ON ANALOGICAL QUERY PROCESSING IN LOGIC DATABASE

Takashi YOKOMORI

IIAS-SIS, FUJITSU LTD.
140 Miyamoto, Numazu
Shizuoka, JAPAN 410-03

Abstract

This paper discusses a problem of query processing in logic databases and proposes a method for optimizing queries which is based on the idea of analogical query processing. First, we introduce the notion of a higher-order relation which leads to a kind of design principle for constructing spacially efficient logic databases. Then, by making use of the notion of a higher-order relation, a method is presented in which a query given in a Horn logic formula is transformed into a string called primitive expression over the alphabet comprising primitive constructs of an internal representation language. A primitive expression of a query represents the essential meaning of the query and is used to detect analogy between queries. By using the analogical property between the transformed queries, a method for optimizing queries is demonstrated in three ways of query processing.

1. Introduction

It is one of the primary issues for those who wish to construct database management systems to establish the efficient methods for data representation, data manipulation, query processing, and so forth. Here, the efficiency is the most important requirement, and it has two aspects: time and space. These two measures on efficiency are mutually related and there exists, in principle, a trade-off relation between them. It totally depends on our intension which benefit should be more pursued.

Since a logic programming language Prolog has been proven to be a powerful tool for designing and constructing relational databases, a large amount of work on logic databases has been reported in reference to the area of query optimization, problem solving, and others in logic databases ([C 81], [G 81], [I 81], [KY 82],[YSI 86]). In fact, being supported by unification mechanism and backtracking control, Prolog has many attractive features as a database query language, and some functional extensions of Prolog are proposed for the purpose of building up its descriptive power as a query language.([OT 84],[T 82],[Y 84]). However, it is also true that Prolog must pay much time(inefficiency) in return for its convenience, which is the current main problem in the areas of logic databases.

This paper discusses a problem of processing queries in logic databases and proposes a method for optimizing queries in which the key idea is based on the utilization of "analogy" among queries.

In analogical query processing, we face several difficulties to overcome. One of them is the problem of how to detect analogy between queries. We propose one possible method for detecting analogy in which the concept of higher-order relations plays the central and essential role. The basic idea underlying the concept may be explained as follows.

In a conventional logical formula, the concept of "ancestor" is defined using the concept "parent" as

$ancestor(X,Y) \leftarrow parent(X,Y)$
$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y)$
where $ancestor(X,Y)$(or $parent(X,Y)$) means that X is an ancestor(or parent) of Y.

On the other hand, we notice there is another way of representing the concept "ancestor". It is not so hard to see that a relation "ancestor" can be obtained by applying infinitely many times the transitive law to a relation "parent". In other words, "ancestor" is the transitive closure of "parent", which is formulated by

$ancestor(X,Y) \leftarrow transitive\text{-}closure(parent,X,Y)$
where the definition of "*transitive-closure*" is assumed,

in an informal manner.

We analyse the difference between the two formulations as follows. That is, the latter way of defining "ancestor" is based on abstracting the higher-order predicate of "transitive-closure", while the former is concerned with a static formulation of the concepts, and as shown below, the notion of a higher-order relation leads to a design principle for spacially efficient logic databases.

The next section introduces the concept of higher-order relations, and based on the concepts we sketch a logic database design. Section 3 shows how a given query is transformed into another representation called primitive expression which can be taken as the semantic essence of the query and is used for detecting

Proceedings of the Twelfth International Conference on Very Large Data Bases Kyoto, August, 1986

analogy between queries. In Section 4, using the analogical property between queries a method for optimizing queries is demonstrated in three ways: two of them concern analogical query optimization, and one concerns parallel query evaluation. Finally, concluding remarks and the future research directions are briefly mentioned in Section 5.

It should be noted that as a terminological convention, we often use "*query*" to mean *its relational definition* throughout this paper.

2.Higher-Order Relations and Logic Databases

This section introduces the notions of a higher-order relation and logic database in a rather informal manner.

2.1 Higher-order Relations

There are, in general, two kinds of levels with which the inference mechanism is usually concerned: the object-level and the meta-level. Following the conventional understanding, the object-level concerns relations about the facts of the world considered, while the meta-level deals with the methods of manipulating relations at the object-level.

We start our discussion with defining the object-level as the world comprising all sorts of *individuals*. It may contain names of persons, physical materials, and all others that do not involve any abstract concecpt. We call statements concerning the object-level *the first-order relation*. Further, by *the second-order relation* we mean the statement which refers to the first-order relation. In this manner, one can think of the higher order of relations, and the notion of a *higher-order relation* plays the central role in this paper.

In the literature, meta-level inference has been often discussed in reference to the subjects of controlling search and deriving control information in various fields such as algebraic manipulation, program verification, and meta interpreter([BW 81],[D 80],[SB 82]), and the concept itself is recongnized as important in its own way. In this paper, we are concerned with a kind of meta-level inference as well as higher-order relations in general.

To illustrate these concepts introduced above, we give a simple example.

Example 1.(Family World)

Consider a small database comprising the following facts. The domain of the world is the set of person's names: { barbara, cathy, jim, mary, nancy, robert, tom }. The set of facts is as follows:

"*barbara*" is a person, "cathy" is a person,
"*jim*" is a person, "*mary*" is a person,
"*nancy*" is a person, "*robert*" is a person,
"*tom*" is a person, "*cathy*" is a child of "*robert*",
"*nancy*" is a child of "*barbara*", "*jim*" is a child of "*mary*",
"*jim*" is a child of "*tom*", "*mary*" is a child of "*nancy*",
"*robert*" is a child of "*nancy*", "*jim*" is a man,
"*robert*" is a man, "*tom*" is a man

gives a set of the first-order relations, while some of the second-order relations are

"*parent*" is the symmetric relation of "*child*",
"*ancestor*" is the transitive closure of "*parent*",
"*woman*" is the complement of "*man*",
"*person*" is the union of "*man*" and "*woman*".

Further, the set of the third-order relations may contain:

The complement of "*complement*" is "*identity*",
The complement of "*union*" is the intersection of "*complement*"s (De Morgan's Law).

As easily seen, it is all right to understand that
(1) the first-order relation is a relation concerning the object-level relation, i.e., a relation among individual constants in the universe of discourse,
(2) the second-order relation is a relation concerning the first-order relations, and inductively
(3) the Nth-order relation is a relation concerning (N-1) th-order relations.

We shall show in the next subsection that this classification of relations leads to a very spacially efficient design technique for representing relations, and hence, for logic databases.

Before moving on to the next discussion, some definitions are needed.

Let p be a relation of some order. If no variable is contained in p, then p is called *instance relation*. Otherwise, p is called *relation schema*. In other words, an instance relation literally represents an instance, while a relation schema specifies a set of instance relations. For example, relations given in Example 1 are all instance relations. A statement like "X is a symmetric relation of Y" is a relation schema.

Finally, as a notation, for $n>0$, R_n denotes the set of all n-th order relations, and for convenience by R_0 we denote the set of individual constants in the domain of discourse world.

2.2 Logic Database Design

Now, we outline the logical design of the logic database which we call LDB. Taking a database shown in Example 1 into a part of the LDB, we illustrate the conceptual configuration of LDB.

LDB mainly consists of three components : HRS (Higher-order Relation Schema) module, BIR (Base and Instance Relation) module, and EM (Evaluation Module). HRS module contains all kinds of higher-order relation schemas which are independent of data domains, while BIR module comprises a finite number of its submodules each of which is used for storing fundamental relations, called base relations (see below, for definition), and instnace relations in each domain.

(As for other components such as interface module, no discussion will be given, simply because they are not our intension of discussion here.)

[HRS module]

HRS module contains the definitions of *higher-order relation schemas* whose orders are greater than one. It may contain "universal" concepts independent of individual domains.

[BIR module]

Each submodule is created per each domain of a world. It contains *instance relations* of higher-order relations as well as base relations which are specific to the domain. *Base relations* in a domain give the minimum sufficient set of the first-order relations from which one can derive any other first-order relations in the domain by using both relation schemas of HRS and instance relations contained in the submodule itself. For example, BIR submodule for the faimly world may contain instance relations of "child" as its base relation, and also instance relations of second-order relation "symmetric".

[EM]

Evaluation module evaluates queries which are given in QL (Query Language). It consists of Transformation module and Evaluator. More details on EM and QL will be given later.

Figure 1 sketches the conceptual design of LDB.

Here we present the full description of BIR submodule 1 which corresponds to the data of a family world given in Example 1.

[BIR submodule for Family World]

[Base relations]

person(barbara)	*man(jim)*	*child(nancy,barbara)*
person(cathy)	*man(robert)*	*child(jim,mary)*
person(jim)	*man(tom)*	*child(mary,nancy)*
person(mary)		*child(robert,nancy)*
person(nancy)		*child(jim,tom)*
person(robert)		*child(cathy,robert)*
person(tom)		

[Instances of second-order relations]

symmetric(child,parent)
transitive-closure(parent,ancestor)
complement(man,woman)
union(person,man,woman)

[Instances of third-order relations]

equivalence(complement+complement,identity)
equivaelnce(complement(union),
intersection(complement))

It should be noted that the set of base relations for Family World contains only instance relations of first-order: "person", "man", and "child", and no other relation concerning a family world (like "parent", "ancestor" and so forth) appears in anywhere of LDB. As shown later, first-order relations other than base relations are derived (constructed) from base relations using higher-order relations.

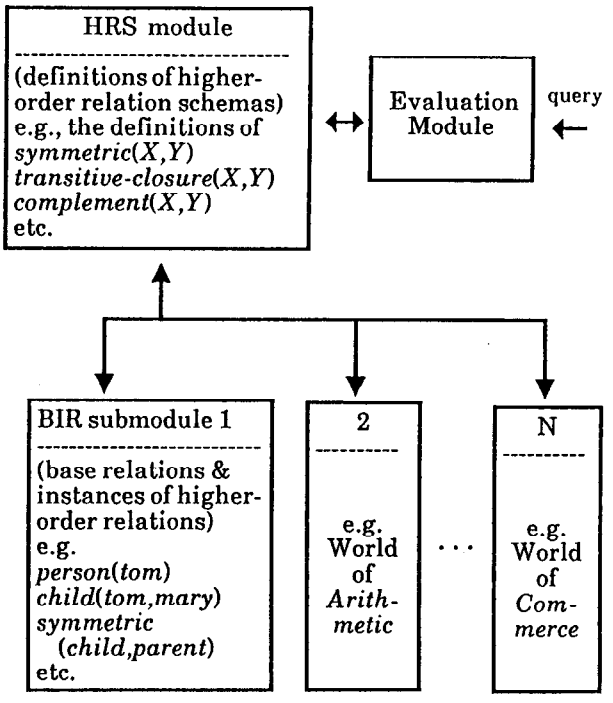


Figure 1. Conceptual Configuration of LDB

The design philosophy behind our configuration may be summarized as follows:

(1) abstracting high-level concepts (higher-order relations),

(2) separating low-level details depending on domains from the universal properties of high-level concepts,

(3) gaining clarity and modularity in handling data and processing queries, leading to a design method for spacially efficient (compact) logic databases.

3. Transforming Queries into Primitives

Using the notions of higher-order relations introduced in the previous section, we shall show in this section how a query given in a Horn logic formulation is transformed into a string of primitive relations with connectives (operations) which consists of base relations and higher-order relations.

3.1 Query Language(QL) and Internal Representation Language(IRL)

Query language QL is assumed to be the *Horn logic*, that is, the subset of the first-order logic on which the programming language Prolog is based. (Note that Prolog is not purely a subset of first-order logic.)

Given a query defined in Horn logic formulation, we transform it into another expression which is

constructed from base relations and instance relations in the domain the query concerns by using relational algebraic operations and set operations. As shown below, relational algebraic operations as well as the set operations are closely related to and easily translated into(or realized by) the first-order logic formulas, and vice versa.

<Relational Algebra Operations>

(1) projection : pr(P,I)
pr(P,I)(X) iff $P(Y,X,Z)$, for some Y,Z, where X is the I-th argument of P.

(2) restriction : res(P,Q,I,J)
res(P,Q,I,J)(X) iff $P(X)$ and $Q(Y,Z)$, where Y,Z are the I-th and J-th arguments in X of P, respectively.

(3) conjunction : AND
(P AND Q)(X) iff $P(X)$ and $Q(X)$.

(4) disjunction : OR
(P OR Q)(X) iff $P(X)$ or $Q(X)$.

(5) complement : com(T,P)
com(T,P)(X) iff $T(X)$ and not($P(X)$), where not($P(X)$) means $P(X)$ does not hold true.

(6) product : $P\times Q$
($P\times Q$)(X,Y) iff $P(X)$ and $Q(Y)$.

Notes
1) In the definition above, an upper-case letter X (or Y, or Z) denotes a sequence of arguments.
2) In the definition of complement "com(T,P)", T is a relation that specifies the total domain in which P is definable.
3) Meta-notion "not(P)" is assumed to be defined as a relation with higher order than that of P exactly by one.

In addition to the operations above, we need some more definitions.(Note that R_0 is the set of all individuals at the object-level in the world of discourse.)

(i) Composition

Let P and Q are binary relation names such that $P(X,Y)$, $Q(Y,Z)$ are in R_n. Then, *the composition of P and Q*, denoted by $P+Q$, is defined as follows:

$$(P+Q)(X,Z) \text{ iff } P(X,Y) \text{ and } Q(Y,Z).$$

The composition of P and Q also results in a relation of R_n.

(ii) Identity

One distinguished binary relation in R_n, which is called *the n-th order identity* and denoted by I_n, is defined by

$$I_n(X,X) \text{ for all } X \text{ in } R_{n-1}.$$

(iii) Logical constants

By T and F we denote "tautology" and "contradiction", respectively.

The set of all relational algebraic operations, all base relation names, an operation +, $I_n(n>0)$, T and F constitute the set of *primitive constructs* of the *internal representation language*(IRL) of LDB.

3.2 Transforming query into primitive expression

Now, recall the BIR submodule for the family world. In what follows, *we always assume the submodule to be in LDB.*

For a given query, say, "Who is a cousin of Jim?", we shall show how the query is trasformed into a string of primitive constructs, and then how it is evaluated for obtaining solutions.

Note that LDB has neither facts nor rules concerning "cousin" relation. However, we observe one of the possible ways to solve this problem in the following manner.

A first-order relation "cousin" is defined in a Prolog-like notation

cousin(X,Y)←*child*(X,Z),*sibling*(Z,W),*parent*(W,Y)(3.1)
sibling(Z,W)←
child(Z,U),*parent*(U,W),not(identity(Z,W))···(3.2)

Using "composition(+)" and "conjunction(AND)", (3.1) and (3.2) can be rewritten as

cousin(X,Y)←[*child*+*sibling*+*parent*](X,Y) ··· (3.1')
sibling(Z,W) ←
[(*child*+*parent*)AND(not(identity))] ···(3.2').

Hence, we have

cousin(X,Y) ←
[*child*+(*child*+*parent*)AND
(not(identity))+*parent*](X,Y).

Note that "parent" is a symmetric relation of "child", and let "parent" be denoted by "symmetric(child)", then finally ,

cousin(X,Y)←
[*child*+(*child*+*symmetric*(*child*))AND
(not(identity))+*symmetric*(*child*)](X,Y)

is obtained.

Abbreviating the right-hand side of this expression, we may as well identify a relation "cousin" with

c+(c+s(c))AND(not(I))+s(c) ··· (3.3)

where c(child) and I(identity) are in R_1,
s(symmetric) is in R_2.

Thus, using base relations and primitive constructs (higher-order relations and operations) a relation "cousin" is transformed into a string consisting of only those elements. The transformed formula is called *primitive expression* for the original relation.

Another example for query transformation is that given a query "Who is an aunt of Cathy?", represented by " ←aunt(X,cathy)" with its definition :

aunt(X,Y)←*sibling*(X,Z),*parent*(Z,Y),*female*(X),

the transformed query is as follows :

$aunt(X,Y) \leftarrow [((c+s(c))AND(not(I))+s(c))AND (com(p,m)\times T)](X,Y)$

where p(person) and m(man) are base relation names in R_1.(Note that com,T,×(poroduct) are primitive constructs of IRL in LDB.)

Getting back to the query "←cousin(X,jim)", one may eventually obtain an answer "X=cathy" by evaluating (3.3).

Generally, it may happen that a relation has its more than one primitive expressions. For example, a relation "cousin" has another formulation as shown below :

$cousin(X,Y) \leftarrow grandchild(X,Z), grandparent(Z,Y), child(X,U), not(identity(U,W)), parent(W,Y)$

that is ,

$cousin(X,Y) \leftarrow [(grandchild+grandparent)AND (child+not(identity)+parent)](X,Y).$

We can eventually have another primitive expression for "cousin":

$(c+c+s(c)+s(c))AND(c+not(I)+s(c)) \cdots (3.4)$

Now, it is almost obvious that the distribution law as well as the associative law hold :

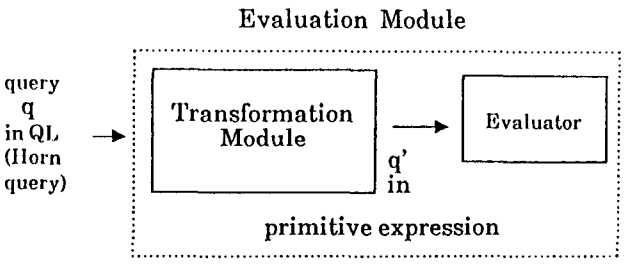
$$\left.\begin{array}{l} X+(Y \text{ AND } Z) = (X+Y) \text{ AND } (X+Z) \\ (X \text{ AND } Y) + Z = (X+Z) \text{ AND } (Y+Z) \\ X+(Y+Z) = (X+Y)+Z. \end{array}\right\} \cdots (L)$$

Then, one can easily see that using these laws the expression (3.3) is reformulated into (3.4), and vice versa.

Let p be a primitive expression for a relation, and suppose it is expressed as : $p=p_1+p_2+\cdots+p_n$ (n>0), where no p_i can be transformed by the laws (L) into the form of q_i+r_i any more. Then, a primitive expression p is called *canonical*.

As shown later, the canonical primitive expression of a query provides a method for treating the meaning of the query, and hence, for the semantic analogies among queries in logic databases. We note that in additon to (L) other conventional laws involving OR, AND, T, F like commutativity, associativity, identity are all available for transforming (optimizing) primitive expressions.

Figure 2 illustrates the query processing line in EM.



where a primitive expression is a string over the alphabet:
$BIR_1 \cup \cdots \cup BIR_m \cup RA \cup \{+, I_n, T, F : n>0\}$;
BIR_i = {r: relation name in BIR submodule i(base or instance whatever it is)}
RA = the set of relational algebra operations (including set operations)

Figure 2. A Query Transformation in EM

4. Analogical Query Processing and Optimization

In the process of evaluating queries, we often observe that there are many cases where exactly the same or quite similar path for retrieving data is repeatedly performed. There may be no doubt that for the purpose of gaining time efficiency the utilization of this "analogical feature" in query processing will be greatly beneficial. In this section we shall discuss this problem of analogical query processing and show a possible way to solve it within the framework of logic databases.

Roughly, there are two primary difficulties in dealing with "analogy" in the general situation. One is how to detect analogies, in other words, how to formalize analogies. Further, the problem of how one can justify the results obtained by means of analogy will be the other. Here, we are concerned with the former, while the latter seems to be much more profound and too hard to solve. In the context of analogical query processing in logic databases, we notice there are two kinds of analogy : syntactic analogy and semantic analogy. Compared with syntactic analogy, semantic analogy has broader meanings and fully covers many kinds of analogies in daily life.

Using several examples, we demonstrate a method for evaluating queries based on analogies among them. Our approach strongly depends on the use of primitive expressions for the queries introduced in the previous section. Taking advantage of the property of canonical primitive expressions, one can get a way of detecting analogy and argue on analogy between the two queries.

4.1 Transforming semantic analogy into syntactic one

Turning back to the discussion given in the previous section, we have observed that the identification of a relation in different formulations can be treated using the primitive expressions for the relation. Actually, a relation "cousin" formulated in two different ways (in syntax) has been identified as a unique representation called the canonical primitive

expression. We notice this is a special case of semantic analogy in daily life in the sense that identity is nothing but the extremity of analogy.

On the other hand, the prefix like "quasi", "semi", or the suffix like "in-law" introduces one of the typical expressions for describing analogial objects.

Example 2

Suppose that a query "Who is a grandparent-in-law of Tom?" is given in a logical formulation:"←in-law(grandparent,X,tom)" A second-order relation schema "in-law(*R*,*X*,*Y*)" is defined as follows :

in-law(*R*,*X*,*Y*)←*R*(*X*,*Z*),*couple*(*Z*,*Y*)
where *R* is a first-order relation(name).

Further, with the help of a logical formula :

couple(*Z*,*Y*)←
parent(*Z*,*W*),*child*(*W*,*Y*),not(identity(*Z*,*Y*))

we eventually have a primitive expression for "in-law" as follows :

in-law(*R*,*X*,*Y*)←[*R*+(s(c)+c)AND(not(I))](*X*,*Y*)···(4.1).

Thus, the meaning of a relation "R" with the suffix "in-law" is represented by its primitive expression (4.1). Note that (4.1) is already canonical if so is R. Since

grandparent(*X*,*Y*)←*parent*(*X*,*Z*),*parent*(*Z*,*Y*)···(4.2)

it follows that "grandparent-in-law" has its canonical primitive expression:

$$s(c)+s(c)+\underline{(s(c)+c)AND(not(I))} \quad \cdots (4.3)$$

and the sub-expression underlined just corresponds to the "in-law" semantics.

Now, from the view point of analogical query processing, we take the following definition of analogical queries :

Let *r* and *r'* be two relations in LDB and let *p* and *p'* be their canonical primitive expressions, respectively. Then, *r and r' have an analogy* if there exists a common sub-expression of *p* and *p'*. Further, a common sub-expression *E* is called *maximal analogy* between *r* and *r'* if the number of occurrences of + contained in *E* is the greatest.(Note that a maximal analogy is not necessarily unique.) In the case above, for example, the maximal analogy between "grandparent" and "grandparent-in-law" is clearly a sub-expression "s(c) +s(c)" whose translation is "grandparent" itself. In the general case, it is not so easy to detect the maximal analogy among queries without the help of primitive expression or of this kind of device. This is understood when one compares (4.2) with the original formula of another query:

grandparent-in-law(*X*,*Y*)←
grandparent(*X*,*Z*), *couple*(*Z*,*Y*) ···(4.4)

On the right-hand sides of these two definitions there is no common syntactic element as they are.

The key idea underlying the algorithm for analogical query processing is now straghtforward:
For a given query,
(1) transform it into the canonical primitive expression,
(2) detect a maximal analogy between the query and relation schemas which have already been processed and added to HRS, provided that LDB keeps both the evaluation results(retrieving paths) and their canonical primitive expressions for whatever relation schemas it has processed.
(3) gain an efficiency by making use of the common retrieving path for evaluating analogical queries.
Thus, in this case LDB is supposed to be a kind of incremental database models. One smiple example for analogical query processing in LDB is as follows:

Given a query "←grandchild-in-law(tom,*X*)" with its logical definition :

grandchild-in-law(*X*,*Y*)←*couple*(*X*,*Z*),*grandchild*(*Z*,*Y*),

the trasformation process results in the expression :

[(s(c)+c)AND(not(I))+c+c](*X*,*Y*).

Using the (meta) relations "s(x+y)=s(y)+s(x)" and "s(s(x))=x", which are supposed to be in HRS, we have

s[s(c)+s(c)+(s(c)+c)AND(not(I))](*X*,*Y*) ··· (4.5)
(Note that s(not(I))=not(I).)

Hence, to answer the query, all we have to do is to evaluate s[s(c)+s(c)+(s(c)+c)AND(not(I))](tom,*X*), that is,

[s(c)+s(c)+(<u>s(c)+c)AND(not(I))</u>](*X*,tom) ··· (4.5').

Provided LDB has ever processed the relation schema "in-law", i.e.,the relation schema (s(*X*)+*X*)AND(not(I)) has been stored in HRS and that the underlined part of (4.5') has already been evaluated for (z,tom) with certain partial answer z, the rest of the task is to compute [s(c)+s(c)](*X*,z) for an answer "*X*=barbara". If (4.4) or equivalently (4.3) has ever been evaluated, then, as a matter of course, nothing remains to be done.

4.2 Making use of analogical mapping

Another type of analogical query processing we would like to discuss here concerns analogies between the two different domains.

Example 3

Suppose that LDB has already processed a large amount of queries concerning the family world to some extent, and that one wish to answer a query on the "Block World" such that "What blocks are the feet blocks of an arch?". All we know about the block world is as follows:

[Base relations]

block(c)	*block*(k)	*on*(n,r)	*on*(r,c)
block(e)	*block*(m)	*on*(m,j)	*on*(n,m)

block(j) *block*(n) *on*(e,k)
block(r)
where "*on*(*x*,*y*)" means that "*x is on y*".
Figure 3 shows the block world above.

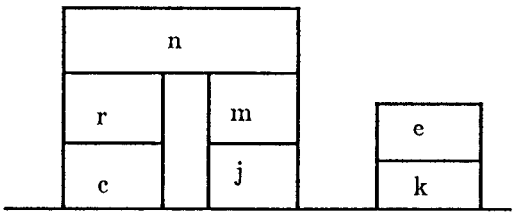


Figure 3. A Block World

Suppose, in addition, that paris (c,cathy), (j,jim), (m,mary),(n,nancy),(r,robert) satisfy the one-to-one mapping T between the block world and the family world such that

on(*x*,*y*) iff *parent*(T(*x*),T(*y*)) ··· (*).

Now, the query is formulated as follows:

feet-of-arch(*X*,*Y*)←*on*(*Z*,*X*),*on*(*U*,*Z*),*on*(*U*,*W*)
not(identity(*Z*,*W*)),*on*(*W*,*Y*) ··· (4.6)

that is,

feet-of-arch(X,Y)←
symmetric(*on*(*X*,*Z*)),*symmetric*(*on*(*Z*,*U*)),
on(*U*,*W*),not(identity(*Z*,*W*)),*on*(*W*,*Y*) ··· (4.7).

Paying attention to the facts that under T "on = parent = symmetric(child)" and "symmetric(on)=child", we end up the transformation of (4.7) with :

feet-of-arch(*X*,*Y*)←
[c+(c+s(c))AND(not(I))+s(c)](*X*,*Y*) ··· (4.8).

This leads to the conclusion that to answer the query of (4.6) we have only to transform answers for the query of "cousin" relation (re-expressed as (3.3)) under the inverse mapping of T.

Thus, we can extend the scope of analogical query processing to the case when there exists a one-to-one mapping from the base world to the world in question.

One may consider the further extensions of the above argument by weakening the restriction on one-to-one mapping T in several ways. Here we briefly mention one of them.

A one-to-one mapping T maps one base relation name to another base relation name in different world with the condition of preserving the circumstance of the original world. (See the definition of T marked(*)) As far as this condition is preserved, one can extend T so that it may map one base relation name in the original world to a primitive expression rather than one base relation name in the other world.

4.3 A Paralell Evaluation Feature

For the purpose of query optimization, we have discussed the method for processing analogical queries based on the primitive expressions. We notice that a careful observation on primitive expressions brings us another interesting aspect of the expressions which suggests a possible way of parallel query evaluation.

Recall the two primitive expressions (3.3) and (3.4) for an identical relation "cousin" in Section 3:

c+(c+s(c))AND(not(I))+s(c) ··· (3.3)
(c+c+s(c)+s(c))AND(c+not(I)+s(c)) ··· (3.4).

Since, unlike the connective "+", the connective"AND" represents a logical "*and*" which forces its both sides to have an identical arguments (that is, (*P* AND *Q*)(*X*) iff *P*(*X*) and *Q*(*X*)), it is possible to evaluate both sides of "AND" in parallel. In such a sense (3.4) is preferable to (3.3) if some kind of parallel processing environment is assumed.

The idea is as follows: Using the transformation laws (L) and other necessary rules available mentioned in Section 3, for a given query we find a primitive expression for it which can contain as many as "AND" connectives as possible. In case there are two or more expressions which contain the same number of "AND" s, we may take the one which gives the largest total sum in length of all the sub-expressions at either side of "AND" connectives.

Example 4

Consider the query "←cousin(*X*,jim)" with its defining formulas (3.1) and (3.2). Then, the transformation line proceeds as follows:

(a) "←*cousin*(*X*,jim)" with (3.1) and (3.2)
→(b) a primitive expression (3.3)
→(c) a primitive expression (3.4)
→(d_1) evaluate [c+c+s(c)+s(c)](*X*,jim)
(in parallel)
→(d_2) evaluate [c+not(I)+s(c)](*X*,jim).

This is interesting in the sense that the transformation procedure from (a) through (d)s described above shows one possible method for extracting the possibility of parallel evaluation from a given query.

Finally, we would like to call one's attention to the fact that whenever a query has no answer, a primitive expression of the form (3.4) has again a great advantage over that of the form (3.3), because for our purpose it suffices to see whether either (d_1) or (d_2) fails.

5. Concluding Remarks

We have discussed a problem of query processing in logic databases in the context of analogical query optimization. By abstracting concepts in some way, the notion of a higher-order relation has been introduced to classify relations and to provide a method for query transformation. Given a query, the method transforms into its primitive expression which reveals the essential meaning of the query. The detection of analogy between queries has been performed through their primitive expressions. Our approach is , we believe, promising in that in contrast to the other traditional methods like a semantic nets, the primitive expression method is much easier and simpler to handle. Similar discussion on both classifying relations and reformulating a Horn query by relational operations can be found in [R 78] and [YSI 86], respectively, in different contexts.

It should be noted that the HRS module in our conceptual design of LDB can be compiled, so that one can improve the process time of query evaluation.

Although we limited to only the use of binary relations throughout demonstrating examples, the primary reason for it is the simplicity and we just intended to show the methodological idea behind it. The principle of the method, we claim, works in the general case of handling relations of any arity.

For the future research direction, we would like to point out the topic of query processing in the logic database with incomplete data. In the framework presented in this paper one may argue for the subject of the problem solving by analogical reasoning([H 85]). In the problem of analogical reasoning, one must deal with an incomplete database in which analogical reasoning mechanism is strongly expected to make up for the incompleteness in the process of query processing. Further, the problem of justifying such an analogical query processing is another important topic, and a recent work in [HA 86] deals with the problem in the framework of logic programs and gives an anwer for it.

ACKNOWLEDGEMENTS

The author is very grateful to Dr.T.Kitagawa, the president of IIAS-SIS, for his ceaseless encouragement and stimulus. He is also indebted to Dr.H.Enomoto, the director of IIAS-SIS, for many fruitful discussion and suggestion.

REFERENCES

[BW 81] Bundy,A. and Welham,B., Using Meta-level Inference for Selective Applications of Muliple Rewrite Rule Sets in Algebraic Manipulation, *Artificial Intelligence* 16, 189-212 (1981).

[C 81] Chang,C.L., On evaluation of queries containing derived relations in a relational data base, in *"Advances in Database Theory"* (H.Gallaire, J.Minker and J.-M.Nicolas, Eds.), Plenum, 235-260 (1981).

[D 80] Davis,R., Meta-rules : reasoning about control, *Artificial Intelligence* 15, 179-222 (1980).

[G 81] Gallaire,H., "Impact of logic on database", *Proc. of 7th Intern. Conf. on VLDB*, 248-259, 1981.

[H 85] Haraguchi,M., Analogical reasoning based on the theory of analogy, *Research Report of RIFIS*, Kyushu Univ., No.105, 1985.

[HA 86] Haraguchi,M. and Arikawa,S., A Foundation of Reasoning by Analogy: Analogical Union of Logic Programs, *Proc. of Logic Programming Conf.'86*, ICOT, June, 1986.

[I 81] Imielinski,T. Algebraic Query Processing in Logical Databases, in *"Advances in Database Theory"* (H.Gallaire, J.Minker and J.-M.Nicolas, Eds.), Plenum, 285-318 (1981).

[KY 82] Kunifuji,S. and Yokota,H., Prolog and relational databases for Fifth Generation Computer Systems, *Proc. of the Workshop on Logical Bases for Data Bases* (Toulouse, France), ONERA-CERT, Toulouse, France, 1982.

[OT 84] Oda,M. and Tanaka,Y., Introducing vocabulary building mechanism into Prolog, *Proc. of the 29th National Conf. of the Inf. Process. Soc. of Japan*, 1205-1206, 1984 (in Japanese).

[R78] Reiter,R., Deductive Question-Answering on Relational Data Bases, in *"Logic and Data Bases"* (H.Gallaire and J.Minker, Eds.), Plenum, 149-177 (1978).

[T 82] Tanaka,Y., Vocabulary building for database queries, *Proc. of the RIMS Symposia on Software Science and Engineering* (Kyoto,1982), *Lecture Notes in Computer Science* 147, Springer, 215-232, 1982.

[SB 82] Sterling,L. and Bundy,A., Meta-level inference and program verification, *Proc. of the 6th Conf. on Automated Deduction* (D.W.Loveland, Ed.), *Lecture Notes in Computer Science* 138, Springer, 144-150, 1982.

[YSI 86] Yokota,H. Sakai,K. and Itoh,H., Deductive Database System based on Unit Resolution, *Proc.of Intern.Conf. on Data Engineering*, Feb., 228-235,1986.

[Y 84] Yokomori,T., A Note on the Set Abstraction in Logic Programming Language, *Proc. of the Intern. Conf. on FGCS'84*, Tokyo, 333-340, 1984.

[Y 85] Yokomori,T., A Logic Program Schema and Its Applications, *Proc. of the 9th IJCAI*, UCLA, Los Angeles, 723-725, 1985.