New Generation Computing, 6 (1988) 249-258
OHMSHA, LTD. and Springer-Verlag

© OHMSHA, LTD. 1988

# Partial Evaluation of Queries in Deductive Databases

Chiaki SAKAMA and Hidenori ITOH
*Institute for New Generation Computer Technology,*
*Mita Kokusai Building 21F,*
*1-4-28 Mita, Minato-ku, Tohyo, Japan.*

Received 17 March 1988

***Abstract*** This paper presents some applications of partial evaluation method to a query optimization in deductive database. A Horn clause transformation is used for the partial evaluation of a query in an intensional database, and its application to multiple query processing is discussed. Three strategies are presented for the compatible case, ordered case and crossed case. In each case, partial evaluation is used to preprocess the intensional database in order to obtain subqueries which direct access to an extensional database.

**Keywords:** Partial Evaluation, Deductive Database, Query Optimization, Horn Clause Transformation, Multiple Query Processing.

## §1 Introduction

A deductive database usually consists of a large extensional database (*edb*) and a comparatively small intensional database (*idb*). The *edb* is a set of database relations which are explicitly stored, while the *idb* is a set of function free Horn clauses which define virtual relations between tuples.

In such deductive database systems, a query is evaluated with an inference system in the *idb* and with a relational database system in the *edb*. There are known two methods to perform this evaluation: the *interpretive method* and the *compiled mothod*.[4] The interpretive method evaluates a query tuple at a time in the *idb* and *edb* until the query is wholly evaluated, while the compiled method compiles a query wholly in advance in the *idb* and produces a set of subqueries evaluable only in the *edb*. When there is a large extensional database, the compiled approach is considered more effective since it partially evaluates a query in the *idb* at first, then can reduce the access cost to the *edb*.[6,8]

Partial evaluation[3] of a program is considered as a specialization technique for runtime environment and is useful for various application in practice. The compiled method in a deductive database is considered as an application of

a partial evaluation technique to a query optimization.

In this paper, some partial evaluation methods for the query optimization is discussed. Section 2 presents the partial evaluation method for query processing in deductive databases, and Section 3 discusses some application to multiple query processing. Section 4 shows some experimental results of performance evaluation.

## §2 Partial Evaluation of a Query

Partial evaluation of a query in a deductive database is defined as a transformation of the query in the *idb*. It is an unfolding of a query which includes virtual relations defined in the *idb* into subqueries which includes only database relations defined in the *edb*.

This process is represented as follows:

$$Q_{idb} = \tau(idb, Q)$$

where $Q$ denotes a query to be evaluated, $\tau(idb, Q)$ denotes an unfolding transformation of a query $Q$ in the *idb*, and $Q_{idb}$ presents the transformed result which is the subquery for the *edb*.

After this transformation, $Q_{idb}$ is evaluated in the *edb*:

$$Q_{idb,edb} = \varepsilon(edb, Q_{idb})$$

where $\varepsilon(edb, Q_{idb})$ denotes the evaluation of $Q_{idb}$ in the *edb* with relational operations and $Q_{idb,edb}$ presents the result of the query evaluation in the *idb* and *edb*, which is a set of answer tuples for the query.

In such a partial evaluation, the problem happens when there is a recursive clause in the *idb*. In such a case, unfolding transformation of a query which includes recursion never terminates in database relations. A lot of algorithms for a recusive query processing which assure termination condition and answer completeness have been studied,[1] and in this paper, *Horn Clause Transformation* (*HCT*) procedure[5] is used for treating such recursive queries.

The *HCT* procedure is as follows.

(1) Predicates defined in the *edb* are extensional predicates.
(2) Predicates which appear twice at first during the unfolding of a given query are recursive predicates.
(3) Select all clauses in the *idb* whose head predicate is the same with a given query or recursive predicates, then unfold their bodies until they contain only extensional predicates or recursive predicates

In this way, the *HCT* procedure transforms an *idb* into the equivalent set of clauses for a given query.

**Example 2.1**

Suppose the following *idb*.

$p(X, Y) :- q(X, Z), r(Z, Y).$
$q(X, Y) :- s(X, Y).$
$q(X, Y) :- s(X, Z), t(Z, Y).$
$t(X, Y) :- u(X, Z), q(Y, Z).$
$v(X, Y) :- w(Y, X).$

Using the *HCT* procedure, a query $? - p(a, Y)$ is partially evaluated in the *idb* as:

$p(X, Y) :- q(X, Z), r(Z, Y).$
$q(X, Y) :- s(X, Y).$
$q(X, Y) :- s(X, Z), u(Z, W), q(Y, W).$

These are the clauses which contain subqueries to be evaluated in the *edb*, where $q$ is a recursive predicate and $r$, $s$ and $u$ are extensional predicates.

After the transformation, the following relational commands are generated:

$$p_2 = \pi_2(\sigma_{1=a}(q \bowtie_{2=1} r)),$$
$$q = s \cup (s \bowtie_{2=1} u \bowtie_{2=2} q)$$

where $\pi$, $\sigma$, $\bowtie$ and $\cup$ denote projection, selection, join and union, respectively. And the subscript number denotes the position of arguments in relation. These relational commands are executed iteratively by a relational database system and terminated in the least fixed points. □

If a query contains some constants which give a condition of the alternative choice of clauses in an *idb*, then unnecessary expansion is avoided by propagating binding information of the query to those clauses. In this case, however, recursive clauses cannot be bound for the repeated usage in general.

## §3 Multiple Query Processing

When there is a set of queries to be evaluated, however, it is inefficient to evaluate each query independently in an *idb*. To minimize the cost of such multiple query processing in deductive database, it is effective to perform evaluation once that is common to some of the queries, and use the common intermediate results to obtain answers for those queries. For example, it is achieved in a non-procedural way by using a connection graph,[2] that is, grouping a set of queries, exploiting the common subexpressions by some heuristics, and generating a single plan to evaluate these queries.

In this section, application of the partial evaluation method to multiple query processing is discussed. In all subsequent discussion, a query is assumed to be composed of a single atom (atomic formula). That is, a query composed of several atoms such as, $? - p(X, Y), q(Y, Z)$, is considered as a query $? - r(X, Y, Z)$ and a clause $r(X, Y, Z) :- p(X, Y), q(Y, Z)$.

### 3.1 Compatible Case

First, it is discussed the case where some queries are *compatible*.

**Definition 3.1**

(1) Queries which have the same predicate and the same number of arguments are called *compatible*.
(2) For a query $Q$, a query $GQ$ which is compatible with $Q$ but has different variables in each argument is called *general form* for it. □

**Example 3.1**

For the compatible queries $\{p(a, Y), p(X, b)\}$, $p(X, Y)$ is a general form for them. □

Suppose a set of compatible queries, then by using their general form, unfolding transformation for them is achieved once in an *idb* as follows:

$$\mathbf{Q}_{idb} = \tau(idb, GQ)$$

where $\mathbf{Q}$ is a set of compatible queries and $GQ$ denotes their general form.

Then $\mathbf{Q}_{idb}$ is evaluated in an *edb* with the selection condition of the given queries:

$$\mathbf{Q}_{idb,edb} = \varepsilon(edb, \sigma_Q(\mathbf{Q}_{idb}))$$

where $\sigma_Q$ denotes a selection under the condition of $\mathbf{Q}$, and $\mathbf{Q}_{idb,edb}$ presents a set of answer tuples for the compatible queries.

**Example 3.2**

Suppose a set of compatible queries: $\mathbf{Q}=\{p(a, Y), p(X, b)\}$ for the *idb* in Example 2.1. Then, $GQ=p(X, Y)$ and it is partially evaluated in the *idb* just the same as Example 2.1. The results are evaluated in the *edb* with the following selection conditions, $\sigma_{1=a \vee 2=b}\, p$. □

When a set of queries contains a number of groups of compatible queries, they are classified into maximal subsets of compatible queries at first, then each general form is evaluated in an *idb*.

### 3.2 Ordered Case

Next, it is discussed the case where some queries are *ordered*.

**Definition 3.2**

A partial ordering over predicates is defined as follows.

(1) Suppose a clause $c$ in an *idb*. If a predicate $p$ appears in the head of $c$ and a predicate $q$ appears in the body of $c$, then $p$ is *higher* than $q$ (or $q$ is *lower* than $p$), and weitten $p \geq q$.
(2) If $p \geq q$ and $q \geq p$, then $p \sim q$. □

In the above definition, it was assumed that atoms with the same predi-

cate are compatible.

**Example 3.3**
Suppose the following *idb*,

$$p(X, Y) :- q(X, Z), p(Z, Y).$$
$$q(X, Y) :- r(X, Z), s(Z, Y).$$
$$s(X, Y) :- q(X, Y).$$

then $p \geq q \geq r$ and $q \sim s$. □

**Definition 3.3**
Queries are called *ordered* iff they have some ordering over their predicates in an *idb*. □

**Example 3.4**
Consider the set of queries $\mathbf{Q}=\{p(a, Y), q(X, b), s(X, Y)\}$ for the *idb* in Example 3.3. Then $p(a, Y)$ and $q(X, b)$ are ordered. □

In the following, the notation $\geq$ is also used to denote the ordering over queries, if there is no confusion.

When a set of queries contains such ordered queries, they should be partially evaluated from the lower ones to the higher ones. It is because a higher perdicate is defined by lower predicates in an *idb*, then unfolding of the higher query can use the unfolded results of the lower queries. In this case, however, the higher query requires *whole* evaluation of the lower predicates in general.

**Example 3.5**
Consider the ordered queries $\{p(X, Y), q(a, Y), q(X, b)\}$, where $p(X, Y) :- q(X, Y)$ is in an *idb*. Then $p(X, Y)$ requires the results of the evaluation of $q(X, Y)$, more than $q(a, Y)$ or $q(\mathrm{X}, b)$. □

Therefore, unfolding of the lower queries is performed using their general form.

Suppose a set of ordered queries $\mathbf{Q}$, and their general form $GQ_i (GQ_i \geq GQ_{i-1})$. Then they are unfolded stepwise as follows:

$$\mathbf{Q}_{idb} = \bigcup_i \tau(idb_i, GQ_i)$$

where $idb_i = \tau(idb_{i-1}, GQ_{i-1}) \cup idb^*_{i-1} (i > 1)$, $idb_1 = idb$, and $idb^*_{i-1}$ denotes the $idb_{i-1}$ except the clauses which have the same predicates with the results of the $\tau(idb_{i-1}, GQ_{i-1})$ in the heads.

This presents $GQ_i$ is partially evaluated in the *idb* using the unfolded lower $GQ_{i-1}$, and the result is a union of each evaluation. (In case of $GQ_{i-1} \sim GQ_i$, either of them can be evaluated firstly.) After that, $\mathbf{Q}_{idb}$ is evaluated in an *edb* with the selection conditions of the given queries.

**Example 3.6**
Suppose the following *idb*:

$$p(X, Y) :- q(X, Z), r(Z, Y).$$
$$q(X, Y) :- s(X, Y).$$
$$q(X, Y) :- s(X, Z), t(Z, Y).$$
$$r(X, Y) :- t(X, Z), v(Z, Y).$$
$$t(X, Y) :- u(X, Z), q(Y, Z).$$

and a set of ordered queries, $\mathbf{Q}=\{p(a, Y), q(X, b), r(X, c)\}$ where $p \geq r \geq q$.

Then $q(X, Y)$ is unfolded in the $idb_1 (= idb)$ at first:

$$q(X, Y) :- s(X, Y).$$
$$q(X, Y) :- s(X, Z), u(Z, W), q(Y, W).$$

and the *idb* is transformed into the following $idb_2$ with these evaluated results:

$$p(X, Y) :- q(X, Z), r(Z, Y).$$
$$q(X, Y) :- s(X, Y).$$
$$q(X, Y) :- s(X, Z), u(Z, W), q(Y, W).$$
$$r(X, Y) :- t(X, Z), v(Z, Y).$$
$$t(X, Y) :- u(X, Z), q(Y, Z).$$

Secondly, $r(X, Y)$ is unfolded in this $idb_2$.

$$r(X, Y) :- u(X, W), q(Z, W), v(Z, Y).$$

where $q$ is a recursive predicate and is not unfolded.
Then the $idb_2$ is transformed into the following $idb_3$ with this result:

$$p(X, Y) :- q(X, Z), r(Z, Y).$$
$$q(X, Y) :- s(X, Y).$$
$$q(X, Y) :- s(X, Z), u(Z, W), q(Y, W).$$
$$r(X, Y) :- u(X, W), q(Z, W), v(Z, Y).$$
$$t(X, Y) :- u(X, Z), q(Y, Z).$$

Finally, $p(X, Y)$ is unfolded in this $idb_3$.

$$p(X, Y) :- q(X, Z), u(Z, W), q(L, W), v(L, Y).$$

Now the partially evaluated queries in the *idb* are obtained by the union of these results.

$$p(X, Y) :- q(X, Z), u(Z, W), q(L, W), v(L, Y).$$
$$q(X, Y) :- s(X, Y).$$
$$q(X, Y) :- s(X, Z), u(Z, W), q(Y, W).$$
$$r(X, Y) :- u(X, W), q(Z, W), v(Z, Y).$$

These results are evaluated in the *edb* with the following selection conditions, $\sigma_{1=a} p$, $\sigma_{2=b} q$ and $\sigma_{2=c} r$. □

### 3.3 Crossed Case

In the last place, it is discussed the case where some queries are *crossed*.

**Definition 3.4**

Queries are called *crossed* iff they have common subqueries. □

**Example 3.7**

Consider the queries $\{p(a, Y), q(a, Y)\}$ for the following *idb*:

$$p(X, Y) :- r(X, Y).$$
$$q(X, Y) :- r(X, Y).$$
$$r(X, Y) :- s(X, Y).$$

then they are crossed since they have the common subquery $r(a, Y)$. □

For these crossed queries, their common subqueries have only to be evaluated once for them. To know that queries have common subqueries, the ordering over predicate in an *idb* is used again. In the above example, the ordering is defined as $p \geq r \geq s$ and $q \geq r \geq s$ then it is easily known that both $p(X, Y)$ and $q(X, Y)$ have a common subquery $r(X, Y)$. ($s(X,Y)$ is also a common subquery, but the first *crossed* one is considered.)

Suppose a set of crossed queries **Q**, and their common subqueries *SubQ*. Then they are partially evaluated as follows:

$$\mathbf{Q}_{idb} = \bigcup_i \tau(idb', GQ_i)$$

where $GQ_i$ denotes the general form of the given queries and *idb'* is the results of the partial evaluation of *SubQ* in an *idb*. That is, $idb' = \tau(idb, GSubQ) \cup idb^*$, where *GSubQ* is a general form of *SubQ* and *idb** denotes the *idb* except the clauses which have the same predicates with the results of the $\tau(idb, GSubQ)$ in the heads.

This presents $GQ_i$ is partially evaluated in the *idb* using the unfolded common subqueries *GSubQ*, and the result is a union of each evaluation. After that, $\mathbf{Q}_{idb}$ is evaluated in an *edb* with the selection conditions of the given queries.

**Example 3.8**

Suppose the following *idb*:

$$p(X,Y) :- r(X, Z), p(Z, Y).$$
$$p(X, Y) :- r(X, Y).$$
$$q(X, Y) :- r(X, Z), s(Y, Z).$$
$$r(X, Y) :- t(X, Z), s(Z, Y).$$
$$t(X, Y) :- s(X, Y), u(Y, X).$$

where $p, q \geq r \geq t \geq s, u$. For a set of crossed queries, $\mathbf{Q} = \{p(a, X), q(X, b)\}$, $r(X, Y)$ is the common subquery of $p(X, Y)$ and $q(X, Y)$.

First, $r(X, Y)$ is unfolded in the *idb*:

$$r(X, Y) \text{ :- } s(X, Z), u(Z, X), s(Z, Y).$$

and the *idb* is transformed into the following *idb'* with this result.

$$p(X, Y) \text{ :- } r(X, Z), p(Z, Y).$$
$$p(X, Y) \text{ :- } r(X, Y).$$
$$q(X, Y) \text{ :- } r(X, Z), s(Y, Z).$$
$$r(X, Y) \text{ :- } s(X, Z), u(Z, X), s(Z, Y).$$
$$t(X, Y) \text{ :- } s(X, Y), u(Y, X).$$

Then $p(X, Y)$ and $q(X, Y)$ is unfolded in this *idb'*.

$$p(X, Y) \text{ :- } s(X, W), u(W, X), s(W, Z), p(Z, Y).$$
$$p(X, Y) \text{ :- } s(X, Z), u(Z, X), s(Z, Y).$$
$$q(X, Y) \text{ :- } s(X, W), u(W, X), s(W, Z), s(Y, Z).$$

These subqueries are evaluated in the *edb* with the following selection conditions, $\sigma_{1=a} p$, $\sigma_{2=b} q$. □

Evaluation of crossed queries is achieved as an extension of ordered queries, that is, in an ordered case, it is considered that common subqueries are contained in a given set of queries.

## §4 Performance Evaluation

The previous section presented some partial evaluation methods for multiple query processing in deductive databases.

Compatible queries are unfolded once by their general form to avoid the same unfolding in an *idb*. While ordered queries are unfolded incrementally from the lower queries to higher queries to use the unfolded results of the lower ones for the unfolding of higher ones. For crossed queries, their common subqueries are unfolded at first, then those queries can share the common intermediate results. In either case, queries are previously analyzed to plan for efficient unfolding, then they are partially evaluated in an *idb*. Moreover, when there is given a set of queries which contains compatible queries, ordered queries and crossed queries, it is possible to combine each method.

Now an experimental results of performance evaluation is presented. For measurement, it is used a sample *idb* which consists of function free Horn clauses, composed of binary relations without constants, and including linear recursive clauses at the rate of 40% for all clauses. And each partial evaluator is implemented in DEC-10 Prolog.

First, compiled execution time of *HCT* procedure is shown in Table 1. In this *idb*, the search space grows nearly exponentially, so does the costs increase with the depth.

Next, the performance improvement obtained by the strategies is present-

**Table 1** Execution time for HCT procedure.

| depth | 5 | 10 | 15 |
|---|---|---|---|
| time (msec) | 863 | 5 224 | 12 647 |

ed. In this experiments, a set of queries which includes some compatible queries and ordered queries is assumed. Figure 1 shows the comparison of the compiled execution time. It is assumed five or ten queries which include three or six compatible queries, respectively. Then two cases are compared; the rest of seven or four queries are not ordered in one case, while they are ordered in the other case. When queries are not ordered, they are evaluated independently, while queries are ordered, they are evaluated incrementally as is mentioned in the previous section.

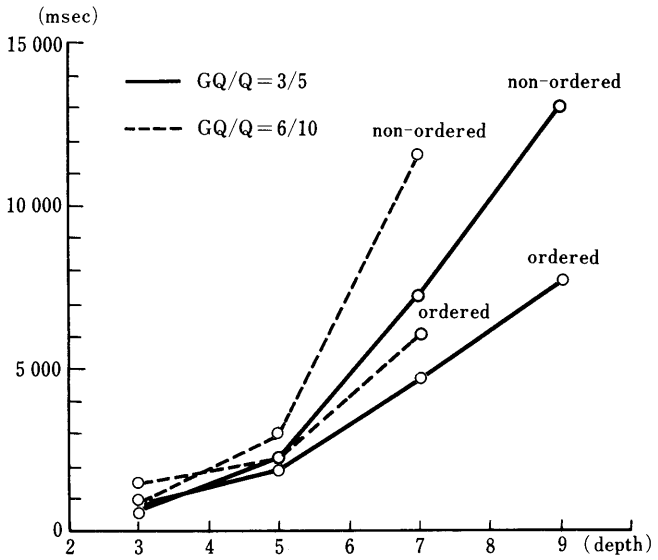


**Fig. 1** Comparison of performance evaluation.

Figure 1 shows that incremental evaluation becomes more effective in deeper depth. A crossed case is basically the same with the ordered case, then the same effect is expected.

## §5 Conclusion

This paper presented an application of a partial evaluation method to query processing in deductive databases. Three strategies for compatible case, ordered case and crossed case can reduce the redundant unfolding transformation in an *idb*, then these partially evaluated queries are compiled into relational operations and evaluated in an *edb* in the same manner of a single query.

Partial evaluation methods presented in this paper did not use the binding information of variables, that is, which variables are bound or not by given queries in an *idb*. In planning efficient computation, however, they are often very useful.[7] For example, when an *idb* contains some constants which give a condition of alternative choice of clauses, it is important to use the binding information for avoiding unnecessary expansion. Further optimization should be achieved according to the application program.

### *Acknowledgements*

We would like to thank Yukihiro Morita, Nobuyoshi Miyazaki and Toshiaki Takewaki for useful discussions and comments on an earlier draft of this paper.

### *References*

1) Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. ACM SIGMOD '86*, pp. 16-52, 1986.
2) Chakravathy, U. S. and Minker, J., "Multiple Query Processing in Deductive Databases Using Query Graphs," *Proc. 12th Int. Conf. on VLDB*, pp. 384-391, 1986.
3) Futamura, Y. "Partial Computation of Programs," *Lecture Notes in Computer Science, Vol. 147*, pp. 1-35, 1983.
4) Gallaire, H., Minker, J. and Nicolas, J. M., "Logic and Databases: A Deductive Approach," *ACM Computing Surveys, Vol. 16, No. 2*, pp. 153-185, 1984.
5) Miyazaki, N., Yokota, H. and Itoh, H., "Compiling Horn Clause Queries in Deductive Databases: A Horn Clause Transformation Approach," *ICOT Technical Report, No. 183*, 1986.
6) Reiter, R., "Deductive Question-Answering on Relational Data Bases," in *Logic and Data Bases*, Plenum Press, 1978.
7) Ullman, J. D., "Implementation of Logical Query Languages for Databases," *ACM TODS, Vol. 10, No. 3*, pp. 289-321, 1985.
8) Yokota, H., Kunifuji, S. et al., "An Enhanced Inference Mechanism for Generating Relational Algebra Queries," *Proc. 3rd ACM PODS*, pp. 229-238, 1984.