Handling Redundancy in the Processing of Recursive Database Queries

Jiawei Han and Lawrence J Henschen [1]
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois 60201

ABSTRACT

Redundancy may exist in the processing of recursive database queries at four different levels precompilation level, iteration level, tuple processing level and file accessing level Techniques for reducing redundant work at each level are studied In the precompilation level, the optimization techniques include removing redundant parts in a rule cluster, simplifying recursive clusters and sharing common subexpressions among rules At the iteration level, the techniques discussed are the use of frontier relations and the counting method At the tuple processing level, we use merging and filtering methods to exclude processed drivers from database reaccessing Finally, at the file accessing level, I/O cost can be further reduced by level relaxation We conclude that even for complex recursion, redundant database processing can be considerably reduced or eliminated by developing appropriate algorithms

1. Introduction

This paper examines some important concepts and methods relating to redundancy in the processing of recursive queries By organizing the processing into distinct stages or levels, we believe that we can provide a cleaner insight into recursive query processing Some of the techniques discussed here have been described previously in the literature and the others are to be published, but they have never been collected into a complete organized presentation Such a presentation, we believe, will greatly facilitate the understanding of the inherent problems and different methods of solving those problems

[1] The work was supported in part by the National Science Foundation under Grant DCR-860-8311

The efficient processing of recursive queries in relational database systems is a focused topic being studied by many researchers As indicated in the performance study on recursive query processing in [BaRa 86], there are three major factors which greatly influence the processing efficiency (i) the duplication of work, (ii) the set of relevant facts, and (iii) the arity of the intermediate relations

Among these three factors, the second factor, *the set of relevant facts*, has been studied by many researchers ([HeNa 84][Ullm 85][BMSU 86][Lozi 86] [KrZa 86]) Most of them use methods of early binding of query constants in order to restrict the size of intermediate results in query processing and bind the processing to the relevant set of facts, the *magic set* The third factor, *the arity of the intermediate relations*, or more generally, *the least involved attributes in the intermediate processing*, has also been studied by [HeNa 84][BMSU 86][VaBo 86], and the developed techniques include using unary instead of binary relations in the intermediate processing and using join indices or indices on join attributes to reduce the accessing cost

However, the first factor, *reducing or eliminating the duplication of work*, although a major concern of most researchers, has not been fully solved Redundancy in the processing of recursive database queries still exist in the best known algorithms Because of the nature of recursive processing, the effects of redundancy often accumulate in processing and contribute an important portion to the total processing cost Moreover, redundant processing complicates the termination of recursion on cyclic data

To systematically study the problem, we break down the processing into four levels precompilation level, iteration level, tuple processing level and file accessing level Although some researchers integrate these levels in the development of their algorithms, we find that the distinction of the levels presents a clear framework for the analysis of the problem

Like most researchers, we consider that a deductive database contains two parts (1) an *extensional database* (*EDB*) consisting of a set of base relations, and (2) an *intentional database* (*IDB*) consisting of a set of deduction rules in Horn-clauses We assume that all Horn-

clauses are safe and function-free [KrZa 86] Since a hybrid intensional predicate can be transformed into a combination of pure intensional and pure extensional predicates [CMG 86], we assume that the *IDB* contains only *pure* intensional predicates

A predicate P is said to *imply* a predicate R (P -> R) if there is a Horn clause in *IDB* with predicate R as the head and P in the body, or there is a predicate Q where P -> Q and Q -> R (transitivity) A predicate R is called *recursive* if R -> R A query is called *recursive* if it links directly or indirectly to one or more recursive predicates

The rules in *IDB* which directly or transitively define a predicate P form a *rule cluster, P-cluster* We group rules in *IDB* into clusters to avoid the unnecessary search on a large body of irrelevant rules and clarify the relationships among rules in compilation and optimization If a rule directly defining a predicate P also defines another predicate R, it belongs to both the *P-cluster* and the *R-cluster* We treat a rule belonging to several clusters as multiple copies of the rule, each copy belonging to one cluster Therefore, the optimization on one cluster, such as the elimination of an intermediate predicate, will not have "side-effects" on other clusters

The variable pattern of a recursive rule influences the compilation of the rule Many interesting recursive rules are in or are transformable to *linear variable patterns* whose variables have the following characteristics (i) the body of the rule forms a well-formed join expression, i e , the neighboring variables of adjacent relations are shared, and the variables of non-consecutive relations are not shared, and (ii) the two variables of the head correspond to the starting and ending variables of the body, respectively For example, the following rule manifests a linear variable pattern

$$R(x, z) - A(x, y), R(y, w), B(w, z)$$

For a rule in linear variable pattern, we do not write variables explicitly, because they are implied Therefore, the above rule is written as

$$R - A, R, B$$

In our discussion we consider that all the variables are vectors, with a through e representing constant vectors, and all the others variable vectors We consider that all the literals are relations, with A through E representing extensional relations and P through T representing intensional relations All the join operations, either explicitly represented by $\rhd\lhd$ or implicitly represented by ABC, A^k, A^*, or A^+ (transitive closure of A) are *projected joins*, i e , the natural join followed by the projection on the non-joined attributes

Clearly, there are recursive rules whose variable patterns are not transformable to linear patterns This discussion is focused on the recursive rules that are in or are transformable to linear variable patterns, because they are frequently used and can make use of regular compilation and processing strategies

Another view of recursive query processing is to view the processing as the traversal of database graphs An extensional database is viewed as a directed database graph in which each directed edge (*driver -> result*) represents a tuple of a data relation in which one attribute value is used as a *driver* and another as a *result* in the processing A database is *cyclic* if its database graph contains cycles

The paper is organized as follows Each of section 2 to 5 discusses the handling of redundancy at one level Section 6 summarizes the work and presents some issues for future research Because of the broad scope of our discussion, the paper is descriptive in nature The detailed and rigorous treatments can be found in the papers listed in the references

2 Handling Redundancy in the Precompilation of Rule Clusters

The compilation approach has been developed to compile an *IDB* using a connection graph analysis The *precompilation* [KOT 86] of a rule cluster is the preprocessing of the cluster before specific queries are submitted to the system In the case that a predicate connection graph contains a single cycle or some special multiple cycles, the result of the precompilation is a set of compiled formulas which reference only the extensional relations and outline the query independent accessing plans of the cluster However, in the case that the connection graph contains general multiple cycles, no compiled formulas can be generated to denote complete and data independent processing plans, and the result of precompilation is a set of simplified rules [HaHe 86b] When a specific query is submitted to the system, the second phase, *query compilation and optimization*, is invoked to incorporate query constants with the results of precompilation to generate optimized query accessing plans

It is critically important at the precompilation level to remove redundant parts of the rule cluster, to simplify the complexity of a cluster and to share common subexpressions Precompilation without such optimization may generate unnecessarily complex and redundant processing plans

2 1. The Removal of Redundant Parts in a Rule Cluster

If a rule in a rule cluster or an atom in a rule cannot generate any new data for a deductive database, it is considered to be redundant, and its removal preserves the completeness of the database

Such removal can be based on the notion of equivalence of deduction rules [Sagi 87] Two rules are considered *equivalent* if they generate the same databases The removal of one of two equivalent rules in a rule cluster preserves the completeness of the deductive database The optimization of recursive rules based on this kind of equivalence has been recently studied in [Sagi 87]

The removal of a rule can also be based on the proper containment of the rules If a rule generates a data relation which is contained in that generated by other rules in the cluster, its removal will not change the

completeness of answers For example, suppose the R-cluster contains

$$R(x, y) - A(x, w), R(w, y) \quad (1)$$

$$R(x, y) - A(x, w), R(w, y), B(x, y, t) \quad (2)$$

The database generated by (2) is properly contained in that generated by (1) Therefore, (2) can be eliminated in the R-cluster

2 2 The Simplification of a Rule Cluster

If the definition of an intensional predicate S in an R-cluster does not contain more than one cycle in its predicate connection graph, S can be eliminated in the precompilation of the R-cluster [HaHe 86a], which reduces the complexity of recursion This occurs in three cases (i) S is a non-recursive intensional literal, (ii) S and R are mutually recursive, and (iii) S is at the lower level recursion than R (A set of mutually recursive predicates are at the same level of recursion If S implies R but R does not imply S, S is at the lower level of recursion than R [Viel 86])

S can be eliminated in cases (i) and (ii) by replacing S with all its definitions in the rule cluster For example, in the R-cluster

$$R - S, A \qquad S - B, R \qquad S - C$$

S can be eliminated by such substitution In case (iii), the lower level recursive predicate can be first precompiled and then the compiled formula treated as an exit expression in the precompilation of the higher level clusters One such example is demonstrated in Ex 2

2 3 Shared Precompilation among Rules in a Cluster

Common subexpressions can often be shared in the precompilation of a cluster We illustrate several typical cases in which sharing can be exploited

1 Multiple exits

An *exit rule* of an R-predicate is the rule whose head is R and whose body contains neither R nor a predicate implied by R In precompilation, multiple exit rules of a predicate can be treated as a single exit rule whose body is the union of the bodies of all its exits Two such examples are in Ex 1 and Ex 2

2 Multiple transitive closure recursive rules

A *transitive closure rule* is a recursive rule (in linear variable pattern) whose body contains only a single-sided join of a non-recursive component with the recursive literal The transitive closure rules defining the same predicate can be shared in precompilation

Ex 1 The precompilation of the R-cluster whose recursive rules are transitive closure rules

$$\begin{array}{llll} R - A_1 & & R - A_k \\ R - B_1, R & & R - B_i, R \\ R - R, C_1 & & R - R, C_j \end{array}$$

results in $B^* A C^*$ where $A = A_1 \cup \quad \cup A_k$, $B = B_1 \cup \quad \cup B_i$, and $C = C_1 \cup \quad \cup C_j$ □

3 Common subexpressions among the definitions of a predicate

If a predicate is defined by several rules which contain a common subexpression and the remaining subexpressions of these rules are located at the same side when joining with the common subexpression and do not contain recursive components of the same or higher level, the common subexpression can be shared in precompilation This can be seen in Ex 2

Ex 2 The precompilation of the following R-cluster

$$\begin{array}{lll} R - S, R, B & R - A, S & R - A, C, E \\ R - S, R, A, C & S - D, S & S - E \end{array}$$

S is first precompiled to $D^* E$, then R is compiled to $\bigcup_{k=0}^{n} (D^* E)^k A (D^* \cup C) E (B \cup AC)^k$ □

By optimization in precompilation, many recursive rule clusters can adopt the processing strategies similar to the following three canonical recursive clusters

1 The *canonical transitive closure cluster*, which consists of

$$R - A \quad (3)$$

and

$$R - A, R \quad (4)$$

and whose compiled formula is A^+

2 The *canonical linear recursive cluster*, which consists of

$$R - B, R, C \quad (5)$$

and (3), and whose compiled formula is

$$\bigcup_{k=0}^{n} B^k A C^k \quad (6)$$

3 The *canonical nonlinear recursive cluster*, which consists of

$$R - B, R, C, R, D \quad (7)$$

and (3), and which cannot be precompiled into simple formulas A multi-formula merged processing algorithm has been developed for efficient query processing on such clusters [HHS 87]

In the following discussion, these canonical recursive clusters are used as models to study the query processing and optimization on a typical recursive query *given a query constant a or a set of query constants, find all z 's in the recursive relation* R, i e ,

$$?- R(a, z) \quad (8)$$

3 At the Iteration Level. Using Frontier Relations and the Counting Method

When a query is submitted to the system, the query compilation phase starts Besides the early binding of query constants, embodied by *performing selection first*, an important optimization technique is to *avoid the reprocessing of the portions of compiled formulas processed in previous iterations*, which reduces redundant

processing at the iteration level This can be done by saving and using certain intermediate relations, which we call *frontier* relations

3 1 The Henschen-Naqvi Algorithm

The technique of saving the intermediate results for the next iteration in the processing of linear recursive queries is developed in the Henschen-Naqvi (HN) algorithm [HeNa 84] Without such saving, the derivation at the k-th iteration has to start from the scratch, repeating or partially repeating the processing of the previous (k - 1) iterations

The processing of query (8) for the canonical linear recursive cluster proceeds as follows The first set of answers (when k = 0) is obtained by $A(a, z)$ At the first iteration (k = 1), *Frontier* is initialized to σB At the k-th iteration, new *Frontier* (σB^k) is obtained by performing the join of the saved *Frontier* (σB^{k-1}) with B, and the remaining processing is to join with A once and C k times to derive the k-th set of answers This can be written as

$$(\sigma B^{k-1} \bowtie B) \bowtie A \bowtie C \bowtie \quad \bowtie C$$

The query processing terminates for acyclic databases when *Frontier* becomes empty

If the predicate C or B is missing in the linear recursive rule (5), the derived plan is the HN algorithm for the case of transitive closure clusters ($\sigma B^* A$ or $\sigma A C^*$)

3 2 The Counting Method

The HN algorithm avoids redundant processing of the *up*-part (B-part) by using the *Frontier* relation However, at the k-th iteration, it still involves k joins in the processing of the *down*-part (C-part) The total number of joins to process a recursive query is $O(n^2)$, where n is the maximum number of iterations in the processing

At first glance, it seems that the *down*-part processing is very hard to be shared among iterations, because each iteration uses a different set of drivers derived from $\sigma B^k A$ However, if there is neither *cyclic data* (data directly or transitively rederived by itself) nor *asynchronous data* (acyclic data but rederived at different iteration levels) in the database, the counting method [BMSU 86] can be used to improve the performance The method groups the processing in the *up*-part and *down*-part into two fixed-point operations, by associating with each intermediate tuple a single level number with the format (*value*, *level*), where *level* indicates the distance of the *value* to the query constant

The first fixed-point operation, the derivation of the *UpClosure*, is initialized by associating the query constant with a level number 0 This is entered into *Frontier* and also *UpClosure* At each iteration, new values are derived by joining *Frontier* (the driver set) with relation B, and each value is associated with a level number which is its driver's level number incremented by 1 These new tuples form the new driver set (*Frontier*) for the next iteration and are also appended to *UpClosure* The derivation of *UpClosure* terminates when *Frontier* becomes empty

The second fixed-point operation is initiated by the join of the *UpClosure* and A (the *flat* relation) which derives the driver set for the processing of the *down*-part The remaining processing may start from the tuples with the highest level number, stepping-down at each iteration with the associated level number decremented by 1, until the maximum level is decremented down to 0 All the tuple values with level number 0 are the answers to the query

Using the counting method, the processing of a linear recursive query is reduced to two fixed-point operations, and the total number of joins is reduced to $O(n)$ where n is the maximum number of iterations in the *up*-part processing

Although the counting method has fewer joins, each involves a larger number of tuples than the HN algorithm The major reason that the counting method may outperform the HN algorithm is that the grouping of larger numbers of tuples facilitates set-oriented processing and reduces page accesses to files The counting method also eliminates the redundancy arising from cylinder-shaped data [BaRa 86] However, it cannot eliminate redundancy arising from asynchronous and cyclic data For the same reason, the algorithm does not work when the query constant contains a set of values which derive some common data at different levels

3 3 Stack-Directed Query Processing of Non-Linear Recursion

Since the canonical nonlinear recursive cluster can not be precompiled into simple formulas, a stack-directed query processing (STK) algorithm [HaHe 86b] has been developed which uses a stack to guide the ordered, complete but non-redundant generation of compiled formulas

According to the algorithm, the processing of the same query for the canonical nonlinear recursive cluster proceeds as follows The query literal R is first pushed onto the stack and marked *active*, and the first compiled formula is generated by calling the exit rule Later processing adopts the following rules If the *Frontier* relation (the portion from the leftmost literal up to but not including the current active literal) is not exhausted, *push* is performed by substituting the active literal using its non-exit definition The left-most intensional literal of the pushed expression is marked *active*, and the remaining newly pushed intensional literals are marked *potentially active* At every push, a new compiled formula is generated by substituting the intensional literals of the new stack element by their exit definitions

The stack is *popped* if the active literal is *exhausted*, i e, the further expansion on this literal cannot derive any new results The active point is shifted to the first potentially active literal, and the saved first *potential Frontier* relation of the last iteration (the portion in front of the first potentially active point of the stack element) is used to derive the new *Frontier* relation The processing terminates when the stack becomes

Stack Operation	Top Stack Element	Compiled Formula
push	**R**	σ A
push	B, **R**, C, *R*, D	(σB) AC AD
push	B, B, **R**, C, *R*, D, C, *R*, D	(σBB) AC ADC AD
push	B, B, B, **R**, C, *R*, D, C, *R*, D, C, *R*, D	(σBBB) AC ADC ADC AD
pop, then push	B, B, R, C, B, **R**, C, *R*, D, D, C, *R*, D	(σBBACB) AC ADDC AD

Fig 1 Stack-Directed Query Processing of the Canonical Non-linear Recursive Cluster

empty

Fig 1 is an example of a stack-directed query plan generating sequence, where the bold-faced literal of a stack element represents the *active* state, the italicized *potentially active*, and the Roman (intensional literal) *exhausted* In the compiled formulas, spaces inside the generated formulas indicate the potential saving points for shared processing in future iterations, and the parentheses indicate the grouping of intermediate results as *Frontier* relations The *Frontier* relation generated by push after a pop, such as ($\sigma BBACB$), is obtained using the saved *potential Frontier* relation $\sigma BBAC$

The stack-directed query processing method is a general method to handle nonlinear recursion A detailed discussion of the algorithm is in [HaHe 86b] However, the method often generates a large number of compiled formulas which could be quite expensive in processing and may also involve redundant processing for individual tuples Further optimization is exploited in the next section

4. Handling Redundancy at the Tuple Processing Level

The techniques discussed so far cannot handle redundancy arising from asynchronous and cyclic data which are not uncommon in real databases Redundancy arising from such kind of data accumulates because a reused driver follows exactly the same derivation paths as in previous iterations Therefore, it may considerably degrade the performance and complicate the testing of termination on cyclic data

Such redundancy can be avoided by the exclusion of processed drivers from reaccessing the database in the processing of the same subformula Three algorithms, the δ wavefront, the level-cycle merging and the multi-formula merged processing, are developed for three kinds of recursive clusters, respectively Since this optimization involves operations associated with each individual tuple, it is considered as the optimization at the tuple level

4.1. The δ Wavefront Algorithm for Processing Transitive Closure Queries

In the processing of transitive closure queries, the exclusion of processed drivers from database reaccessing can be easily realized using the *δ wavefront algorithm* [HaHe 86a], which is a modification of the HN algorithm

The idea of the algorithm is to use two temporary relations *Frontier* and *Closure*, where *Closure* collects the processed drivers in previous iterations and *Frontier* collects those derived from the last iteration but **not already in** *Closure* Asynchronous and cyclic data, when rederived, are excluded from *Frontier* because they are already in *Closure*, so that their database reaccessing is avoided The processing terminates because *Frontier* will eventually become empty for any finite database The answer is the set of values which have been derived and collected in *Closure*

The δ wavefront algorithm is different from the semi-naive algorithm [BaRa 86] The former is to derive the partial transitive closure (σA^+) using query constant a while the latter is to derive the whole transitive closure (A^+)

4.2 The Level-Cycle Merging for Processing Linear Recursive Queries

Based on the same philosophy, a *level-cycle merging method* [HaHe 87] is developed for processing linear recursive queries It is a modification of the counting method, and is composed of two subalgorithms (i) the *level merging (LM) algorithm* for handling asynchronous data and (ii) the *level-cycle merging (LCM) algorithm* for handling cyclic data

To handle asynchronous data by level merging, we associate each intermediate tuple value with *a list of level numbers* rather than a single level number When an element is rederived, merging is performed on its existing level list to integrate new level information, and the newly merged level information is propagated to its processed descendants With such modification, the processing of a linear recursive query can be accomplished using two fixed-point operations with no driver accessing a data relation more than once This is shown in Ex 3

Ex 3. Suppose the relations B, A and C are shown in Fig 2 The query (8) on the canonical linear recursive cluster is processed as follows

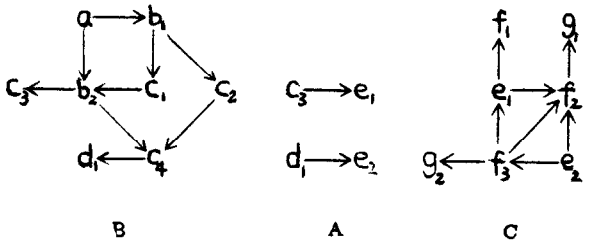

Fig 2 The Database Graphs of Relations B, A and C

1 The *UpClosure* derivation

Starting from $(a, [0])$, the first iteration derives $\{ (b_1, [1]), (b_2, [1]) \}$ The second iteration derives the new *Frontier* $\{ (c_1, [2]), (c_2, [2]), (c_3, [2]), (c_4, [2]) \}$ The third iteration merges on b_2 deriving $(b_2, [1, 3])$ which propagates a new level to c_3 and c_4 deriving $\{ (c_3, [2, 4]), (c_4, [2, 3, 4]) \}$, and derives a new tuple $(d_1, [3, 4, 5])$ The process terminates at the fourth iteration which derives empty results The *UpClosure* contains all these values with their associated level lists

2 The *DownClosure* derivation and answer extraction

The join of the *UpClosure* with the flat relation A derives $\{ (e_1, [2, 4]), (e_2, [3, 4, 5]) \}$ The first iteration on C derives $\{ (f_1, [1, 3]), (f_2, [1, 2, 3, 4]), (f_3, [2, 3, 4]) \}$ The second iteration generates $\{ (g_1, [0, 1, 2, 3]), (g_2, [1, 2, 3]) \}$ and modifies the existing tuple $(e_1, [1, 2, 3, 4])$ by merging and propagation The propagation of e_1 obtains $\{ (f_1, [0, 1, 2, 3]), (f_2, [0, 1, 2, 3, 4]) \}$, with no change on g_1 Since g_1 and g_2 have no descendants, the process terminates Therefore, the answer to the query is $\{ f_1, f_2, g_1 \}$

The algorithm can be outlined based on the above example

1 Each intermediate tuple is represented as (*value*, *level_list*) The new level list of a derived tuple is obtained by incrementing (or decrementing) each level number of the level list of its driver by 1 in the *up-* (or *down-*) part processing The answers to the query are those containing level 0 in the *DownClosure*

2 When a tuple value is rederived, merging is performed by unioning two level_lists of the tuple the *existing* level_list and the newly *derived* level_list After merging, the new level information should be propagated to the processed descendants of the rederived value Propagation is performed by following the *driver-result* paths in a processed-pair buffer (a buffer for keeping track of processed *driver-result* pairs), and incrementing or decrementing each level number by 1 in the level lists of the *result* tuple in the *Frontier* and *Closure* if the result level number is greater or equal to 0

The processed-pair buffer is usually much smaller than the whole data relation Therefore, processing by examining the buffer is relatively less expensive than reaccessing data relations Moreover, since one driver is associated with several level numbers after such propagation, one access later would require several accesses without merging Redundant tuple accessing is eliminated because rederived elements are excluded from reentering *Frontier*

Further development leads to the *level-cycle merging algorithm (LCM)* which uses two fixed-point operations to find all solutions and terminate on cyclic databases According to the algorithm, the complete level and cycle information is registered for each derived value using a level-cycle list in derivation The first fixed-point operation derives *UpClosure* which registers a generated level-cycle list for each derived element The second closure operation derives *DownClosure* which registers not only a generated level-cycle list for each derived element but also an inherited level-cycle list (inherited from its up-closure predecessors) The answers to the query are those which meet (i e , can be generated) at the same level in their two lists in the *DownClosure*

To illustrate the idea, we present a tiny example as follows

Ex 4 The relations B, A and C are shown in Fig 3 which contains cyclic data We discuss the processing of query (8) on the same cluster as in Ex 3

The counting method does not work on cyclic data If we use the HN algorithm and mark a driver *dead* when the driver cannot derive new results on cyclic data, it often takes many iterations before all the active drivers are marked *dead* In Fig 3, the up-relation has a cycle of length 6 and the down-relation has a cycle of 5 It takes $5 \times 6 = 30$ iterations to mark all the drivers dead and the number of joins to be performed is $\sum_{k=1}^{30} (k+1) = 495$ However, the LCM algorithm needs only $6 + 5 = 11$ joins (or less if it contains sub-cycles or asynchronous data) The processing is illustrated as follows

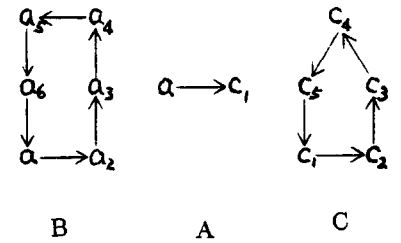

Fig 3. The DB Graphs Which Contains Cyclic Data

1 The *UpClosure* derivation derives *UpClosure*

$\{ (a, [0(6)]), (a_2, [1(6)]), (a_3, [2(6)]), (a_4, [3(6)]), (a_5, [4(6)]), (a_6, [5(6)]) \}$

The notation $i(j)$ means that the corresponding value is derived from the query at level i with cycle length j, i e , it appears at level $i + j \times k$ for every nonnegative integer k

2 After passing A, there is only one tuple left $(c_1, [0(6)])$, which drives the *DownClosure* processing and derives *DownClosure*

$\{ (c_1, [0(6)], [0(5)]), (c_2, [0(6)], [1(5)]), (c_3, [0(6)], [2(5)]), (c_4, [0(6)], [3(5)]), (c_5, [0(6)], [4(5)]) \}$

where a tuple, such as $(c_5, [0(6)], [4(5)])$, follows the format (value, [inherited level-cycle list], [generated level-cycle list]), which indicates that c_5 is generated by starting from the query constant, going up $0 + 6 \times j$ levels and join down $4 + 5 \times k$ levels, where j and k are nonnegative integers Two expressions meet at $j = 4$ and $k = 4$ That is, c_5 is reached by

going up from a 24 levels, across the flat relation and down 24 levels Therefore, c_5 is an answer In fact, every c_i in the *DownClosure* meets at some levels, so the set of answer is $\{c_1, c_2, c_3, c_4, c_5\}$

If data relations contain complicated cycles and asynchronous data, the derivation process becomes complex and challenging We are studying methods to preprocess databases at the compile time to derive a query independent self level-cycle list for each data element At query processing time, the major operations are merge and propagation The compiled self level-cycle lists are data dependent, so the cost of updating the database is higher However, the cost of query processing in cyclic database will be very close to that in acyclic databases The detailed discussion is in [HaHe 87]

We have three algorithms in handling linear recursive database queries the counting method for acyclic databases containing no asynchronous data, the LM algorithm for general acyclic databases, and the LCM algorithm for cyclic databases Although the more sophisticated algorithm works on simpler databases, it is wise to select the matched algorithm for a known extensional database

4 3 The Multi-Formula Merged Processing of Non-Linear Recursive Queries

The stack-directed processing algorithm often generates many similar subformulas for a nonlinear recursive query This is costly in processing However, the processing can be optimized by sharing and merging of the generated formulas and by accessed driver filtering

Suppose we process the same query on the canonical nonlinear recursive cluster in acyclic databases First, since the generated formulas contain subsequences like $\bigcup_{k=0}^{n} B^k A (CAD)^k$, level-merging can be used to derive the *UpClosure* for B^k and then the *DownClosure* for $(CAD)^k$ Secondly, since the formula $B^k A (CRD)^k$ may contain deeper levels of recursion, such as $B^k A (C (B^j A (CAD)^j) D)(CRD)^{k-1}$, the processing of the innermost level subformula $B^j A (CAD)^j$ needs its own level_list to represent the innermost nesting level information For example, suppose the value (c , [5, 7]) is obtained after evaluating $B^k AC$ The value c is used to begin an evaluation of $\bigcup_{j=0}^{n} B^j A (CAD)^j$ The list [5, 7] cannot be used for this processing because it is relevant only to the processing of $(CAD)^k$, i e to processing at the outermost level A new independent level list must be used for $B^j A (CAD)^j$ Suppose the *UpClosure* for B^j produces { (d, [5, 7][3, 4] } Then values in the *DownClosure* of $(CAD)^j$ whose levels match 3 or 4 would be answers to the whole subexpression $B^j A (CAD)^j$ These values would be used to continue the processing of $D (CAD)^{k-1}$ at the outermost level, where, of course, the original level list, [5, 7], would again be relevant Thus, after the processing of the innermost subformula, the innermost level_list is dropped before returning to the outer nesting levels Therefore, it is necessary to have a collection of level_lists associated with intermediate tuples with one level list for each level of nesting

Based on such observations, the processing is divided into two kinds of processes, *Deriving_UpClosure* and *Deriving_DownClosure* Each *Deriving_UpClosure* opens a new scope (a new level list) to register level numbers, and each *Deriving_DownClosure* encloses two sub-processes (i) the processing of a deeper level of recursion (*UpClosure* and *DownClosure*) and (ii) the processing of the down-part relations and returning results to the outer nesting level For example, for $B^k A (C (B^j A (CAD)^j) D)(CRD)^{k-1}$, the processing results of $B^j A (CAD)^j$ are returned one level up, joining with D and continuing the processing at the $B^k A (CAD)^k$ level When the outermost level-list of a returned tuple becomes empty, it contributes to the answers to the query

Furthermore, the processing of a subformula can be split into two paths the processed driver path and the unprocessed driver path If a driver of a subformula has not been processed before, database accessing is initiated However, if the driver has been previously processed, the result can be obtained by examining the processed pair buffers Therefore, duplicated database accessing is eliminated After the processing of the subformula, the result of the two paths are merged to facilitate set-oriented processing

For cyclic databases, the subalgorithm of the multi-formula merged processing is the level-cycle merging algorithm The other aspects are the same as in handling acyclic databases Using multi-formula merge processing, the processing of a nonlinear recursive query can be reduced to a small number of fixed-point operations A detailed study of the method is in [HHS 87], and the performance study on a tiny database demonstrates that the algorithm may have significantly better performance than the query-subquery approach [Viel 86]

5 Iteration Level Relaxation at the File Accessing Level

We have examined the processing at as low as the tuple level It seems that there is not much room for further optimization However, the examination of a more detailed level, the *database accessing level*, discloses that further optimization is still possible

In relational databases, data or index relations are usually stored in many pages (or segments) on disks, and I/O accessing has always been a major cost in query processing Since recursive query processing often involves iterative processing on the same data relations, a data or index page may have to be repeatedly fetched and accessed in the processing of a query This is another potential source of redundancy

Such repeated fetching and accessing of the same data pages arises from the strict level distinction in the processing, i e , the processing of the next iteration level cannot start until that of the current level is finished However, such level restricted processing is not necessary when the iteration level distinction is not important,

such as transitive closure queries, or the correct level information is registered in intermediate results, such as in level-cycle merging algorithms In these cases, if the strict level distinction in the processing is relaxed, it is possible to fully exploit the drivers accessible in main memory before a page is swapped out, and the I/O accessing cost can be further reduced

As an example, we examine the processing of transitive closure σA^+ using the level-relaxed version of the δ wavefront algorithm Similar to the processing of conventional database queries, we assume that the data/index pages of extensional relations are sorted on join attributes, and the tuples of the same drivers reside in the same page

Two temporary relations, *Frontier* and *Closure* are initiated to the query constants During the processing, the newly derived value is entered into *Closure* if it is not already there If a newly entered value is still accessible in main memory, the value is used immediately to derive additional results Otherwise, there are two possible cases the value is either within or beyond the range of the accessible values of the current main memory In the first case, the driver cannot generate new data and should be dropped from or not included in *Frontier* In the second case, the driver may reside in other pages on the disks Therefore, it should be entered into *Frontier* and be examined when new pages are fetched into the main memory When all drivers in *Frontier* are inaccessible in the main memory, new pages are fetched (based on the live drivers) The process continues until all the drivers in *Frontier* are dead We use a simple example to illustrate the idea

Ex 5 The comparison of the processing of σA^+ with and without level-relaxation based on the following simple database

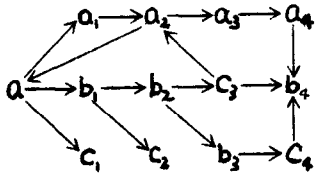

Fig 4 The Database Graph of Relation A

We use the level-relaxed version of the δ algorithm Suppose the main memory can hold only *Frontier* and *Closure* buffers and one data page, and all the drivers starting with letter a are stored in page A, those starting with letter b are stored in page B, and similarly for page C Without level-relaxation, 5 iterations are required The first iteration (using a as driver) takes 1 page access, and each of the remaining iterations takes at least 2 page accesses Therefore, it takes at least 9 page accesses in total With level-relaxation, the first page access (page A) derives a, a_1, a_2, a_3, a_4, b_1, b_4 and c_1 in *Closure* and leaves b_1, b_4 and c_1 in *Frontier* The second page access (B) appends c_2, b_2, b_3, c_3, c_4 into *Closure* and leaves c_2, c_3 and c_4 in *Frontier* The third page access adds nothing to *Closure* and empties *Frontier* It takes only 3 page accesses in total

By level-relaxation, the paths traversible in main memory are fully exploited Therefore, the chances of refetching and reaccessing of the same data page are considerably reduced Nevertheless, it does not indicate that a page will be fetched to main memory at most once Bridges linking a derivation path may not reside fully in main memory, the page may have to be swapped out and re-entered into main memory when the required bridge is built-up later

The level relaxation method can also be used in the processing of other linear and nonlinear recursive queries Since the algorithms developed at the tuple processing level register correct level information for each intermediate tuple in the processing, the query processing is independent of the derivation level and the relative processing order, which also indicates that parallel processing can be easily exploited based on these algorithms

6 Discussion and Conclusion

The elimination of redundant or duplicate database accessing is a challenging task in recursive query optimization We discussed the processing of recursive queries at four different levels and examined some techniques for handling redundancy at each level The level separated analysis presents a global picture of recursive query optimization However, a query optimizer may generate query processing plans by integrating these levels We feel that the entire process should contain two separate phases the first phase, precompilation, and the second phase, the generation of query processing plans by integrating the optimization techniques discussed in the three lower levels Moreover, our discussion demonstrates that the increase of the complexity of recursion usually increases the complexity in handling redundancy in query processing However, even for nonlinear recursion, redundant work can be considerably reduced or eliminated by appropriate algorithms

Our study assumes that the recursive rules are in or are transformable to linear variable patterns This assumption covers many interesting recursive rules However, the techniques studied here may not be directly applicable to the rules in other variable patterns Compilation and optimization of rules in other variable patterns are discussed in [HeNa 84][IoWo 86][Sagi 87] Further study is needed on reducing redundant processing on such recursive rules

Our study concerns the techniques to derive partial closures based on the query information Some applications may require to derive whole closures Processing techniques for the derivation of the whole transitive closure, such as the logarithmic algorithm, semi-naive algorithm, and Warren's algorithm, have been studied in [VaBo 86][BaRa 86][LMR 87] Techniques for the derivation of the whole closure for more general recursion is another interesting topic for future study

Our study concerns only a simple query of finding all solutions in a single recursive cluster Further study is needed on the optimization techniques for more complex and interesting queries on one or multiple recursive clusters, such as finding one or several solutions with certain features (e g minimum or suboptimal values, information related to traversal paths) [RHDM 86][Agra 87]

References

[Agra 87] R Agrawal, "Alpha An Extension of Relational Algebra to express a class of recursive queries", *Proc 3rd Int'l Conference on Data Engineering*, Feb 1987

[BaRa 86] F Bancilhon and R Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies", *Proc 1986 ACM-SIGMOD Conference on Management of Data*, May 1986

[BMSU 86] F Bancilhon, D Maier, Y Sagiv and J Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc of 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986

[CGM 86] U S Chakravarthy, J Grant and J Minker, "Foundations of Semantic Query Optimization for Deductive Databases", *The Workshop on Foundations of Deductive Database and Logic Programming*, Washington D C , 1986

[HaHe 86a] J Han and L J Henschen, "Compiling and Processing Transitive Closure Queries in Relational Database Systems", *Northwestern Univ EECS Tech Rep 86-06-DBM-02*

[HaHe 86b] J Han and L J Henschen, "Stacked-Directed Compilation of Complex Recursive Rules in Deductive Databases", *Northwestern Univ EECS Tech Rep 86-04-AI-01*

[HaHe 87] J Han and L J Henschen, "Processing Recursive Database Queries by Level and Cycle Merging", *Northwestern Univ EECS Tech Rep 87-05-DBM-01*

[HHS 87] J Han, L J Henschen and D Shi, "Optimizing the Processing of Non-Linear Recursive Database Queries", *Northwestern Univ EECS Tech Rep 87-04-AI-02*

[HeNa 84] L J Henschen and S Naqvi, "On Compiling Queries in Recursive First-Order Databases", *JACM 31(1)*, 1984

[IoWo 86] Y E Ioannidis and E Wong, "An Algebraic Approach to Recursive Inference", *Proc 1st Int'l Conference on Expert Database Systems*, April 1986

[KOT 86] C Kellogg, A O'Hare and L Travis, "Optimizing the Rule-Data Interface in a Knowledge Management System", *Proc 12th Int'l Conference on Very Large Data Bases*, Japan, Aug 1986

[KrZa 86] R Krishnamurthy and C Zaniolo, "Safety and Optimization of Horn Clause Queries", *Workshop on Foundations of Deductive Datdbase and Logic Programming*, Washington D C , Aug 1986

[LMR 87] H Lu, K Mikkilineni and J Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation", *Proc 3rd Int'l Conference on Data Engineering*, Feb 1987

[Lozi 86] E L Lozinski, "A Problem-Oriented Inferential Database System", *ACM Transactions on Database Systems 11(3)*, 1986

[RHDM 86] A Rosenthal, S Heiler, U Dayal and F Manola, "Traversal Recursion A Practical Approach to Supporting Recursive Applications", *Proc 1986 ACM-SIGMOD Conference on Management of Data*, May 1986

[Sagi 87] Y Sagiv, "Optimizing Datalog Programs", *Proceedings of the 6th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1987

[Ullm 85] J D Ullman, "Implementation of Logical Query Languages for Databases", *ACM Transactions on Database Systems 10(3)*, 1985

[VaBo 86] P Valduriez and H Boral, "Evaluation of Recursive Queries Using Join Indices", *Proc 1st Int'l Conference on Expert Database Systems*, April 1986

[Viel 86] L Vielle, "Recursive Axioms in Deductive Databases The Query-Subquery Approach", *Proc 1st Int'l Conference on Expert Database Systems*, April 1986