OPTIMISATION OF QUERIES

Each query is first compiled into a parse tree. Algebraic languages are particularly suited to this approach, as has been found in distributed databases.[4] The tree represents the operations to be carried out, along with their parameters and an indication of the ordering of operations. The ordering will be determined purely on the grounds of the way in which the query has been expressed by the user. There may be alternative sequences of operations which will produce the desired results more efficiently because they

(*a*) involve less processing, perhaps due to the most expensive operations being carried out on smaller operands
(*b*) involve less knowledge being moved between nodes

In distributed database research, the latter of these has been found to be the more significant factor.[9,1]

The object of logical optimisation is to apply appropriate optimising transformations to the parse tree to yield a logically-equivalent expression that will be more efficient in execution. The system does not have sufficient information available to be able to evaluate execution efficiency precisely. Some heuristics are required to determine whether or not a given transformation is sensible.

First of all the optimiser must determine what is the best decomposition of the current expression into subqueries. Following the approach for distributed databases adopted in,[5] we take the view that usually an expression should be broken only where necessary, with as few subqueries as possible. The fewer the subqueries, the fewer the number of intermediate results to be sent between nodes. Further, a subquery involving a large number of operations will often produce a result substantially smaller than the sum of the sizes of its constituent relations. So this strategy should produce low communication costs. For some types of expression it will also reduce local processing costs by doing processing on locally stored knowledge, rather than on external knowledge (knowledge sent from another node) for which fast access paths are not available.

The approach of[5] can be applied to DEAL expressions to identify the breakpoints of an expression (i.e. the vertices at which it should be split into subexpressions). From a query expression, and the list of breakpoints which describe its decomposition, we seek transformations which can improve the query expression. There are two classes of transformations to be considered:

(1) distribute a unary operation over a binary operation
(2) change the order of two adjacent unary operations

In fact some of the transformations in class (1) involve the unary operator being applied to only one of the operands of the binary operator, as will be discussed in the next section.

In,[5] four rules have been proposed for determining which transformations to apply:

Rule 1

Distribute a unary operation over a binary operation if the binary operation is a breakpoint of the expression and the unary operation tends to reduce the size of its operand.

Rule 2

Distribute a unary operation over a binary operation if the binary operation is a breakpoint of the expression; the unary operation does not significantly increase the size of its operand and is best done on locally-stored knowledge; and the operand *is* locally stored (i.e. no descendant of the binary operation in the expression tree is a breakpoint).

Rule 3

Change the order of two adjacent unary operations if the first operation (i.e. the first to be evaluated) is a breakpoint and the second operation reduces the size of its operand.

Rule 4

Change the order of two adjacent unary operations if the first is an expensive operation and the second reduces the size of its operand.

We shall next consider the application of these rules to DEAL queries.

7. CONDITIONS FOR APPLYING TRANSFORMATIONS

The general form of a DEAL query is

SELECT ⟨attribute specification⟩
FROM ⟨base expression⟩
WHERE ⟨selection predicate⟩

The natural order of execution would be to first evaluate the base expression, then apply the selection predicate, and finally to project according to the attribute specification. In the semantic definition presented in section 5, the last two stages of this process were combined in an operation called Evalquery. Here we shall continue to regard Evalquery as a single operation, though clearly it could be divided into a sequence of smaller operations. Where a transformation involves Evalquery, it will be quite clear when the applicability of the transformation could be extended by using instead only a suboperation in the transformation.

Since Evalquery is an operation which tends to reduce very substantially the size of its operand, one potential source of optimisation lies in carrying out Evalquery earlier, before completion of evaluation of the base expression. There are other potential transformations within the base expression itself. In order to identify all of them, we shall follow the four rules listed in section 6. We can then test the validity of each transformation by using the semantic definitions of section 5 to determine overall pre- and post-conditions for each sequence of operations.

A query can be regarded as a sequence of operations, such a sequence being denoted by listing the component operations, with semi-colons used as separators. Thus OP1; OP2 represents OP1 followed by OP2. Denoting by pre(OP) and post(OP) the pre-condition and post-condition respectively of operation OP, the semantics of a sequence of operations can be found from

$$\text{pre}(OP1; OP2) = \text{pre}(OP1) \wedge \text{pre}(OP2)$$
$$\text{post}(OP1; OP2) = \text{post}(OP2)$$

with the result of OP1 being replaced in both pre(OP2) and post(OP2) by its definition from post(OP1).

Then a transformation from a sequence seq1 to another sequence seq2 of operations is valid iff

(1) $\text{pre}(seq1) \Rightarrow \text{pre}(seq2)$
and
(2) $\text{pre}(seq1) \wedge \text{post}(seq2) \Rightarrow \text{post}(seq1)$

The point here is that, whenever the specification of seq2 is satisfied, it should be guaranteed that the specification of seq1 is also satisfied. Loosely, this requires that seq2 should have a weaker pre-condition and stronger post-condition than seq1. However, the original sequence of operations is only valid under the condition of pre(seq1) – hence the inclusion of pre(seq1) in (2).

Since the number of possible transformations is large, we shall present a proof of validity for just two transformations and shall then state without proof the conditions for validity of the remaining transformations. Each operation in a sequence will be written in the form

⟨operation name⟩ ((⟨argument list⟩)) ⟨result name⟩

Example 1

First we consider the sequence

```
seq1 = Extend(r,att,val)res1;
  Evalquery(res1,as,sp)res
```

Relation r is first extended with new attribute att, producing res1. Selection and projection are then performed on res1 to yield res. The transformation sought in this case is to do the Evalquery before the Extend. This means that the projection must be onto as-{att} because r does not possess an attribute att. Thus

```
seq2 = Evalquery(r,as-{att},sp)R1;
  Extend(R1,att,val)R
```

Now using the formulae for overall pre- and post-conditions, with the definitions of Extend and Evalquery given in section 5,

```
pre(seq1)=
  att ∉ dom structure(r) ∪ rng hierarchy(r) ∧                    (1)
  ∀ m1 ∈ dom val . ∀ t1 ∈ state(r) . dom m1                      (2)
  ◁ t1 ∈ dom val ∧
  as ⊆ dom structure(r) ∪ {att} ∪ rng                            (3)
  hierarchy(r) ∧
  ∀ m2 ∈ dom sp . ∀ t2 ∈ state(r) . ∀ m3 ∈                       (4)
  dom val . dom m2 ◁ (t2 ∪ {att → val(dom
  m3 ◁ t2)}) ∈ dom sp
```

and

```
pre(seq2)=
  as-{att} ⊆ dom structure(r) ∪ rng                              (5)
  hierarchy(r) ∧
  ∀ m1 ∈ dom sp . ∀ t1 ∈ state(r) . dom m1                       (6)
  ◁ t1 ∈ dom sp ∧
  att ∉ leaves(as-{att}) ∪ rng hierarchy(r) ∧                    (7)
  ∀ m3 ∈ dom val . ∀ t2 ∈ state(r) . ∀ m2 ∈                      (8)
  dom sp . sp(dom m2 ◁ t2) ⇒ (dom m3 ∩
  leaves(as-{att})) ◁ t2 ∈ dom val
```

The first requirement for validity of the transformation is

$$\text{pre(seq1)} \Rightarrow \text{pre(seq2)}$$

First we observe that condition (7) follows from condition (1), because leaves is a function which produces a set of atomic attributes. Since dom structure(r) is the complete set of atomic attributes in r, it follows that leaves(as-{att}) is certainly contained in dom structure(r). Further, it is easy to see that (5) can be deduced from (1) and (3). To deduce (6) from (4), however, requires the further condition that att is not in the domain of the map m2, so that dom m2◁(t2∪{att→X}) reduces to dom m2◁t2. Conditions (6) and (4) are both stating that the selection predicate must be defined on each tuple of the relation, but in the case of (4) the original relation has been extended with an extra attribute. The condition derived is that the extra attribute att must be irrelevant to the selection predicate. Moreover, (8) follows from (2) only if dom m3⊆leaves(as), so that the intersection of those expressions is equivalent to dom m3 (it is obvious that att is not in dom m3, since the latter is the set of attributes needed to generate att). Thus we derive two requirements for the validity of the transformation:

(i) att (the extension attribute) is not used in the selection predicate (more formally, ∀ m1 ∈ dom sp . att ∉ dom m1)

(ii) the projection does not remove any attributes that are needed for defining the extension attribute (more formally, ∀m3 ∈ dom val . dom m3 ⊆ leaves(as)).

We still need to consider the post-conditions, to establish whether any further conditions are required.

```
post(seq1)=
  structure(res) = leaves(as) ◁                                  (9)
  (structure(r) ∪ {att → d}) ∧
  hierarchy(res) = {a ∈ dom hierarchy(r) .                       (10)
  leaves({a}) ∩ leaves(as)≠ {}} ◁ hierarchy(r) ∧
  state(res) = {leaves(as) ◁ (t ∪ {att →                         (11)
  val(dom m1 ◁ t)}) . t ∈ state(r) ∧ m1 ∈
  dom val ∧∀ m2 ∈ dom sp . sp(dom m2 ◁ t)}
```

and

```
post(seq2)=
  structure(R) = leaves(as-{att}) ◁                              (12)
  structure(r) ∪ {att → d} ∧
  hierarchy(R = {a ∈ dom hierarchy(r) .                          (13)
  leaves({a}) ∩ leaves(as) ≠ {}} ◁ hierarchy(r)
  ∧
  state(R) = {leaves(as-{att}) ◁ t ∪ {att →                      (14)
  val((dom m1 ∩ leaves(as-{att})) ◁ t)} . t ∈
  state(r) ∧ m1 ∈ dom val ∧ ∀m2 ∈ dom sp
  . sp(dom m2 ◁ t)}
```

To meet the requirement pre(seq1) ∧ post(seq2) ⇒ post(seq1) we must show that conditions (9)–(11) can be deduced from (1)–(4) and (12)–(14). In fact we can immediately deduce (10) from (13). Further, (11) follows from (14) and (1), subject to condition (ii) which was found to be a requirement earlier. But (9) follows from (12) only if att belongs to leaves(as), thereby ensuring that the maplet att→d appears in the structure. The difference between (9) and (12) is the result of interchanging the extension and the projection. The new attribute att is added in each case, but in (9) there is the possibility that it is removed by the subsequent projection. Thus the analysis of post-conditions has revealed the need for a third requirement

(iii) the extension attribute is not removed by the projection (i.e., att ∈ leaves(as)).

Example 2

We now consider the sequence

```
seq1 = Cartprod(r1,r2)res1;
Replace(res1,oldatt,newatt,val)res
```

First res1 is formed as the Cartesian product of r1 and r2. In this relation, we then replace oldatt by newatt, derived according to the function val. Here we seek to carry out the replace operation on one of the relations first, and then to form the Cartesian product. The motivation for this is that it would make the replace operation faster (since it would be performed on a much smaller operand), without greatly affecting the performance of the Cartesian product. Thus

```
seq2 = Replace(r1,oldatt,newatt,val)R1;
Cartprod(R1,r2)Res
```

Again we begin by analysing the pre-conditions:

pre(seq1)=
newatt ∉ dom structure(r1) ∪ dom structure(r2) ∪ rng hierarchy(r1) ∪ rng hierarchy(r2) ∧ (1)
oldatt ∈ dom structure(r1) ∪ dom structure(r2) ∪ rng hierarchy(r1) ∪ rng hierarchy(r2) ∧ (2)
∀ m ∈ dom val . ∀ t1 ∈ state(r1) . ∀ t2 ∈ state(r2) . dom m ◁ (t1 ∪ t2) ∈ dom val (3)

and

pre(seq2)=
newatt ∉ dom structure(r1) ∪ rng hierarchy(r1) ∧ (4)
oldatt ∈ dom structure(r1) ∪ rng hierarchy(r1) ∧ (5)
∀ m ∈ dom val . ∀ t1 ∈ state(r1) . dom m ◁ t1 ∈ dom val (6)

In examining whether pre(seq2) follows from pre(seq1), it is clear that (4) can be deduced from (1), but (5) cannot be deduced. Thus (5) is needed as one of the conditions for applicability of the transformation. Moreover, (6) follows from (3) only if dom m is contained within dom t1, so that dom m ◁ (t1 ∪ t2) reduces to dom m ◁ t1. Conditions (3) and (6) derive from one of the pre-conditions of the replace operation, which stated that the generating function for the new attribute must be defined on each tuple of the relation. The difference between (3) and (6) arises as a result of different relations being considered in the two cases, since in one case the cartesian product has already been formed. But if the generating function uses only attributes from one of the original relations, this difference will not be significant. Thus we derive two requirements for the validity of this transformation:

(i) one relation contains all the attributes needed for defining the extension attribute (more formally, ∀m ∈ dom val . ∀t1 ∈ state(r1) . dom m ⊆ dom t1)
(ii) the same relation contains the replaced attribute (more formally, oldatt ∈ dom structure(r1) ∪ rng hierarchy(r1)).

Now analysing the overall post-conditions, we find that

post(seq1)=
∃d . rng val ⊆ d ∧ structure(res) = leaves({oldatt}) ⩤ (structure(r1) ∪ structure(r2) ∪ {newatt → d}) ∧ (7)
hierarchy(res) = {a ∈ dom hierarchy(r1) ∪ dom hierarchy(r2) . leaves({a}) ∩ leaves(dom structure(r1) ∪ dom structure(r2) ∪ {newatt} − leaves({oldatt})) ≠ {}} ◁ (hierarchy(r1) ∪ hierarchy(r2)) ∧ (8)
state(res) = {leaves({oldatt}) ◁ (t ∪ {newatt → val(dom m ◁ t)})|t ∈ {t1 ∪ t2|t1 ∈ state(r1) ∧ t2 ∈ state(r2)} ∧ m ∈ dom val} (9)

and

post(seq2)=
structure(Res) = (leaves({oldatt}) ⩤ (structure(r1) ∪ {newatt → d})) ∪ structure(r2) ∧ (10)
hierarchy(Res) = ({a ∈ dom hierarchy(r1) . leaves({a}) ∩ leaves(dom structure(r1) ∪ {newatt} - leaves({oldatt})) ≠ {}} ◁ hierarchy(r1)) ∪ hierarchy(r2) ∧ (11)
state(Res) = {leaves({oldatt}) ⩤ (t1 ∪ {newatt → val(dom m ◁ t1)}) ∪ t2|t1 ∈ state(r1) ∧ m ∈ dom val ∧ t2 ∈ state(r2)} (12)

We need to show that conditions (7)–(9) can be deduced from (1)–(3) and (10)–(12). In fact, (7) can be seen to follow from (10), subject to the earlier requirement (ii) being satisfied. Similarly, (8) follows from (11) and (ii). Finally, (9) can be deduced from (12) provided that both (i) and (ii) are satisfied. Thus analysis of the post-conditions reveals that no further conditions are required for the validity of this transformation, beyond those discovered by analysis of the pre-conditions.

Applying the same tests to all the possible transformations we find the following conditions for making transformations:

Under rule 1

(i) Distribute Evalquery over Union and Difference always.

Union(r1,r2)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r1,as,sp)R1; Evalquery(r2,as,sp)R2; Union(R1,R2)Res

and similarly for Difference

(ii) Distribute Evalquery over Cartesian product if

sp = sp1 ∧ sp2 where sp1 is defined on r1 only, and sp2 on r2 only

Cartprod(r1,r2)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r1,as ∩ (dom structure(r1) ∪ rng hierarchy(r1)),sp1)R1; Evalquery(r2,as ∩ (dom structure(r2) ∪ rng hierarchy(r2)),sp2)R2; Cartprod(R1,R2)Res

(iii) Distribute Evalquery over Join if

(*a*) sp = sp1 ∧ sp2 where sp1 is defined on r1 only, and sp2 on r2 only
(*b*) a1 ∈ leaves(as)

Join(r1,r2,a1,a2)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r1,as ∩ (dom structure(r1) ∪ rng hierarchy(r1)),sp1)R1; Evalquery(r2,(as ∪ {a2}) ∩ (dom structure(r2) ∪ rng hierarchy(r2)),sp2)R2; Join(R1,R2,a1,a2)Res

Under rule 2

(i) Distribute Extend or Replace over Union or Difference always

Union(r1,r2)res1; Extend(res1,att,val)res

transformed to

Extend(r1,att,val)R1; Extend(r2,att,val)R2; Union(R1,R2)Res

and similarly for other cases

(ii) Distribute Extend over Cartesian product (extending one relation only) if one relation contains all the necessary attributes for defining the extension attribute

Cartprod(r1,r2)res1; Extend(res1,att,val)res

transformed to

Extend(r1,att,val)R1; Cartprod(R1,r2)Res

(iii) Distribute Replace over Cartesian product (replacing in one relation only) if

(*a*) one relation contains all the necessary attributes for defining the extension attribute, and
(*b*) the same relation contains the replaced attribute

CartProd(r1,r2)res1; Replace(res1,oldatt, newatt,val)res

transformed to

Replace(r1,oldatt,newatt,val)R1; Cartprod(R1,r2)Res

(iv) Distribute Extend over Join (extending one relation only) if one relation contains all the necessary attributes for defining the extension attribute

Join(r1,r2,a1,a2)res1; Extend(res1,att,val)res

transformed to

Extend(r1,att,val)R1; Join(R1,r2,a1,a2)Res

(v) Distribute Replace over Join (replacing in one relation only) if

(*a*) one relation contains all the necessary attributes for defining the new attribute
(*b*) the replaced attribute belongs to the same relation
(*c*) the replace operation does not remove the Join attribute a1

Join(r1,r2,a1,a2)res1; Replace(res1,oldatt, newatt,val)res

transformed to

Replace(r1,oldatt,newatt,val)R1; Join(R1,r2,a1,a2)Res

(v) Distribute Extend over Outerjoin (extending one relation only) if

(*a*) one relation contains all the attributes needed for defining the extension attribute
(*b*) the other relation does not contain all the necessary attributes

Outerjoin(r1,r2,a1,a2)res1; Extend(res1,att,val)res

transformed to

Extend(r1,att,val)R1; Outerjoin(R1,r2,a1,a2)Res

(Note that condition (b) is needed because the extension attribute might be defined only in terms of the join attribute, or indeed it might be just a constant. In such cases, to do the extension before the outer join would mean that the unmatched r2 tuples would no longer get a value for the extension attribute).

(vii) Distribute Replace over outer join (replacing in one relation only) if

(*a*) one relation contains all the necessary attributes for defining the new attribute
(*b*) the replaced attribute belongs to that relation
(*c*) the other relation does not contain all the necessary attributes for defining the new attribute
(*d*) the replace operation does not remove the join attribute

Outerjoin(r1,r2,a1,a2)res1; Replace(res1,oldatt,newatt,val)res

transformed to

Replace(r1,oldatt,newatt,val)R1; Outerjoin(R1,r2,a1,a2)Res

Under Rule 3 (these could also be done under Rule 4)

(i) Do Evalquery before Extend if

(*a*) the extension attribute is not used in the selection predicate
(*b*) the extension attribute is not removed by the projection
(*c*) the projection does not remove any attributes that are needed for defining the extension attribute

Extend(r1,att,val)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r1,leaves(as)-{att},sp)R1; Extend(R1,att,val)R

(ii) Do Evalquery before Replace if

(*a*) the new attribute is not used in the selection predicate
(*b*) the new attribute is not removed by the projection
(*c*) the projection does not remove any attributes that are needed for defining the extension attribute
(*d*) the replaced attribute is not removed by the projection

Replace(r1,oldatt,newatt,val)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r1,leaves(as)-{att},sp)R1; Replace(R1,oldatt,newatt,val)Res

Under Rule 4

Do Evalquery before Unnormalise if

(*a*) leaves(inf) is contained in leaves(as)
(*b*) sup $\notin$ as

Unnorm(r,sup,inf)res1; Evalquery(res1,as,sp)res

transformed to

Evalquery(r,as,sp)R1; Unnorm(R1,sup,inf)Res

The transformations made under rule 1 are similar to those described by Ullman for relational algebra queries,[13] though there are some differences due to our use of unnormalised relations as the underlying data structures. The transformations under rules 2, 3 and 4

are all new, since they involve the extensions to relational algebra which we have adopted specifically for a distributed knowledge base system.

8. CONCLUSION

Good query optimisation techniques are essential if distributed knowledge base systems are to be accepted. An important aspect of query optimisation is logical optimisation, by which we mean the application of semantic-preserving transformations to a query expression. Before making a transformation, two considerations are vital:

(1) does the transformation preserve the semantics of the query?
(2) is the transformation likely to improve the execution efficiency of the query?

To judge (2), we have adapted a set of heuristics developed for distributed database systems. The main result of this paper lies in the approach to (1). By formally defining the semantics of the language, we are able to determine precise conditions under which a given transformation is valid. Sometimes the parameters of an operation need to be changed when a transformation is applied. In cases where such changes are not immediately obvious, the formal semantic definitions have been found to provide useful assistance in highlighting the need for changes.

The transformations developed here have been for a subset of DEAL, sufficient to represent complex objects and to perform first-order deductions. Optimisation of other features of the language, including recursive expressions, remains to be tackled. It is anticipated that the same approach can contribute further in developing an optimiser for the full language.

REFERENCES

1. P. Bernstein *et al.*, 'Query processing in a system for distributed databases (SDD-1)', ACM TODS 6:4, December (1981).
2. D. Bjorner, 'Formalisation of database models', Abstract Software Specifications, Springer-Verlag LNCS 86 (D. Bjorner ed.) (1979).
3. D. Bjorner and H. Lovengreen, 'Formalisation of database systems – and a formal definitions of IMS', Proc. of 8th VLDB, Mexico City (1982).
4. S. M. Deen, R. R. Amin and M. C. Taylor, 'Query decomposition in Preci*', Distributed data sharing systems, publ. North Holland (1984) (Schreiber/Litwin eds.)
5. S. M. Deen, R. R. Amin and M. C. Taylor, 'A strategy for decomposing complex queries in a heterogeneous DDB', Proc. 10th VLDB, Singapore (1984).
6. S. M. Deen, 'A relational language with deductions, functions and recursions', Data and Knowledge Engineering Vol 1 (1985).
7. S. M. Deen, 'An overview of issues in linked knowledge base systems', Proc. of EEC COST 11 conference, Vienna (1988).
8. S. M. Deen and R. V. L. Hinds, 'A non-normal form representation for knowledge base systems', University of Keele Internal Report (1988).
9. A. Hevner and S. B. Yao, 'Query processing in distributed database systems', IEEE trans. on Software Engineering, SE-5:3, May (1979).
10. C. B. Jones, 'Systematic Software Development Using VDM', Prentice-Hall (1986).
11. M. Stonebraker, 'The design of the POSTGRES storage system', Proc. 13th VLDB, Brighton (1987).
12. M. C. Taylor, 'Formal development of query decomposition algorithms for distributed databases', University of Keele Internal Report (1988).
13. J. D. Ullman, 'Principles of database systems', 2nd edition (chapter 8), Computer Science Press (1982).
14. C. Zaniolo, 'The representation and deductive retrieval of complex objects', Proc. 11th VLDB, Stockholm (1985).

Announcements

8–10 MAY 1990

BUDAPEST, HUNGARY

COMNET '90 International Conference

Forward into the Second Quarter-Century in Networking

This Conference is to be a continuation of the traditional and successful COMNET meetings held first in '77 and subsequently in 1981 and 1985 in Budapest. It is the most important meeting of its kind in Central Eastern Europe, with the participation of experts in the field of computer networks from all over the world.

The COMNET series of Conferences are milestones in networking, that is, important results came out for the first time on each occasion (eg message handling-X.400 protocols, the very first reports about ETHERNET, Petri-net models for protocols and others.)

Scope

The state-of-the-art in networking will be thoroughly discussed. Results, bottlenecks and trends in networking (design, operation and use of networks for different purposes). LAN-s, special networks, WAN-s, MAN-s and world-wide nets.

In 1990 networking will complete its 25 years as the idea and very first results of computer networks appeared in 1965. So this provides an appropriate opportunity to evaluate the outstanding achievements of the past quarter-century, to outline the future trends and new network applications to come.

Within this framework major manufacturers will also have the chance to present their views regarding trends in networking developments and applications.

Venue

Hotel TOT, located at a panoramic spot in the hilly green belt of Budapest:
54 Normafa u. Budapest 12.

Information

For more information please contact:

COMNET '90 Conference Secretariat, c/o John v. Neumann Society, P.O.B. 240, H-1368 Budapest. Tel: 36-1-329-349, 36-1-329-390. Telex: 22-5359. Fax: 36-1-354-317.

20–22 JUNE 1990

BARCELONA

Fourth International Conference on Data Communication Systems and Their Performance

Main Objective

The main objective of this conference is to provide a large professional forum for the exchange of recent and original developments on all kind of theories and techniques regarding the data communication systems and their modeled or measured performance. This meeting follows the conferences organized in Paris 1981, in Zürich 1984 and in Rio de Janeiro 1987.

Conference Chairman

Prof. Ramon Puigjaner, Univ. Illes Balears, Vice-President of ATI.

Conference Secretariat

BRP - Barcelona Relaciones Públicas, Edificio Layetana, C/Pau Claris, N.° 138, 7.°, 4.ª, 08009 Barcelona, Spain. Tel: (93) 215 72 14; Fax: (93) 215 72 87.