TR-716

PHI: A Deductive Database System

by

H. Haniuda, Y. Abiru & N. Miyazaki (Oki)

December, 1991

PHI: A Deductive Database System*

Hiromi Haniuda†, Yukihiro Abiru‡ and Nobuyoshi Miyazaki†

† Oki Electric Industry Co., Ltd.
4-11-22 Shibaura, Minato-ku, Tokyo 108, Japan.

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

Abstract

PHI is an experimental distributed deductive database system developed in the Fifth Generation Computer Systems Project in Japan. It is implemented on Personal Sequential Inference Machines (PSI) also developed in the project. The design of PHI started in 1985 and it has been operational since 1988.

A deductive database consists of intensional and extensional databases. PHI is composed of two layers which handle intensional and extensional databases respectively. Query language of PHI is Datalog and its query processing is performed in a bottom-up manner. Query optimization of PHI is the combination of query transformation and techniques developed for relational databases.

This paper discusses the design, implementation and evaluation of the deductive database part of PHI.

1 Introduction

In the past decade, the deductive database has become one of the most attractive technologies in the database field as a key to realize new systems such as knowledge information processing systems or next generation database systems. Traditional databases such as relational databases are not powerful enough to realize these new systems. The deductive database is attractive because of its declarativeness and theoretical concreteness.

The Fifth Generation Computer Systems (FGCS) project in Japan aims to develop a prototype system for a knowledge information processing system. The aim of the knowledge base research in the project is to realize a subsystem in the logic programming paradigm to manage large shared knowledge bases. Coordination of various knowledge bases and processing knowledge bases in a distributed environment is important for future information processing systems. One of the most fundamental issues in the research is the knowledge base model as a framework. We have selected a deductive database as a fundamental platform to study knowledge bases in distributed environment.

PHI is an experimental distributed deductive database system developed in the FGCS project, and is implemented on Personal Sequential Inference Machines (PSI) also developed in the project. The design of PHI started in 1985 and it has been operational since 1988. The overall view of PHI and the knowledge base research in the FGCS project can be found in [WM+88,IM+88]. In this paper, we concentrate on the deductive database part of PHI.

The deductive database is a natural extension of the relational database. Although non-recursive queries can be processed by algebraic operations in relational database systems, recursive queries cannot be processed by RDBs. Thus the central issues of query processing in PHI are how to deal with recursions and how to use algebraic operations of relational database systems to improve the system performance.

In this paper, we discuss the overview of the system in section 2, the query interface in section 3, and the query processing strategies in section 4. Finally we discuss the results of evaluation of PHI.

2 Overview of the System

A deductive database is a set of clauses, which consists of an intensional database (IDB) containing rules and an extensional database (EDB) containing facts. We assume that the EDB is much larger than the IDB. There is a well known one-to-one correspondence between a fact of a logic program and a tuple of a

* An earlier version of this paper appears in the proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 1991.

relation. Set oriented relational database (RDB) operations are very powerful for dealing with a large amount of data. Thus RDB operations must be used to realize the deductive inference in order to improve the overall performance.

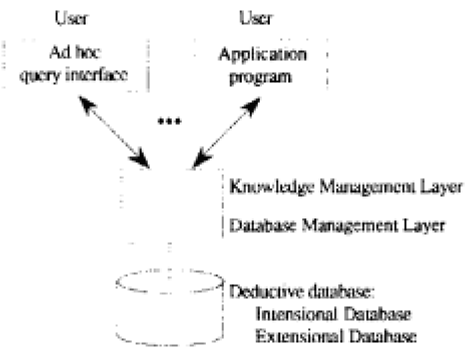

Figure 1 Logical configuration of PHI.

The query processing of PHI consists of two phases. The first is the processing of the IDB related to the query. This phase includes the extraction of related rules from the IDB and the compilation of the goal and rules to extended-relational algebra which includes ordinary relational operations and others handling iterative computation. The second phase is the execution of the produced operations on the EDB to compute the answer. The detail of query processing strategy performed in PHI is presented in section 4.

The principal part of PHI consists of two layers which are designed to process the above two phases. Figure 1 shows the logical configuration of PHI. The upper layer is the Knowledge Management Layer (KML) which performs the first phase of query processing. The lower is the Database Management Layer (DBML) which performs the second phase and is essentially a relational database management system with some extensions for the iterative processing of RDB operations.

3 Query Language and Interface

The query language of PHI is Prolog without function symbols, that is Datalog with negation. A query consists of a goal and rules. The answer set of a query to a deductive database is defined as follows. Let *goal'* be a ground instance of *goal*. Then, the answer set of the query is the set $G=\{goal' | IDB \cup EDB \cup R \Rightarrow goal'\}$ where R is a set of rules and *goal* is the goal atom in the query, and $\Rightarrow$ means logical consequence.

PHI computes the set of answers which satisfies the query condition in the same way as relational database systems. It can process ad hoc queries through its query interface (see Figure 1). And it also has an embedded interface in Extended Self-contained Prolog (ESP)[C84], which is the system programming language of PSI based on object-oriented and logic paradigms. Because both Datalog and ESP use Horn clauses, programmers do not have to use completely different languages. PHI returns an answer for the query by unifying contents in the answer set to variables in the goal. Alternative answers will be returned if backtracks occur in ESP.

We show below an example of a query embedded in ESP. The interface querying PHI from ESP is realized by a method *:refute* which is a message passed to PHI in the object-oriented paradigm.

```
:refute(PHI,commonAncestor(CommAnc,myName,yourName),
    (commonAncestor(CA,Name1,Name2) :-ancestor(CA,Name1),ancestor(CA,Name2)))
```

where the first argument is the object corresponding to PHI itself, the second is a goal atom and the third is a rule which defines common ancestors given names of two persons (variables begin with a capital letter and constants begin with a small letter). In this example it is assumed that the rules and facts to compute ancestors are in the deductive database.

At first, when PHI receives the message, it computes the answer set, virtual relation *commonAncestor*, and it returns an answer such as *CommAnc=hanako*. Then if a backtrack occurs in the application program, PHI returns the next answer *CommAnc= ume*, and so on.

4 Query Processing in PHI

As mentioned above, the main issue of query processing in PHI is how to process recursions. There are two major approaches to recursive query processing: the top-down approach based on procedural semantics and the bottom-up approach based on fixpoint semantics[BR88,L87].

The bottom-up approach computes the least fixpoint of the database corresponding to a query by iteration of RDB operations. This method is well suited to applying the techniques developed for relational databases. However its overall performance may not be good because of two problems:

(1) The iterative procedure which computes the least fixpoint involves a lot of redundant computations, which yield a lot of duplications.
(2) It computes unnecessary results because it computes all elements of the least fixpoint, that is least Herbrand model, of the database.

Several methods have been proposed to improve the performance of the bottom-up methods. There are two main ways for improving the performance: making the execution algorithm more efficient, and rewriting of rules before execution. Methods such as the semi-naive and differential methods[BR87,B85] contribute to decreasing the duplications. Rule rewriting methods such as magic sets[BMSU86], counting and reverse counting[BMSU86], Alexander[RLK86], Kifer-Lozinskii[KL86], generalized magic sets[BeR87], generalized counting[BeR87, SZ86], and magic counting[SZ87] contribute to making the model smaller.

Query processing of PHI is performed by the bottom-up method to utilize techniques developed for RDBs. Fixpoint computation is performed by the semi-naive method for linear queries and the naive method for non-linear queries. Its query optimization is realized by query transformation in KML, by ordering of operations in KML and DBML, and by traditional RDB techniques in DBML. Query transformation is a rule rewriting method which transforms queries to other forms that have smaller least fixpoints while preserving the equivalence of answers. Although execution order of RDB operations such as joins is usually decided in DBML, KML can specify part of the execution order so as to realize optimization similar to the magic sets.

Overall query processing of PHI is performed as follows:

(1) The part of the IDB related to a query is merged into rules in the query.
(2) The query is transformed for optimization.
(3) The transformed query is compiled to extened-relational algebraic commands.
(4) The resulting commands are executed to compute the least fixpoint of the query and the EDB.

The least fixpoint is obtained by iterative computation of algebraic commands. This computation grows virtual relations corresponding to recursive atoms and it terminates when all virtual relations do not change. In order to realize that iterative computation, we augmented KBML to have additional commands such as iteration and set comparison.

4.1 Query Transformation

The role of the query transformation is to obtain a query form that can be executed by subsequent execution algorithms more efficiently than the original form.

Three types of query transformations called Horn Clause Transformations (HCT)[MHYI89] are used to transform a query (a goal and rules) to an equivalent form in PHI. A transformed set of clauses can be evaluated more efficiently, because it has a smaller least fixed point than the original. The first transformation, HCT by partial evaluation, simplifies queries by eliminating some predicate symbols using resolution. The second, HCT by substitution, is a generalization of the distribution of selections[AU79]. The third, HCT by restrictor[MYHI89], is similar to the magic sets method.

Since the output of these transformations are sets of clauses, these transformations can be used together and can be combined with any other methods including the top-down method.

In the following sections, goals are denoted like "*?-goal* ", variables begin with a capital letter and constants begin with a small letter.

4.2 HCT/P (HCT by Partial Evaluation)

This transformation is a procedure that uses resolution to obtain a more efficient query. This procedure is regarded as a generalization of a procedure that substitutes the relational algebra expression of a derived relation for the relation symbol. It is called HCT/P because it is based on the partial evaluation technique developed for program transformation. For example, a query

```
?-p(X,Y).
p(X,Y) :- p1(X,Y).
p(X,Y) :- q(X,Z),p2(Z,Y).
q(X,Y) :- p(X,Y).
```

is transformed to an equivalent query

```
?-p(X,Y).
p(X,Y) :- p1(X,Y).
p(X,Y) :- p(X,Z),p2(Z,Y).
```

In this example the third rule defining a virtual relation *q* is eliminated.

4.3 HCT/S (HCT by Substitution)

This transformation is a procedure that substitutes ground terms for variables of a clause to obtain an equivalent query. It can be regarded as a generalization of procedures that move the constant in transitive closure operation. Next is an example of HCT/S.

```
?-commonAncestor(A,tarô,jirô).
commonAncestor(A,B,C) :- ancestor(A,B),ancestor(A,C)
ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(A,C),ancestor(C,B).
```

is transformed to

```
?-commonAncestor(A,tarô,jirô).
commonAncestor(A,tarô,jirô) :-ancestor(A,tarô),ancestor(A,jirô).
ancestor(A,tarô) :- parent(A,tarô).
ancestor(A,tarô) :- parent(A,C),ancestor(C,tarô).
ancestor(A,jirô) :- parent(A,jirô).
ancestor(A,jirô) :- parent(A,C),ancestor(C,jirô).
```

HCT/S is always possible for non-recursive queries, but it may not be possible for certain recursive queries. For example, let us suppose the goal of above example is *?-commonAncestor (hanako,tarô,jirô)*. At first the procedure tries to substitute the constant *hanako* for the first argument of *commonAncestor* rule and it succeeds. Next it tries to substitute the constant for the first arguments of *ancestor* rules but it fails because this substitution does not preserve the equivalence of clauses. Thus the constant *hanako* is not substituted in *ancestor* rules. In other words a selection to the first argument of the virtual relation *commonAncestor* is not distributed to the virtual relation *ancestor*. A more general procedure is discussed in [M90].

4.4 HCT/R (HCT by Restrictor)

HCT/S fails to optimize certain queries as mentioned in previous section. HCT/R is based on subsumption and can optimize this kind of query. HCT/S represents selections by constants (or identical variables in different arguments) in atoms. There is another way to represent selections, that is selections represented by separate predicates. We introduce new predicates called restrictors that correspond to selections. This transformation corresponds to a generalization of magic sets and its variations[MYHI89].

Suppose we have two clauses in the database:

p :- $q,\ldots,r$.
p :- $p^*,q,\ldots,r$.

where arguments of atoms are omitted in behalf of explanation.

Since the first clause subsumes the second, the latter is a logical consequence of the former. Hence, a transformation that replaces the first clause with the second can be formulated. HCT/R is a procedure that, given a set of predicates S, introduces a new restrictor predicate r^* for each predicate r in S. We show an example of HCT/R below.

Suppose we have a same generation query:

?-$sg(c,X)$.
$sg(X,X)$.
$sg(X,Y)$:- $p(X,A),p(Y,B),sg(B,A)$.

Although HCT/S can not distribute the constant c in the goal to the same generation rules (sg), HCT/R can transform this query to

?-$sg(c,X)$.
$g^{*bf}(c)$.
$sg^{*bf}(B)$:- $sg^{*fb}(Y),p(Y,B)$.
$sg^{*fb}(A)$:- $sg^{*bf}(X,),p(X,A)$.
$sg(X,X)$:- $sg^{*bf}(X)$.
$sg(X,X)$:- $sg^{*fb}(X)$.
$sg(X,Y)$:- $sg^{*bf}(X),p(X,A),p(Y,B),sg(B,A)$.
$sg(X,Y)$:- $sg^{*fb}(Y),p(X,A),p(Y,B),sg(B,A)$.

Atoms with a superscripted asterisk are restrictors which play the role of selections to succeeding atoms (relations). b/fs in restrictors are called adornments and are introduced to eliminate variables which do not contribute to making the model smaller. A fact as a restrictor is called an initial restrictor clause ($sg^{*bf}(c)$ in the above example), and can be regarded as a seed of selections. In the above example, sg^{*bf} and sg^{*fb} simulate selections on the first and second arguments of relation p, and they grow from the seed $sg^{*bf}(c)$.

HCT/R works correctly for any type of queries even if it is mutually recursive. For example, suppose the query is

?-$p(c,X)$.
$p(X,Y)$:- $a(X,Y)$.
$p(X,Y)$:- $a(X,A),p(A,B),q(B,Y)$.
$q(X,Y)$:- $b(X,Y)$.
$q(X,Y)$:- $p(X,A),c(A,B),q(B,Y)$.

HCT/R transforms this query to

?-$p(c,X)$.
$p^{*bf}(c,X)$.
$p(X,Y)$:- $p^{*bf}(X),a(X,Y)$.
$p(X,Y)$:- $p^{*bf}(X),a(X,A),p(A,B),q(B,Y)$.
$p^{*bf}(A)$:- $p^{*bf}(X),a(X,A)$.
$q^{*bf}(B)$:- $p^{*bf}(X),a(X,A),p(A,B)$.
$q(X,Y)$:- $q^{*bf}(X),b(X,Y)$.
$q(X,Y)$:- $q^{*bf}(X),p(X,A),c(A,B),q(B,Y)$.
$p^{*bf}(X)$:- $q^{*bf}(X)$.
$q^{*bf}(B)$:- $q^{*bf}(X),p(X,A),c(A,B)$.

In this example, two restrictors p^{*bf} and q^{*bf} are introduced.

Ordinarily, initial restrictor clauses cannot be compiled to algebraic operations. So we add a DBML command generating an intermediate relation corresponding to an initial clause, which can be embedded in the sequence of algebraic commands.

The transformed query which includes restrictors is compiled to the sequence of algebraic and other additional commands of DBML, and then the sequence is optimized by DBML in the same way as in RDBs. If restrictors are treated as ordinary relations in optimization in DBML, the resulting order of operations might not be optimal. Thus we designed PHI so that KML could partially control the ordering of operations which include a restrictor as an operand.

5 Evaluation

Performance of query processing in PHI has been evaluated for a number of queries. In this section, we present the results of a comparative performance evaluation of the query processing for three different versions of *ancestor* queries.

Descendant:
?-ancestor(c,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

Linear ancestor:
?-ancestor(X,c).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

Non-linear ancestor:
?-ancestor(X,c).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).

We assume the rules which define *parent*:

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

and facts *father* and *mother* are in the deductive database.

Tuples in *father* and *mother* relations form a cylinder structure similar to the evaluation in [BR88]. We vary the total size of these relations from 2^7 to 2^{13} tuples. The constant *c* in the goal is chosen so that it corresponds to the node at the middle height of the structure. Thus the size of the answer depends on height of the structure. We assume the total size of the relations to be proportional to 2^{height}.

In Figure 2, for each query and query transformation, we plot the response time against the size of the data. The total response time for a query includes processing time for query transformations, compilation to relational operations and fixpoint computation. In the following, we refer to each curve by an abbreviation such as nl-PS defined in Figure 2.

It can be seen that the bottom-up strategy without optimization is very inefficient (see the top two curves). Neither HCT/P, S, SP (composition of S and P) for descendant and non-linear ancestor queries nor HCT/P for linear ancestor query improve the performance, since the constant in a goal is not used in these cases. The curve l-S shows that HCT/S improve the performance of the linear ancestor query. However its improvement is insufficient, because the constant *c* is not entirely distributed to all facts. This insufficiency is resolved by applying HCT/P to the query before HCT/S (see l-PS).

Another observation is that HCT/R gains certain performance improvements for all queries (see l-R, d-R, nl-R). Since HCT/R introduces new virtual relations, it has a certain overhead. However this overhead can be reduced if HCT/R is combined

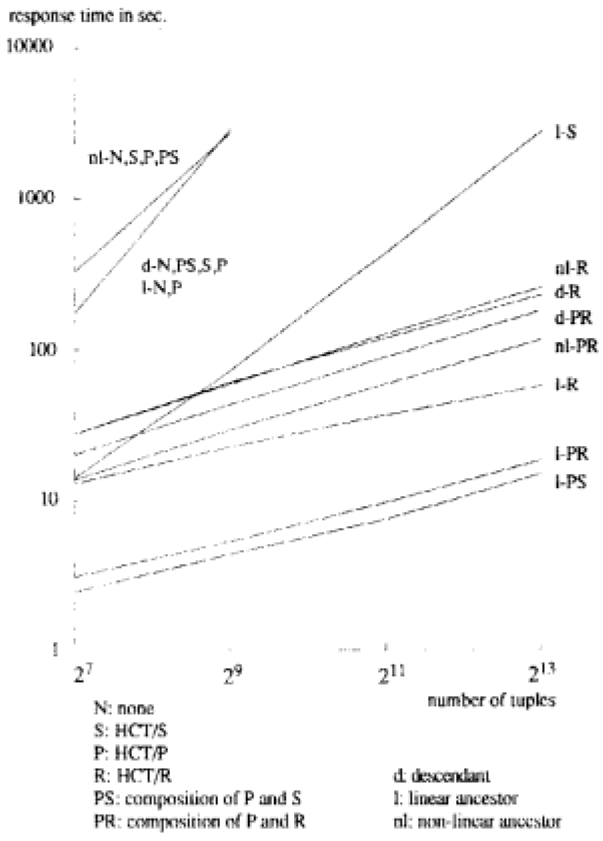

Figure 2 Query processing performance.

with HCT/P (see l-PR, d-PR, nl-PR). The difference between curves l-PR and l-PS corresponds to the overhead based on introducing a restrictor by HCT/R and it is very small.

The results of measurements are similar to those of analytical methods[BR88] except for nl-PR. Although the performance of d-PR is analytically better than nl-PR, the result is the opposite. In this case, nl-PR shows better performance because it converges more rapidly than d-PR and the number of iteration is more dominant than the size of the model. From this, we can see that set oriented operations of RDB operate efficiently in the bottom-up method.

6 Conclusions

In this paper, we discussed the design, implementation and evaluation of the deductive database part of PHI. In PHI, relational database technologies are utilized to realize a deductive database system, and its query optimization is performed by query transformations and techniques developed for RDBs. As the result of the performance evaluation, our approach to deductive databases arrives at two points:

(1) RDB technologies are powerful for realizing deductive databases,
(2) HCTs contribute to the improvement of system performance.

We are now designing and implementing a new experimental object-oriented deductive database system which can handle composite objects[MHY90].

References

[AU79] Aho, A. V. and Ullman, J.D., "Universality of Data Retrieval Languages," *Proc. 6th ACM POPL*, 1979, pp.110-120.

[B85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," *On Knowledge Base Management Systems*, edited by Brodie. et al., Springer, 1985, pp.165-178.

[BeR87] Beeri, C. and Ramakrishnan, R., "On the Power of Magic," *Proc. 6th ACM PODS*, 1987, pp.269-283.

[BMSU86] Bancilhon, F., et al., "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. 5th ACM PODS*, 1986, pp.1-15.

[BR87] Balbin, I. and Ramamohanarao, K. A., "A Generalization of Differential Approach to Recursive Query Evluation," *Journal of Logic Programming*, Vol. 4, No. 3, 1987, pp. 59-262.

[BR88] Bancilhon, F. and Ramakrishnan, R. "Performance Evaluation of Data Intensive Logic Programs," in *Foundations of Deductive Databases and Logic Programming*, 1988, pp.439-517.

[C84] Chikayama, T., "Unique Features of ESP," *Proc. FGCS*, 1984, pp.292-298.

[IM+88] Itoh, H., et. al., "Knowledge Base System in Logic Programming Paradigm," *Proc. FGCS*, 1988, pp.37-53.

[KL86] Kifer, M. and Lozinskii, E. L., "Filtering Data Flow in Deductive Databases," *Proc. ICDT*, 1986, pp.186-202.

[L87] Lloyd, J. W., *Foundations of Logic Programming*, Second edition, Springer-Verlag, 1987.

[M90] Miyazaki, N., "Selection propagation in deductive databases -From pushing selections to magic sets-," *Data & Knowledge Engineering Vol.5, No.4*, 1990.

[MHY90] Morita, Y., Haniuda, H. and Yokota, K., "Object Identity in *Quixote*," *SIGDBS & SIGAI of IPSJ*, Nov. 1990.

[MHYI89] Miyazaki, N., et al., "A Framework for Query Transformations in Deductive Databases," *Journal of Information Processing*, Vol.12, No.4, 1989, pp.351-361.

[MYHI89] Miyazaki, N., et al., "Horn Clause Transformation by Restrictor in Deductive Databases," *Journal of Information Processing*, Vol.12, No.3, 1989, pp.266-279.

[RLK86] Rohmer, J., et al., "The Alexander Method," *New Generation Computing*, Vol. 4, No. 3, 1986, pp.273-286.

[SZ86] Sacca, D. and Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," *Proc. ICDT*, 1986, pp.31-53.

[SZ87] Sacca, D. and Zaniolo, C., "Magic Counting Method," *Proc. 6th ACM SIGMOD*, 1987, pp.49-59.

[WM+88] Wada, M., et. al., "A Superimposed Code Scheme for Deductive Databases," in *Database Machines and Knowledge Base Machines*, Kluwer Academic Publishers, 1988, pp.571-584.