ICOT Technical Report: TR-026

TR-026

An Enhanced Inference Mechanism for Generating Relational Algebra Queries (Extended Abstract)

by
Haruo Yokota, Susumu Kunifuji,
Takeo Kakuta, Nobuyoshi Miyazaki,
Shigeki Shibayama and Kunio Murakami

October, 1983

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456 3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

An Enhanced Inference Mechanism for Generating Relational Algebra Queries

Haruo Yokota, Susumu Kunifuji, Takeo Kakuta,
Nobuyoshi Miyazaki, Shigeki Shibayama, Kunio Murakami

Institute for New Generation Computer Technology (ICOT)
Mita Kokusai Building, 21 F
1-4-28 Mita, Minato-ku, Tokyo 108 Japan

ABSTRACT

A system for interfacing Prolog programs with relational algebra is presented. The system produces relational algebra queries using a deferred evaluation approach. Least fixed point (LFP) queries are automatically managed. An optimization method for removing redundant relations is also presented.

1. Introduction

Japan's Fifth Generation Computer System (FGCS) project is aimed at constructing a useful knowledge information processing system [6]. As the first stage of building the system, we are developing a working model of a relational database machine (RDBM) called Delta [9,12], which will accept relational algebra-based queries, and working models of personal sequential inference (PSI) machines [11] used for invoking programs written in a Prolog-based logic programming language called the Fifth Generation Kernel Language [6]. Delta will be a hardware backend machine connected to a number of host machines, mainly PSI Machines. Possible physical connections between the PSI machines will be by a local area network or a shared common memory on a bus. See Figure 1.

Under this environment, to form a software development support system using large-scale databases, we must develop a logical interface between Prolog and relational algebra *. We will collect experimental data on the connections for use in constructing a knowledge base mechanism during the next stage of the project.

In this paper, we propose an efficient interface between Prolog and relational algebra. By adding simple specifications to the program, users are able to handle least fixed point (LFP) operations [1,10,8]. Well-known LFP examples are the ancestor finding problem and airline connecting flights problem, neither of these can be handled by relational algebra nor by relational calculus. Our interface is capable of retrieving large-scale databases. An inference mechanism capable of handling large-scale databases is the first step towards building a knowledge base mechanism.

The interface program for the PSI Machines

---

* Ordinary Prolog queries can be seen as high-level database queries, which derive answers one by one from a small database in main memory. Using the system we describe, and with a single Prolog query, a user automatically receives the answer to his query, when the database is kept in both main memory and external storage.

generates optimized queries and sends them to Delta for a Prolog query. Using this interface, host-machine users can write Prolog programs in a natural way, utilizing cut-operator, not-predicate, evaluable predicates, and ordinary predicates. They must explicitly specify which Prolog predicates are stored in Delta. Otherwise, a user can make a slight modification to the system to allow it to set the specifications for those location.

2. A Prolog and Relational Algebra Interface.

A number of papers have proposed several approaches to an interface between logic programming languages and relational databases [3,5,8,13]. Their approaches call for dividing Prolog programs (which are treated as a collection of Horn clauses) into two parts: an extensional database (EDB) which collects the majority of the Prolog facts (unit Horn clauses containing no variables), and an intensional database (IDB) which collects all Prolog rules (Horn clauses except facts) and a few Prolog facts for temporary use. According to the idea proposed by Chakravarthy [3], we developed an enhanced inference mechanism which deduces Prolog goals using only the intensional database, under the assumption that the predicates located in the extensional database normally exist, and simultaneously generates query plans for the extensional database. Tanaka [13] proposed a similar partial evaluation approach which also calls for dividing a Prolog program into two parts. To deduce a query using the intensional database and extensional database, our system defers the evaluation of the predicates in the extensional database; Tanaka uses a partial evaluation method. We call this a deferred evaluation approach.

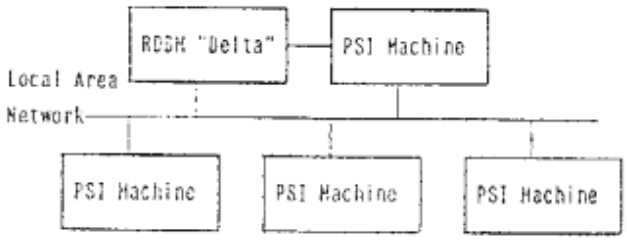

Figure 1 Overview of 1st Stage FGCS

We have already described this principle and developed an experimental system in DEC-10 Prolog [8]. The system illustrated in Figure 2 is composed of a plan generator, a plan converter, and a Delta simulator which executes relational algebra commands using a Prolog fact as a relation's tuple. Here, notice that the Delta simulator simulates only functions of Delta, but it does not simulate the internal behavior of Delta.

The plan generator ** deduces Prolog goals using only the intensional database. At the same time, it generates a plan, which is a sequence of Prolog predicates in the extensional database. This sequence of predicates are verified using the relational database machine. The plan converter converts this plan into a sequence of relational algebra commands, which is accepted by the Delta simulator. This process is repeated to obtain the entire set of answers.

For example, the predicates

idbl((ancestor(X,Y) :- parent(X,Y))). (1)

idbl((ancestor(X,Y) :- parent(X,Z),
ancestor(Z,Y))). (2)

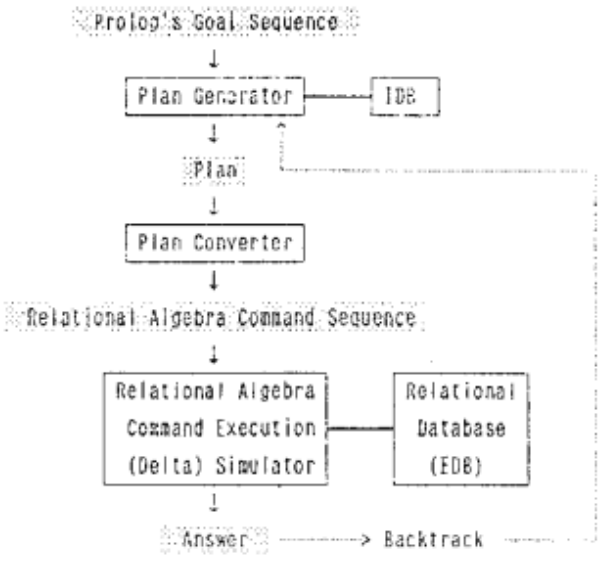

Figure 2 Previous System Configuration

** This plan generator is based on Bowen's [2] meta-predicate *demo*, used to represent provability in logic. See also Chakravarthy [4], Kunifuji [8].

are given in the intensional database as a Prolog program to relate an ancestor with a parent. The predicate of *ancestor(X,Y)* means *Y* is the ancestor of *X* and the predicate of *parent(X,Y)* means *Y* is the parent of *X*. If the *parent* predicates are located in the extensional database as a relation, the following clause is added to the intensional database to indicate this fact.

idbl((parent(X,Y) :- edb(parent(X,Y))). (3)

The clauses stored in the intensional database will be modified into one-argument unit clauses which must be identified as a user program in the intensional database. Thus we can recognize different worlds in our system using different functor names, for example, *idbl* and *idb2* (Figure 3).

If we invoke the plan generator for the query "?-ancestor(taro,X)." to find the ancestors of *taro*, we will get a plan which invokes a search for the parents of *taro* as follows:

[parent(taro,X)] (4)

Then the plan converter converts this into relational algebra commands as follows:

[selection(parent,[1]=taro,temp1),
projection(temp1,[2],temp2), (5)
get(temp2)] ***

and sends them to the Delta simulator. The simulator

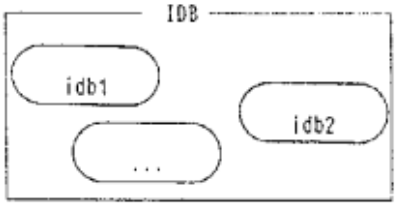

Figure 3 Multiple idb Worlds

*** In the relational algebra commands used in this paper, [n] means n-th attribute of a relation.

searches the relation named *parent* for tuples, the first attribute of which is *taro*, and forms a relation *temp1* as the collection of those tuples. Then it forms a relation *temp2* which has only the second attribute of *temp1*, as a candidate for the results of the query. If the result is not satisfactory to the human, we request a backtrack. But the simulator and plan converter have no alternative results for the plan, so the backtracking arrives at the plan generator. Then the plan generator generates the next alternative plan which again invokes a search for the grandparents of *taro* as follows:

[parent(taro,X1),parent(X1,X)] (6)

The plan converter translates this into

[selection(parent,[1]=taro,temp3),
join(temp3,parent,[2]=[1],temp4),
projection(temp4,[4],temp5), (7)
get(temp5)]

The simulator then forms another relation *temp5*. If a new backtrack takes place, another plan, such as

[parent(taro,X1),parent(X1,X2),parent(X2,X)] (8)

is formed, and so on. The processes of getting a result are controlled by the human. Each plan yields several results. A detailed description of this system can be found in [8].

3. Improvement

In this paper, we describe the refinement of the system into a more efficient one which generates optimized plans much faster than our old system and which can automatically handle LFP operations.

Each time Prolog processes a query, it returns the next answer which satisfies that query. It has achieved this using only information from the intensional database. There are other plans which can also satisfy a given

query. Figure 4 illustrates an example of the relationship between a search tree and plans.

In our previous system, to get all the results, a user had to constantly interact with the process, invoking compulsory backtracks as well as interactive Prolog deduction. It was difficult to know when all the results had been obtained. If the system automatically backtracked, the result would be a recursive program which could not be automatically stopped. For instance, the clause presented in (3) means that the *parent* predicate always exists in the extensional database and that the recursion of (2) has no stop-condition. From another point of view, program -(1),(2),(3)- can be regarded as an LFP operation. Our older implementation [8] did not handle LFP operations in a completely automatic way.

To handle LFP operations automatically, the system must always invoke a compulsory backtrack to obtain an alternative plan and distinguish an empty-backtrack from a nonempty-backtrack. The empty-backtrack is generated only if there is no new tuple satisfying the plan, otherwise the nonempty-backtrack is generated. Aho [1] presented an iterative procedure (Figure 5) to support LFP operations. The nonempty-backtrack corresponds to the while loop because of dissatisfaction of the condition $(R' \neq R)$, and the empty-backtrack corresponds to the case of satisfaction of the condition $(R' = R)$.

Prolog cannot discriminate among causes of the backtrack because the binding information of variables are freed when backtracking occurs. We decided to use

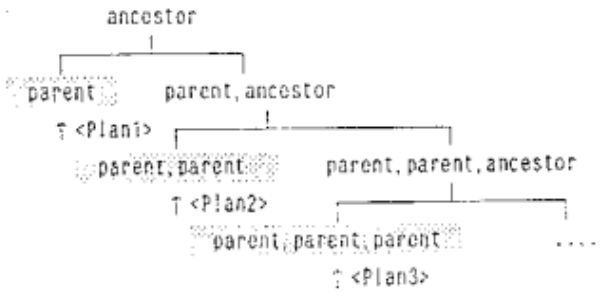

Figure 4 Search-tree/Plan Relation

$R \leftarrow R0$
do
$R' \leftarrow R$
$R \leftarrow R \cup f(R)$
while $(R' \neq R)$

Figure 5 Iterative Procedure of LFP operation

side-effects of Prolog such as **assert** and **retract** to discriminate between the two types of backtrack. When no new tuple is detected for a relational algebra command sequence, an *empty* predicate which acts as a flag is asserted (created) to indicate the empty-backtrack, before a backtrack takes place. When at least one new tuple is found, nothing is asserted to indicate the nonempty-backtrack, before the backtrack takes place. Figure 6 shows the new process flow. Figure 7 shows the main part of the new system program.

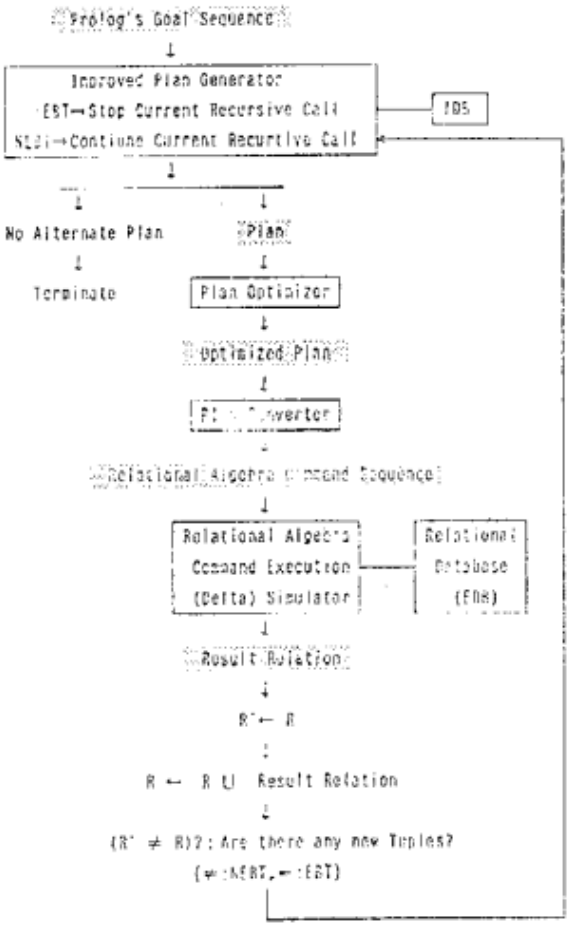

Figure 6 Improved System Configuration

```
go(IDB,Goal):- generate(IDB,Goal,_,Plan),
        optimize(Plan,OptPlan),
        convert(OptPlan,Command),
        delta(Command,Relation),
        backtrack(Relation).
go(_,_):- listing(out).

backtrack(Relation):-
        delta([copy(out,outl),
              union(Relation,out),
              equal(outl,out)],eq_ans),
        eq_ans(no), !,
        retract(empty), fail.
backtrack(_):- eq_ans(yes), !,
        assert(empty), fail.
```

Figure 7 Main Part of System Program

The predicate *generate* (Figure 7) which corresponds to the plan generator generates plans by checking whether *empty* is asserted or not. If there is no *empty*, the plan generator grows the branch along which the previous result exists. If there is *empty*, it grows other branches and retracts the *empty* by checking it next. This checking works as a check of stop conditions for the recursive program. The time when checking takes place depends on the structure of the program. It is necessary to add a few specifications to recursive programs as stop conditions. We decided to use a predicate *check* for indicating that timing. When a user asserts a clause which is a recursive program, he must add this *check* predicate to the top of condition parts of that clause, in order to make the plan generator check the existence of the predicate *empty* when it grows the search tree. For instance, clause (2) is changed into

```
idb1((ancestor(X,Y):- check,
        parent(X,Z),ancestor(Z,Y))).        (9)
```

This specification is not so extraordinary, since the predicate *parent* in (2) acts as a stop condition of recursion. If we interchange this predicate with the predicate *ancestor*, it means a change of program style from a right-recursive call to a left-recursive call. This recursive call cannot be stopped by present Prolog processors, although the program is not logically wrong. The *check* predicate is used simply to make that stop condition clear. This system can also handle the left-recursive program by using the predicate *check*. In that case, search strategy is changed from depth-first to breadth-first. Whichever strategy we may use, the set of the last results is the same when we get all the results for a query.

There can be any number of recursions in a program, all of which use only one type of *check* as a stop-condition-checking predicate. This is because the same *empty* predicate indicates a stop condition to the currently developing recursion. When the currently developing recursion is stopped by the *check* predicate, the program then returns to the caller. The caller's recursive call could also be stopped by the same *check* predicate, by this time the *empty* predicate is already retracted at the end of the former stop condition checking.

To examine the convergence condition of LFP operations, the system has two relations *out* and *outl* which correspond to R and R' (Figure 5), respectively. It executes a union between these two relations. The *backtracking* predicate (Figure 7) checks equality between *out* and *outl* and discriminates between nonempty-backtracks and empty-backtracks. Initially, *out* and *outl* are empty.

If the system were to merely check tuple generation per plan and invoke an empty-backtrack only when there is no generated tuples, it could not necessarily stop some types of recursions even if the program used the *check* predicates. If we accept facts which will make the plan generation cyclic, for example, *parent(taro,taro)* or a pair facts such as *parent(taro,hanako)* and *parent(hanako,taro)* in the extensional database, program (1),(3),(9) will not stop, even if it contains *check* predicates. In that case, the user must check the integrity constraints of database [7] or carefully specify his query as

"?-ancestor(X,Y),X\=Y." Since the new system checks not only tuple generation but also checks whether the generated tuples are new or not using the set operation, it can handle cyclic data in the database without integrity constraint checking and without explicit restriction of query. If the converted relational algebra expression is monotonic [1], this system is not only capable of obtaining all the tuples which satisfy a query, but can also terminate.

One of the principal goals of this improvement is the speedup of plan generation and optimization of the plan as well as the capability to handle LFP operation. We will describe how to implement that speedup and optimization in the following sections. There are a few other modifications to the system. Our previous paper [8] showed how to handle evaluable predicates and not-predicate. It did not show how to implement them. There was a bug in the plan converter when a p(X,X) type predicate was given. In the improved system, we have changed the plan converter to deal with evaluable predicates, not-predicate, and p(X,X) type predicates. The Delta simulator is unchanged.

The evaluable predicates contained in a user's query are put in the relational algebra commands as conditions of *selection* or *join* by the plan converter. The not-predicate is transformed to difference command by the plan converter, so it imposes restrictions on the space of variables which appear in the predicates placed before the not-predicate. Here, a not-predicate cannot be placed on top of a goal sequence. In the present Prolog system, the not is implemented as "Negation as Failure" by using cut-operator. The not described herein is not same to that of "Negation as Failure". If users want to use the not in Prolog sense, they should so define the not.

The modified plan converter can now be thought of as translating relational calculus into relational algebra. Only *selection*, *join*, and *projection* were used in the converted plan (relational algebra commands) when plans did not contain a not.

4. Improved Plan Generator

The implementation of the plan generator proposed in our previous paper [8] was complex enough to process even unifications by program. It was time consuming to generate a plan, especially when searching a clause for variables which are unifiable by analyzing its structure. To generate plans faster, we implemented a more efficient plan generator which utilizes Prolog's built-in unification facility to deduce Prolog's queries. In [7], we developed an enhanced *demo* predicate [2] based on the *execute* predicate found in "A Prolog Interpreter in Prolog" proposed in [4], which can also deal with Prolog's cut-operator (!). Figure 8 shows the main part of the new plan generator, *generate*, which uses the idea of the enhanced *demo* predicate. The size of the program has been reduced to about one fifth that of the previous one.

The principle of deduction which evaluates predicates located in the extensional database in deferred manner has not changed. The previous plan generator deduces a goal sequence from the top of the sequence. When it finds a predicate located in the extensional database, it sends the predicate to the tail of the sequence. This deduction stops when the predicate which is sent at first appears at the top of the sequence. At this time, the sequence contains only the predicates located in the extensional database, and the sequence represents a plan for the goal sequence. Our improved plan generator is able to deduce the goal sequence as well as our previous generator. When it finds a predicate located in the extensional database, it appends the predicate to a list for output.

In Figure 8, the first argument of the *generate* predicate indicates the world name of the intensional database. Thus the plan generator can deduce in one of the multiple worlds (Figure 3) by indicating the world name as the value of that argument. The second argument takes the Prolog query as input. The last one gives a sequence of transformed goals as output. We use a difference list $d(X,Y)$ for output in which X points to the head of the list and Y points to the tail of the list to

```
generate(IDB,true,_,d(X,X)):- !.
generate(IDB,!,_,d(X,X)).
generate(IDB,!,cut,d(X,X)).
generate(IDB,check,_,d(X,X)):-
        (empty,!,retract(empty),fail;true).
generate(IDB,edb(P),_,d([P|X],X)):-!.
generate(IDB,not(P),V,d([not(L)|X],X)) :-
        !, generate(IDB,P,V,L).
generate(IDB,(P;Q),V,L) :-
        !, (generate(IDB,P,V,L); generate(IDB,Q,V,L)).
generate(IDB,(P,Q),V,d(X,Y)) :-
        !, generate(IDB,P,C,d(X,Z)),
        (C==cut, V=cut, !; generate(IDB,Q,V,d(Z,Y))).
generate(IDB,P,V,L) :-
        idb_clause(IDB,P,Q), generate(IDB,Q,V,L),
        (V==cut, !, fail; true).
```

Figure 8 Main part of the New Plan Generator (Based on [4,7])

make list processing efficient. The third argument is used as work space to handle the cut-operator. This cut-operator only controls the production of a plan, which enables the control of search space. This operator simulates the control function of the DEC-10 Prolog cut-operator. The predicate, *idb_clause(IDB,P,Q)*, searches the intensional database for a predicate which is unifiable with *P* and returns the predicate's body as *Q* with the unified variables.

Notice that certain kinds of evaluable predicates (e.g. ">", "=", "\=") are not evaluated here. Instead, they are passed through as a condition of *selection* or *join* to the plan converter, and evaluated later in the relational algebra command execution (Delta) simulator. The plan generator has no part which directly handles these kinds of predicates. To pass it to the plan converter, there needs to be a corresponding predicate in the intensional database; for example,

```
idb((X=Y:-edb(X=Y))).  (10)
```

is needed for the evaluable predicate "=".

This plan generator, which is compiled in DEC-10 Prolog on a DEC-2060, generates plans in about 5 to 110 ms (CPU time) for the above examples. This is about 20 to 50 times faster than the previous plan generator. Further speedup can be obtained as the depth of the search tree becomes greater, since the counts of dynamic loop in the program decrease.

5. Plan Optimizer

In the previous system, redundant relational algebra commands and redundant temporary relations were produced. For example, the temporary relation *temp3* in (7) has exactly the same information as the relation *temp2* in (5), so *selection(parent,[1]=taro,temp3)* in (7) is not necessary if the relation *temp2* is kept. This means that the subplan *parent(taro,X1)* in (6) can be replaced with *temp2(X1)*.

To avoid the production of redundant commands and relations, generated plans and relation forms are stored in pairs. A sample pair for (4) and (5) is

```
plan([parent(taro,X)],temp2(X)).  (11)
```

A search is made for a match between subplans in the new plan and in the set of previous plans using a string matcher (Figure 9) made from *append* predicates. By replacing the subplan from the new plan with the matched subplan from a previous plan, potential redundancy is removed. This string matcher can search substrings of arbitrary length which can be found anywhere in the string. The predicate *eq_struct* in Figure 9 checks the equality of two arguments' structures, since simple unification between two arguments causes harmful instantiations.

The plan optimizer is placed between the plan generator and the plan converter (Figure 6). Using the plan optimizer, for example, the plan of (6) is translated into

```
optimize(IN,OUT1) :-
        plan(PL1,Rel),
        append(TPL,Rest,IN),
        append(Top,PL2,TPL),
        eq_struct(PL1,PL2),
        PL1 = PL2,
        append(Top,[Rel],TO),
        append(TO,Rest,OUT2),
        optimize(OUT2,OUT1).
optimize(X,X).
```

Figure 9 Main part of the Plan Optimizer

```
[temp2(X1),parent(X1,X)]                    (12)
```

and the relational algebra command for (7) is replaced by

```
[join(temp2,parent,[1]=[1],temp3)
    projection(temp3,[3],temp4)].           (13)
```

It is a basic and simple optimization, but it greatly improves the efficiency of the Prolog/relational algebra interface. This amplifies effectiveness when the program contains recursive call such as in the above example.

6. Conclusion

In this paper, we described an efficient interface for use between a logic programming language and relational algebra. Our interface provides the means for processing a useful database query language capable of handling LFP operations. We showed how we improved the system by enabling it to handle LFP operations, the cut-operator, evaluable predicates and the not-predicate. We showed that a faster plan generator and a plan optimizer was possible. Both improvements make this interface practical for actual use.

APPENDIX

This appendix shows execution examples of interfacing program between Prolog and relational database.

```
| ?- go(ide1,ancestor(taro,A)).

plan :  [father(taro,_40)]
opt_plan :  [father(taro,_40)]
command :  [selection(father,[1]=taro,temp1),projection(temp1,[2],temp2)]
memo :  plan([father(taro,_40)],temp2(_40))
eq_ans :  no

plan :  [mother(taro,_40)]
opt_plan :  [mother(taro,_40)]
command :  [selection(mother,[1]=taro,temp3),projection(temp3,[2],temp4)]
memo :  plan([mother(taro,_40)],temp4(_40))
eq_ans :  no

plan :  [father(taro,_210),father(_210,_40)]
opt_plan :  [temp2(_210),father(_210,_40)]
command :  [join(temp2,father,[1]=[1],temp5),projection(temp5,[3],temp6)]
memo :  plan([father(taro,_210),father(_210,_40)],temp6(_40))
eq_ans :  no

plan :  [father(taro,_84),mother(_84,_14)]
opt_plan :  [temp2(_84),mother(_84,_14)]
command :  [join(temp2,mother,[1]=[1],temp7),projection(temp7,[3],temp8)]
memo :  plan([father(taro,_84),mother(_84,_14)],temp8(_14))
eq_ans :  no

plan :  [father(taro,_84),father(_84,_149),father(_149,_14)]
opt_plan :  [temp6(_149),father(_149,_14)]
command :  [join(temp6,father,[1]=[1],temp9),projection(temp9,[3],temp10)]
memo :  plan([father(taro,_84),father(_84,_149),father(_149,_14)],temp10(_14))
eq_ans :  yes

plan :  [father(taro,_84),father(_84,_149),mother(_149,_14)]
opt_plan :  [temp6(_149),mother(_149,_14)]
command :  [join(temp6,mother,[1]=[1],temp11),projection(temp11,[3],temp12)]
memo :  plan([father(taro,_84),father(_84,_149),mother(_149,_14)],temp12(_14))
eq_ans :  yes

plan :  [father(taro,_84),mother(_84,_149),father(_149,_14)]
opt_plan :  [temp6(_149),father(_149,_14)]
command :  [join(temp6,father,[1]=[1],temp13),projection(temp13,[3],temp14)]
memo :  plan([father(taro,_84),mother(_84,_149),father(_149,_14)],temp14(_14))
eq_ans :  yes

plan :  [father(taro,_84),mother(_84,_149),mother(_149,_14)]
opt_plan :  [temp6(_149),mother(_149,_14)]
command :  [join(temp6,mother,[1]=[1],temp15),projection(temp15,[3],temp16)]
memo :  plan([father(taro,_84),mother(_84,_149),mother(_149,_14)],temp16(_14))
eq_ans :  yes

plan :  [mother(taro,_84),father(_84,_14)]
opt_plan :  [temp4(_84),father(_84,_14)]
command :  [join(temp4,father,[1]=[1],temp17),projection(temp17,[3],temp18)]
memo :  plan([mother(taro,_84),father(_84,_14)],temp18(_14))
eq_ans :  no

plan :  [mother(taro,_84),mother(_84,_14)]
opt_plan :  [temp4(_84),mother(_84,_14)]
command :  [join(temp4,mother,[1]=[1],temp19),projection(temp19,[3],temp20)]
memo :  plan([mother(taro,_84),mother(_84,_14)],temp20(_14))
eq_ans :  no

plan :  [mother(taro,_84),father(_84,_159),father(_159,_14)]
opt_plan :  [temp18(_159),father(_159,_14)]
command :  [join(temp18,father,[1]=[1],temp21),projection(temp21,[3],temp22)]
memo :  plan([mother(taro,_84),father(_84,_159),father(_159,_14)],temp22(_14))
eq_ans :  yes

plan :  [mother(taro,_84),father(_84,_159),mother(_159,_14)]
opt_plan :  [temp18(_159),mother(_159,_14)]
command :  [join(temp18,mother,[1]=[1],temp23),projection(temp23,[3],temp24)]
memo :  plan([mother(taro,_84),father(_84,_159),mother(_159,_14)],temp24(_14))
eq_ans :  yes
```

```
plan : [mother(taro,_84),mother(_84,_159),father(_159,_14)]
opt_plan : [temp20(_159),father(_159,_14)]
command : [join(temp20,father,[1]=[1],temp25),projection(temp25,[3],temp26)]
memo : plan([mother(taro,_84),mother(_84,_159),father(_159,_14)],temp26(_14))
ec_ans : yes

plan : [mother(taro,_84),mother(_84,_159),mother(_159,_14)]
opt_plan : [temp20(_159),mother(_159,_14)]
command : [join(temp20,mother,[1]=[1],temp27),projection(temp27,[3],temp28)]
memo : plan([mother(taro,_84),mother(_84,_159),mother(_159,_14)],temp20(_14))
ec_ans : yes

out(ichiro).
out(hanako).
out(yasuo).
out(keiko).
out(shigeki).
out(etsuko).

A = _14

yes
| ?- listing([idb1,father,mother,temp1,temp2]).

idb1((ancestor(_1,_2):-parent(_1,_2))).
idb1((ancestor(_1,_2):-check','parent(_1,_3)','ancestor(_3,_2))).
idb1((parent(_1,_2):-father(_1,_2))).
idb1((parent(_1,_2):-mother(_1,_2))).
idb1((father(_1,_2):-edb(father(_1,_2)))).
idb1((mother(_1,_2):-edb(mother(_1,_2)))).
idb1((_1=_2:-edb(_1=_2))).
idb1((_1\=_2:-edb(_1=\=_2))).
idb1((_1>_2:-edb(_1>_2))).
idb1((_1>=_2:-edb(_1>=_2))).
idb1((_1=<_2:-edb(_1=<_2))).
idb1((_1<_2:-edb(_1<_2))).

father(taro,ichiro).
father(ichiro,yasuo).
father(hanako,shigeki).

mother(taro,hanako).
mother(ichiro,keiko).
mother(hanako,etsuko).

temp1(taro,ichiro).

temp2(ichiro).


| ?- go(idb2,manager_one(X,Y)).

plan : [manager(_38,_54),not(d([manager(_38,_54),manager(_38,_289),_54\==_28
9|_292],_292))]

plan : [manager(_38,_54),manager(_38,_289),_54\==_289]
opt_plan : [manager(_38,_54),manager(_38,_289),_54\==_289]
command : [join(manager,manager,[1]=[1],temp4),selection(temp4,[2]\==[4],tem
p2),projection(temp2,[1,2],temp3)|_405]
memo : plan([manager(_38,_54),manager(_38,_289),_54\==_289],temp3(_38,_54))

opt_plan : [manager(_38,_54),not(temp3(_38,_54))]
command : [join(manager,manager,[1]=[1],temp4),selection(temp4,[2]\==[4],tem
p2),projection(temp2,[1,2],temp3),difference(manager,temp3,temp4),projection(
temp4,[1,2],temp5)]
memo : plan([manager(_38,_54),not(temp3(_38,_54))],temp5(_38,_54))
ec_ans : no

out(clark,scott).
out(smith,martin).

X = _38,
Y = _54


yes
| ?- listing([idb2,manager]).

idb2((manager_many(_1,_2):-manager(_1,_2)','manager(_1,_3)','_2\==_3)).
idb2((manager_one(_1,_2):-manager(_1,_2)','not(manager_many(_1,_2)))).
idb2((manager(_1,_2):-edb(manager(_1,_2)))).
idb2((_1=_2:-edb(_1=_2))).
idb2((_1\=_2:-edb(_1=\=_2))).
idb2((_1>_2:-edb(_1>_2))).
idb2((_1>=_2:-edb(_1>=_2))).
idb2((_1=<_2:-edb(_1=<_2))).
idb2((_1<_2:-edb(_1<_2))).

manager(jones,blake).
manager(jones,clark).
manager(jones,smith).
manager(blake,allen).
manager(blake,turner).
manager(clark,scott).
manager(smith,martin).
```


Acknowledgement

The authors thank Mr. Y. Tanaka of Hokkaido University and Mr. H. Katsuno of the NTT Musashino Telecommunications Laboratory for their useful comments, and Mr. D. S. Anderson of Xytec Systems for his helpful advice on the refinement of the paper.

REFERENCES

[1] Aho, A. V. and Ullman, J. D. Universality of Data Retrieval Languages. In *Proc. of ACM/SIGPLAN Conf. on of Programming Languages*, San Antonio, Jan. 1979, pp.110-117.

[2] Bowen, K. A., and Kowalski, R. A. Amalgamating Language and Metalanguage in Logic Programming. In Clark, K. L. and Tarnlund, S. -A. (Eds.), *Logic Programming*, Academic Press, 1982.

[3] Chakravarthy, U. S., Minker, J., and Tran, D. Interfacing Predicate Logic Languages and Relational Databases. In *Proc. of the first Int. Logic Programming Conf.*, Faculte des Sciences de Luminy Marseilles, France, Sept. 14-17th 1982, pp.91-98.

[4] Coelho, H., Cotta, J. C., and Pereira, L. M. *HOW TO SOLVE IT WITH PROLOG*. 2nd. edition, Laborato'rio Nacional de Engenharia Civil, LISBOA, 1980.

[5] Gallaire, H. Logic Data Bases vs Deductive Data Bases. In *Proc. of Logic Programming Workshop*, Algrave/PORTUGAL, June 1983, pp.608-622.

[6] ICOT (Ed.): *Outline of Research and Development Plans for Fifth Generation Computer Systems*. May 1982.

[7] Kernel Language Design Group of ICOT Research Center: Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1). ICOT Technical Memorandum (to appear).

[8] Kunifuji, S. and Yokota, H. PROLOG and Relational Data Base for Fifth Generation Computer Systems. In Gallaire, H., Minker, J., and Nicolas, J. M. (Eds.) *Logical Bases for Data Bases*, ONERA-CERT, Toulouse, Dec. 1982.

[9] Murakami, K., Kakuta, T., Miyazaki, N., Shibayama, S., and Yokota, H. A Relational Data Base Machine: First Step To Knowledge Base Machine. In *Proc. of 10th Annual International Symposium on Computer Architecture*, Stockholm/SWEDEN, June 1983, pp.423-426.

[10] Naqvi, S. A., and Henschen, L. J. Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs. In *Proc. of 8th IJCAI*, Karlsruhe/West Germany, Aug. 1983, pp.25-28.

[11] Nishikawa, N., Yokota, M., Yamamoto, A., Taki, K., and Uchida, S. The Personal Sequential Inference Machine (PSI): Its Design and Machine Architecture. In *Proc. of Logic Programming Workshop*, Algrave/PORTUGAL, June 1983, pp.53-73.

[12] Shibayama, S., Kakuta, T., Miyazaki, N., Yokota, H., and Murakami, K. A Relational database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor. to appear in *New Generation Computing*, Ohmsha, Springer-Verlag, 1984.

[13] Tanaka, Y., Horiuchi,K., and Tagawa,R. Combining Inference System and Data Base System by a Partial Evaluation Mechanism. In *Proc. of First Knowledge Engineering Symposium*, Tokyo, March, 1983 (in Japanese).