TR-123

Deductive Database System based on Unit Resolution

by

H. Yokota, K. Sakai and H. Ito

June, 1985

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

Deductive Database System based on Unit Resolution

Haruo Yokota, Kō Sakai, Hidenori Itoh

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

June 1985

ABSTRACT

This paper presents a methodology for constructing a deductive database system consisting of an intensional processor and a relational database management system. A setting evaluation approach is introduced. The intensional processor derives a setting from the intensional database and a given goal and sends the setting and the relationship between setting elements to the management system. The management system performs a unit resolution with setting using relational operations for the extensional databases. An extended least fixed point operation is introduced to terminate all types of recursive queries.

1. Introduction

One of the current topics in both database research and artificial intelligence is how to combine relational databases with first-order logic. The use of logic programming languages for relational database queries is regarded as an improvement over database query languages for handling integrity constraints and transitive closures of relations [Gallaire et al. 84]. From an artificial intelligence point of view, the logic programming language is an attractive choice for manipulating knowledge. The combination with relational databases can be seen as an experiment in knowledge base system construction [Murakami et al. 83]. Development of a knowledge information processing system is a goal of the Fifth Generation Computer Systems (FGCS) project in Japan. A database in which queries have the form of first-order formulas is called a deductive database [Gallaire 83, Gallaire et al. 84].

Reiter proposed a method for decomposing a question-answering system into intensional and extensional processors [Reiter 78]. The extensional processor is a relational database management system while the intensional processor compiles first-order queries using a theorem prover. Besides compiling the queries, the intensional processor can also access the extensional processor by interpreting the queries. The former method is referred to as the compiled approach and the latter as the interpreter approach [Chakravarthy et al. 82]. Because of the separation of processing mechanisms, interface overhead between the two processors is a very important factor in the efficiency of the whole system. We expect that the interpreter approach will generate more overhead than the compiled approach.

Kunifuji and Yokota implemented a deductive database system using a deferred evaluation method, one type of the compiled approach [Kunifuji and Yokota 82]. However, it cannot automatically handle transitive closures of relations. The deductive database system needs to be able to handle recursive queries, since a recursive call program can be written in the logic programming language. A number of methods for handling recursive queries have been proposed. Some approaches transform the recursive query into a non-recursive iterative program [Chang 81, Naqvi 83]. Others use a connection graph to analyze the recursive call [Henschen 84, Ullman 84]. Yokota et al. improved on the deferred evaluation approach to terminate recursive query evaluation using a least fixed point operation [Yokota et al. 84]. Each of these methods, however, can handle only certain types of recursive queries.

In this paper we first consider why deferred evaluation using the least fixed point operation cannot terminate all types of recursive queries. The reason is briefly that deferred evaluation has the intensional processor drive the database management system in top-down manner, based on input resolution. We propose setting evaluation and an extended least fixed point operation. Setting evaluation is based on a kind of semantic resolution, a unit resolution using setting. In Section 2 we briefly present our previous work investigating the reason why deferred evaluation cannot handle all types of iterations. Section 3 describes various resolution procedures as a basis for setting evaluation. Section 4 describes the strategy of the setting evaluation approach and the extended least fixed point operation and discusses the completeness and efficiency of the approach. The results and observations are summarized in Section 5.

2. Background

We assume the reader is familiar with relational databases [Ullman 82] and the resolution method for first-order logic [Chang and Lee 73, Loveland 78].

The logic programming language we use is a collection of Horn clauses, each clause containing at most one positive literal. If a clause is composed of a positive literal alone, it is called a *unit clause*. If there are no variables among its arguments, it is called a *ground unit clause*. A finite set of ground unit clauses is called an *extensional database* (EDB). If a clause is composed of negative literals only, we call it a *goal*. Horn clauses other than goals are called *definite clauses*. A finite set of definite clauses that are not ground unit clauses is called an *intensional database* (IDB). Throughout this paper we express Horn clauses in Prolog style. Strings beginning with lowercase letters denote constants, while those beginning with uppercase letters denote variables. We use uppercase letters also to express literals abstractly. A :- $B_1, .., B_n$. intuitively means that B_1 and ... and B_n implies A.

2.1 Plan Generation

Since the capacity of the extensional database is expected to be enormous, it is inappropriate to manage the extensional database in main memory. It is not enough to derive one answer at a time, as Prolog processors do. Instead, the answers have to be returned as a set. The extensional database can be regarded as a relational database by treating the ground unit clauses as tuples of relations. A large relational database is efficiently handled by a database machine or database management system and the answers to queries are returned in table form. The intensional processor's job is to direct the relational database management system to search the (extensional) databases according to the intensional database and a given goal. We call the directive a *plan*.

In deferred evaluation, a given goal is resolved with an intensional database clause in the usual way (left-to-right) until a literal resolved with an extensional database clause appears. The literal is appended to the plan list which is initially empty, and the resolution process using the intensional database is continued. When the resolution process is finished, the plan list obtained becomes a plan for the goal. The plan is transformed into a sequence of relational algebra operations for the extensional database. The resulting relation for the given goal is derived in the extensional database using these relational algebra operations [Kunifuji and Yokota 82]. Any relation that can be derived from the intensional database in this way is called a *virtual* relation.

Example 2.1: An intensional database (1) through (3) is given.

(1) $uncle(X,Y)$:- $parent(X,Z), brother(Z,Y)$.
(2) $parent(X,Y)$:- $father(X,Y)$.
(3) $parent(X,Y)$:- $mother(X,Y)$.

The ground unit clauses stored in the extensional database are indicated as follows:

(4) $father(X,Y)$:- $edb(father(X,Y))$.
(5) $mother(X,Y)$:- $edb(mother(X,Y))$.
(6) $brother(X,Y)$:- $edb(brother(X,Y))$.

If the given goal is

(7) $:-\ uncle(hart, X).$

then one of the plans generated is

(8) $[father(hart, Z), brother(Z, Y)]$

The resolution and plan generation processes for the goal are illustrated in Figure 2.1. There is an alternative plan corresponding to the alternative parent definition. Plan (9) is generated as an alternative.

(9) $[mother(hart, Z), brother(Z, Y)]$

Each plan is also delivered to the relational database management system. The relation for a plan is generated by *projection*, *selection*, and *equi-join* operations determined by the plan. The relation for plan (8) is derived as follows:

$$temp1 \leftarrow selection(father, first\text{-}attribute{=}hart)$$
$$temp2 \leftarrow equi\text{-}join(temp1, brother, second\text{-}attribute{=}\ first\text{-}attribute)$$
$$result \leftarrow projection(temp2, second\text{-}attribute)$$

The result relation for goal (7) is a set derived by *union* operations between the relations generated by (8) and (9).

■

— resolvent —	— plan list —
$parent(hart, Z), brother(Z, Y)$	[ ]
$father(hart, Z), brother(Z, Y)$	[ ]
$brother(Z, Y)$	$[father(hart, Z)]$
[ ]	$[father(hart, Z), brother(Z, Y)]$

Figure 2.1 Resolution and plan generation process

2.2 Content Dependent Plan Generation

The intensional database may also contain recursively defined clauses. These clauses can be evaluated iteratively in the logic programming language. As we introduce iterations, we must consider their termination. In Prolog programs, a termination condition is implicitly indicated by the absence of an instance. In deferred evaluation, it is not until ground unit clauses of the extensional database are examined that the absence of an instance for termination is made clear. Thus, the termination depends on the contents of the extensional database.

Example 2.2: Instances of *parent* are stored in the extensional database. The recursively defined intensional database (1) through (3) and a goal (4) are given.

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

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

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

(4) :- $ancestor(X,Y)$.

The following plans are successively generated.

$[parent(X,Y)]$
$[parent(X,Z), parent(Z,Y)]$
$[parent(X,Z), parent(Z,Z1), parent(Z1,Y)]$
$\vdots$

■

The plan generator has no idea how many plans it must generate to derive all instances for the virtual relation called *ancestor*. Therefore, the relational database management system has to inform the plan generator of the termination point by monitoring the contents of the virtual relation. To do this the management system needs to detect a least fixed point (LFP) [Aho and Ullman 79]. The least fixed point operation is performed by *union* and *equality-check* operations between two relations.

Let $R(i)$ be a relation derived from the i-th plan. We can obtain a relation $R'(i)$ that is a result of a *union* operation between $R'(i\text{-}1)$ and $R(i)$. Now, let $R'(0)$ be an empty relation. If the contents of $R'(i)$ are equal to the contents of $R'(i\text{-}1)$, i is the least fixed point. Even though the extensional database contains cyclic instances, e.g. $parent(a,b)$ and $parent(b,a)$, this method can find the least fixed point.

However, the management system does not know when it performs the least fixed point operation, because the pattern of plan generation is not always so straightforward as the above example. If there are alternatives in one level of recursion, the pattern may become more complex. For instance, one's parent is one's father or mother, like Example 2.1. The plans are generated as follows:

$[father(X,Y)]$
$[mother(X,Y)]$
$[father(X,Z), father(Z,Y)]$
$[father(X,Z), mother(Z,Y)]$
$[mother(X,Z), father(Z,Y)]$
$[mother(X,Z), mother(Z,Y)]$
$[father(X,Z), father(Z,Z1), father(Z1,Y)]$
$\vdots$

In this case, the least fixed point operation must be performed after the last plans of each level are performed, i.e. after the plans $[mother(X,Y)]$, $[mother(X,Z), mother(Z,Y)]$, and so on. We introduced a *check* predicate in the previous system [Yokota et al. 84] to indicate the timing from the plan generator when the least fixed point operation is performed. It corresponds to an explicit indication of a termination condition in the logic programming language. For instance, the second clause in Example 2.2 is transformed into the following clause.

(2') $ancestor(X,Y)$:- $check, parent(X,Z), ancestor(Z,Y)$.

The plan generator delivers a least fixed point operation directive to the management system when it evaluates the *check* predicate.

We developed an experimental system using the above method and called it IRIS (Inference/Relational database Interface System). IRIS was demonstrated at the ICOT Open House after the FGCS'84 Conference in November 1984. We used the relational database machine Delta and a microcomputer as a host machine and connected these two machines using the local area network. We used a Prolog processor on the microcomputer.

In fact, however, IRIS cannot handle all types of iterations. It can only terminate simple recursively defined iteration, as in the above example. If more than one termination check point exists in a deduction sequence, for example a conjunction of two recursively defined virtual relations, the plan generator cannot judge which iteration should be terminated, because the management system only informs the plan generator that no new instances can be found for the plan. The Prolog processor checks the absence of instances one by one to decide which iteration should be terminated. Since IRIS generates plans by compilation, however, each plan may contain a number of absence checks corresponding to *check* predicates. It is useless to provide different kinds of *check* predicates. The reason why the termination points cannot be found is that plans are generated top-down using input resolution.

3. Resolution Variation

There may be many irrelevant computations of resolvents in the basic resolution principle proposed by Robinson [Robinson 65]. A number of refinements of resolution have been proposed to prevent these useless resolvents from being generated [Chang and Lee 73, Loveland 78]. The basic strategy of these refinements is to divide a set of clauses into two groups and prevent clauses within the same group from being resolved with each other. It can significantly reduce the number of resolvents computed. Resolution adopting these refinements are called *semantic resolutions*.

Input resolution and *unit resolution* are well known as complete resolution procedures for Horn clauses. Both can be seen as kinds of semantic resolutions.

Input resolution takes one of the parents of a resolvent as the given goal or the resolvent computed in a former step. The other parent, called an input clause, is a member of the definite clause set. Most of the implemented Prolog processors use the input resolution procedure.

Unit resolution requires one of the parents of a resolvent to be a unit clause. Resolution using a set of unit clauses is another kind of semantic resolution. Since the extensional database consists of ground unit clauses, unit resolution seems best suited for the deductive database system. However, there may still be many irrelevant computations even in unit resolution, because the resolvents are computed independently of the given goal.

Loveland introduced a setting to prove the completeness of semantic resolution for Horn clauses [Loveland 78]. Now we consider how to use a kind of setting to prevent irrelevant resolvents in unit resolution from being computed.

4. Setting Evaluation

4.1 Setting Assembly Algorithm

The setting is derived from the given goal and definite clause set H to provide an efficient theorem proving procedure for a logic programming language. The following algorithm is used to assemble the setting.

Step 1 : Let both S and T be empty sets.

Step 2 : Enter all literals in the given goal into both S and T.

Step 3 : If T is empty, then S is the desired setting. Otherwise, go to the next step.

Step 4 : Remove a (negative) literal, $\neg L$, from T.

Step 5 : Let A be an empty set.

Step 6 : Search all definite clauses for a clause C which is not a member of A and whose positive literal is unifiable with L. If there is no clause satisfying these restrictions, then go to step 3. Otherwise, go to the next step.

Step 7 : Search C for atoms whose complements are not members of S and add their complements to both S and T.

Step 8 : Add C to A and go to step 6.

This algorithm terminates since the number of literals in H is finite.

Just by looking at the above algorithm, it is clear that the obtained setting S contains the complements of all atoms involved in clauses used to derive the resolution. In other words, if we derive a setting S from H using the setting assembly algorithm, a unit parent clause of a resolvent is the complement of some literal in S. Okuno proposed a similar algorithm to determine a setting [Okuno 83]. However, his algorithm can only collect ground unit clauses used for resolution.

4.2 Unit Resolution with Setting using Relational Operations

We now assume that the ground unit clauses are in the extensional database. We derive the setting S from the given goal and only the intensional database by using the setting assembly algorithm. Then some of the elements of the setting S contain negative literals in the form *edb* (L) where L is a literal unifiable with ground unit clauses in the extensional database. Thus, we can set up the ground unit clauses from the extensional database to be used in the resolution procedure.

Our objective is to derive all answers for the given goal. It is not effective to transport the applicable ground unit clauses from the extensional database to the intensional processor to perform the unit resolution in the processor. Instead, we consider a method for performing unit resolution in the extensional processor (relational database management system) using the setting.

When we examine each step of the unit resolution procedure, a resolution between a parent unit clause C_1 and another parent clause C_2 can be regarded as a search for a complementary instance (C_1) of the negative literal in C_2. The negative literals in any unit resolution step for the given goal are members of the setting. We let the relational database management system enumerate all complementary instances for each element of the setting. Then all answers for the goal can be obtained, since the goal is involved in the setting as an element.

If an element of the setting is of the form *edb* (L), instances for L are tuples of the relation corresponding to the ground unit clauses unifiable with L. Other formed elements appear in the intensional database as positive literals. Instances of positive literals can be enumerated from complementary instances of negative literals in the same clause by using relational algebra operations (Figure 4.1). If a clause has only one negative literal, instances for the positive literal can be derived using a *projection* operation for the virtual relation corresponding to the negative literal. If a clause has at least two negative literals, their common variable indicates an *equi-join* relationship between the virtual relations corresponding to them. If there is more

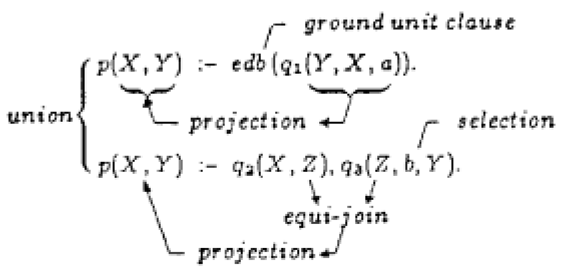
ground unit clause
p(X, Y) :- edb(q1(Y, X, a)).
union
projection
selection
p(X, Y) :- q2(X, Z), q3(Z, b, Y).
equi-join
projection

Figure 4.1 Unit resolution using relational operations

than one clause with the same positive literal, complementary instances for the element are derived by using *union* operations between virtual relations corresponding to these clauses. If the literal has constants as its arguments, restricted tuples using a *selection* operation can be selected.

Example 4.1: Consider an intensional database (1) through (3) and a goal (4).

(1) $uncle(X,Y)$:- $parent(X,Z), edb(brother(Z,Y))$.

(2) $parent(X,Y)$:- $edb(mother(X,Y))$.

(3) $parent(X,Y)$:- $edb(father(X,Y))$.

(4) :- $uncle(X,Y)$.

We can obtain the following setting S by using the setting assembly algorithm.

$S \quad = \{$:- $edb(father(X,Y))$, :- $edb(mother(X,Y))$, :- $edb(brother(Z,Y))$,
:- $parent(X,Z)$, :- $uncle(X,Y)\}$

The elements, $edb(father(X,Y))$, $edb(mother(X,Y))$ and $edb(brother(Z,Y))$, indicate that instances for father, mother, and brother are tuples of the corresponding relations. Instances for $parent(X,Y)$ in (2) can be obtained by using a *projection* operation for the relation *father*. In this case, because the mapping is identical, tuples of the relation *father* become tuples of the virtual relation *parent*. Similarly, tuples of the relation *mother* becomes tuples of the virtual relation *parent*. Thus, the virtual relation parent can be obtained by a *union* operation between *father* and *mother* relations. Instances for $uncle(X,Y)$ in (1) can be obtained by using an *equi-join* operation between *parent* and *brother* relations, since the negative literals in (1) have a common variable Z. Now the virtual relation *parent* exists, so we can derive the virtual relation *uncle*. The tuples of the *uncle* relation are the desired answers.

■

4.3 Iteration

In this section, we show how setting evaluation can handle iterations and propose an extended least fixed point operation to terminate iterations.

```
begin
  initialize $R_l(0)$, $l = 1,..,n$
  $i \leftarrow 1$
  repeat
    begin
      $t \leftarrow true$
      for $j = 1$ to $n$
        begin
          enumerate $R_j(i)$ using $R_l(i\text{-}1)$, $l = 1,..,n$
          $R'_j(i) \leftarrow R_j(i)$ union $R'_j(i\text{-}1)$
          $t \leftarrow (R'_j(i) = R'_j(i\text{-}1))$ and $t$
        end
      $i \leftarrow i + 1$
    end
  until($t = true$)
end
```

Figure 4.2 The extended least fixed point operation algorithm

The extended least fixed point operation is based on the least fixed point operation used in deferred evaluation. Setting evaluation requires that the least fixed points of enumeration be checked for all setting elements. It is not enough to individually check the least fixed points of each element, because there may be elements related each other. All the checking operations must be continued until no new instances are enumerated for any element. In other words, all answers for the given goal are obtained when all complementary instances of every setting element are enumerated.

Let S be a setting and $S = \{E_1, .., E_j, .., E_n\}$. Let $R_j(i)$ be an i-th transitive relation corresponding to an element E_j. If $E_j (j = 1, .., n)$ is of the form $edb(L)$, $R'_j(0)$ can be derived from relation L. Otherwise, let $R'_j(0)$ be empty. Figure 4.2 shows the extended least fixed point operation algorithm.

Example 4.2: An intensional database (1) through (4) and a goal (5) are given.

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

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

(3) $parent(X,Y)$:- $edb(father(X,Y))$.

(4) $parent(X,Y)$:- $edb(mother(X,Y))$.

(5) :- $ancestor(X,Y)$.

The setting S is assembled as follows:

S = { :- $edb(father(X,Y))$, :- $edb(mother(X,Y))$, :- $parent(X,Y)$, :- $ancestor(X,Y)$}

The enumeration process is illustrated in Figure 4.3. As the initial state, the *father* and *mother* relations have some tuples and the *parent* and *ancestor* relations are empty. First, the *parent* relation is derived by a *union* operation between the *father* and *mother* relations.

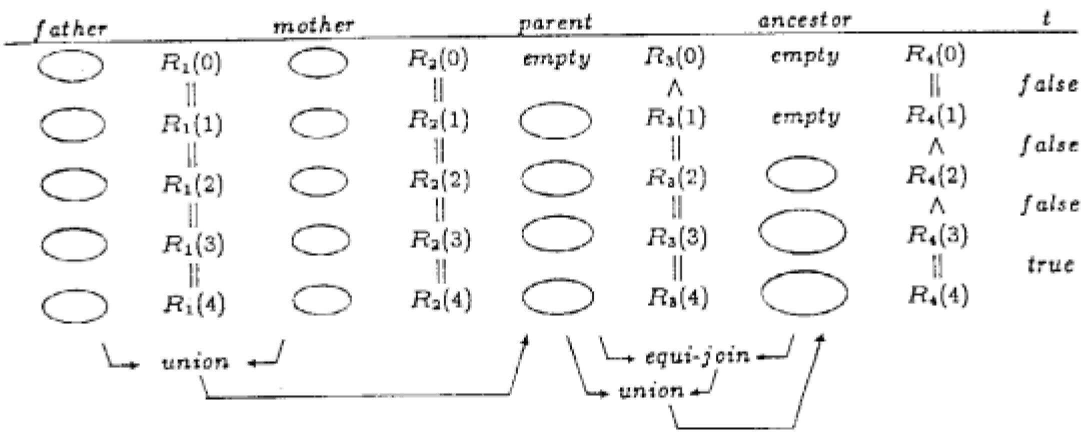

Figure 4.3 Instance enumeration process

Now, the value of t in the extended least fixed point operation algorithm (Figure 4.2) is *false*, because instances of the *parent* relation increase. Next, the *ancestor* relation is derived using a *union* operation between the *parent* and a virtual relation derived by *equi-join* between the *parent* and *ancestor* relations. These operations are continued until instances of the *ancestor* relation do not increase. When instances of the *ancestor* relation do not change, the value of t becomes *true*. Tuples of the *ancestor* relation are the desired answers.

■

Obviously, since instances of the ground unit clauses never increase, there is no need for checking operations on elements corresponding to ground unit clauses, so this can be omitted. Similarly, a checking operation for elements whose instances are enumerated merely by *union* operations between instances of ground unit clauses can also be omitted after the *union* operations are performed.

To decrease interface overhead between the relational database management system and the intensional processor, we propose to let the management system perform the extended least fixed point operation. In deferred evaluation, since the intensional processor must generate complex plans one by one, the management system cannot handle least fixed point operations competely. In setting evaluation, since the intensional processor does not have to do anything after sending the setting and relationship between these elements, the management system can handle the extended least fixed point operations.

4.4 Completeness and Termination

A deductive database system can be implemented using setting evaluation and extended least fixed point operation. We now show that this deductive database system is guaranteed to enumerate all correct answers and to terminate.

The completeness of unit resolution for Horn clauses is proved in Loveland 78. A set of Horn clauses H is divided into the intensional database, extensional database, and a given goal. The setting assembly algorithm terminates and returns a setting using only the intensional database and the goal. All literals in the intensional database that may be used in resolutions for the given goal are contained in the setting. All literals in the extensional database that may be used in the resolution are complementary instances of some members of the setting. These instances are tuples of the relation in the extensional database. We are using the notation *edb* (L) to indicate explicitly that L has instances in the extensional database. Complementary instances of other setting members are represented as tuples of virtual relations. These instances are enumerated using relational algebra operations. Since ground unit clauses do not contain variables, a unification is simply treated as an equality check or substitution. A set derived from finite sets by a finite number of applications of *equi-join* and *union* operations is finite. If H is function-free, the extended least fixed point operation can reach the fixed point in a finite number of steps.

In Section 2.2, we considered the reason why deferred evaluation cannot handle all types of recursive queries. The problem is that the plans are generated using input resolution. With the top-down approach it is difficult for the plan generator to get information about the contents and sequence of plans to obtain sufficient results. The check predicate is not completely effective. Since the setting evaluation uses a bottom-up approach, this kind of control is unnecessary. The required results are accumulated automatically without unnecessary computation.

Even if the set of definite clauses contains any types of recursive call (e.g. left recursive call, mutual recursive call, nested recursive call), the results can always be obtained by the bottom-up approach. If the clauses are function-free, the enumeration is saturated at some point in time and the system detects the precise moment. All the results have now been obtained and the system outputs them.

4.5 Efficiency

When the extensional database is very large, it is cleary appropriate for the database to be manipulated by the database management system rather than the intensional processor. Thus, there are two processing sectors and interface overhead between these two areas is one of the most important factors in the efficiency of the whole system. Interface overhead depends on the number of items and the amount of transferred data.

In the interpreter approach, the extensional database is accessed as soon as a literal corresponding to tuples of the extensional database is evaluated in the interpretation process, and each time the tuples are transferred from the database management system to the intensional processor. Then both the number of items and the amount of transferred data is enormous.

Deferred evaluation lets the intensional processor accumulate the extensional database literals until a deduction using only the intensional database is completed. A set of literals for the deduction is a plan. Several plans may be generated for a given first-order query, but the result is only transferred to the intensional processor once for that query. The number of items and the amount of transferred data is smaller than in the interpreter approach. However, if the query contains a recursive call, deferred evaluation lets the system generate a number of plans corresponding to each recursive level. In the case of simple recursive queries, like Example 2.2, if one recursive level has M alternative plans and the least fixed point of the recursion is the N-th level, M^N plans are generated.

In setting evaluation, the intensional processor sends the setting and relationship between the elements to the management system once at the start. The result is transferred once, even if the query contains recursive calls.

Moreover, we can even expect parallel execution of complementary instance enumeration processes for each setting element to be an effective option.

5. Conclusion

We presented a methodology for constructing a deductive database system consisting of an intensional processor and a relational database management system. We expected that unit resolution would be effective for obtaining all answers, but the strategy for performing unit resolution in the relational database system was not clear. We introduced the setting evaluation approach to perform unit resolution using relational algebra operations. Another big problem was the termination of recursively defined queries. We presented the extended least fixed point operation corresponding to setting evaluation. We showed that the deductive database system using setting evaluation and the extended least fixed point is guaranteed to obtain all correct answers and terminate for any recursively defined queries. Furthermore, it was shown that the setting evaluation approach is more efficient than the deferred evaluation approach and allows us to omit consideration of the termination condition of recursive calls and the sequence of clauses or literals.

ACKNOWLEDGMENTS

The authors thank Dr. D. S. Kerr of Ohio State University for his helpful advice and Dr. H. Gallaire of ECRC and Dr. K. Yazdanian of ONERA-CERT for useful discussions.

REFERENCES

1. Aho, A. V. and Ullman, J. D. Universality of Data Retrieval Languages. *Proceedings of ACM/SIGPLAN Conference on Principles of Programming Languages*, San Antonio, Jan. 1979, pp.110-117.

2. Chang, C. L. and Lee, R. C. T. *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

3. Chang, C. L. On Evaluation of Queries Containing Derived Relations in a Relational Data Base. *Advances in Data Base Theory Vol. 1*, ed. Gallaire et al., PLENUM, 1981, pp.235-260.

4. Chakravarthy, U. S., Minker, J., and Tran, D. Interfacing Predicate Logic Languages and Relational Databases. *Proceedings of the First International Logic Programming Conference*, Faculte des Sciences de Luminy Marseilles, France, Sept. 14-17th 1982, pp.91-98.

5. Gallaire, H. Logic Data Bases vs Deductive Data Bases. *Proceedings of Logic Programming Workshop*, Algrave, June 1983, pp.608-622.

6. Gallaire, H., Minker, J., and Nicolas J-M. Logic and Databases: A Deductive Approach. *Computer Surveys*, Vol. 16, No. 2, June 1984.

7. Henschen L. J., and Naqvi S. A. On Compiling Queries in Recursive First-Order Databases. *Journal of the ACM*, Vol. 31, No. 1, January 1984, pp.47-85

8. Kunifuji, S. and Yokota, H. PROLOG and Relational Data Base for Fifth Generation Computer Systems. *Proceedings of ONERA-CERT Workshop on "Logical Bases for Data Bases"*, ed. Gallaire, et al,, ONERA-CERT, Toulouse, Dec. 1982.

9. Loveland, D. W. *Automated Theorem Proving: A Logical Basis*, North-Holland, 1978.

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

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

12. Okuno, H. The Execution Mechanism for Logic Programming Language to Perform an Efficient retrieval of All Solutions. *Proceedings of the Logic Programming Conference '83*, March 1983.

13. Reiter, R. Deductive Question-Answering on Relational Data Bases. *Logic and Data Base*, ed. Gallaire and Minker, PLENUM, 1978, pp.149-177.

14. Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, Vol. 12, No. 1, January 1965, pp23-41.

15. Ullman, J. D. *Principles of Database Systems*, 2nd ed. Computer Science Press, Potomac, Md., 1982.

16. Ullman, J. D. Implementation of Logical Query Languages for Databases. Technical Report of Stanford University, STAN-CS-84-1000, May 1984.

17. Yokota, H. et al., An Enhanced Inference Mechanism for Generating Relational Algebra Queries, *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, April, 1984, pp.229-238.