Traversal Recursion A Practical Approach to Supporting Recursive Applications

Arnon Rosenthal Sandra Heiler Umeshwar Dayal Frank Manola

Computer Corporation of America
Four Cambridge Center
Cambridge Massachusetts 02142

lastname@cca (ARPA) or decvax!cca!lastname

Abstract

Many capabilities that are needed for recursive applications in engineering and project management are not well supported by the usual formulations of recursion We identify a class of recursions called "traversal recursions" (which model traversals of a directed graph) that have two important properties they can supply the necessary capabilities and efficient processing algorithms have been defined for them First we present a taxonomy of traversal recursions based on properties of the recursion on graph structure and on unusual types of metadata This taxonomy is exploited to identify solvable recursions and to select an execution algorithm We show how graph traversal can sometimes outperform the more general iteration algorithm Finally we show how a conventional query optimizer architecture can be extended to handle recursive queries and views

1 Introduction

Recursive queries over databases are important both for direct applications and for supporting inference in intelligent systems Consequently definition and efficient processing of these queries have received a great deal of attention in the research literature In this paper we concentrate on recursive computations over paths trees and graphs These recursive structures arise in CAD/CAM (part hierarchies) cartography (feature hierarchies) planning and scheduling (task networks) information retrieval (hierarchically organized subject indices) and semantic database systems (generalization hierarchies) These applications are poorly served by many proposals for adding recursive facilities to database systems because the proposals do not directly address the need for arithmetic aggregation of information and control over the creation of result entities

Rather than adding general capabilities that are inefficient and not guaranteed to terminate we have developed a special class of recursions *traversal recursions* that provide the needed computational power We exploit the class s structural simplicity to define new types of assertions about the underlying database to limit the data accessed and the computation performed and to provide convenient user interface facilities

This work was supported by the Defense Advanced Research Projects Agency and by the Space and Naval Warfare Systems Command under Contract No N00039-85-C-0263 The views and conclusions contained in this paper are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency the Space and Naval Warfare Systems Command or the U S Government

Traversal recursions can be considered a generalization of transitive closure They support most computations that a human programmer would place in a graph traversal We permit recursively defined functions and predicates that prune subtrees Two different treatments are available when paths converge at a node — a separate result may be generated for each path or the function values along the paths may be somehow aggregated into a single value Recursions may also be specified over cyclic graphs our algorithms for acyclic graphs extend straightforwardly to handle the common solvable cyclic recursions

The special properties of traversal recursions are exploited by the query optimizer In addition to the usual transformations [AHO79] traversal recursion supports a transformation that moves "monotonic" selection predicates into the recursion step Several algorithms are provided for executing the resulting query — iteration one-pass traversal and main memory processing The query optimizer selects an algorithm based on the category of the recursion (aggregation or path enumeration) the properties of the graph and derived functions the predicates that prune the solution space and data layout on disk We show how all this can be integrated within the normal steps of query processing

Previous work on adding recursion to database systems has included several ad hoc extensions to query languages (e g adding syntax for transitive closure operations to variants of QBE [ZLOO77 HEIL85] or SQL [JAME77 CLEM81]) These proposals tend to work on a very limited set of graphs and only a single implementation strategy is supported

Recently most work has focused on the problem of coupling powerful recursive Horn clause languages such as PROLOG to relational databases [AHO79 CHAN82 CHAK82 HENS84 JARK84 YOKO84 ULLM85] Few of these efforts can match the performance of special purpose algorithms like graph traversal or support

recursions involving arithmetic entity creation or aggregation

The facilities described here are being implemented at CCA as part of PROBE a DBMS prototype that supports recursion complex objects and spatial processing [DAYA85 OREN86 DAYA86]

The paper is organized as follows Section 2 introduces the basic concepts of Traversal recursion over graphs Section 3 gives formal definitions of various forms of Traversal recursion in terms of abstract graphs Section 4 shows how Traversal recursion is interpreted in terms of database concepts (entities and functions) Section 5 provides an overview of query processing techniques for Traversal recursion

2 Traversal Recursion — A Graphical Interpretation

Traversal recursion is intended to capture the power of computations that traverse a graph starting from an initial node (or set of nodes) This section illustrates the different classes of traversal recursion by means of a simple example on a graph whose edges are labelled by a nonnegative "length" function

A graph G consists of Nodes and directed Edges Information associated with a node or edge is termed a node- or edge "label Multiple edges may connect the same pair of nodes G is *reconvergent* if it contains a cycle or if it contains more than one path between the same pair of nodes The nodes and edges on paths starting at a node N are called *reachable* from N A recursion derives the reachable subgraph from some specified starting node and possibly a set of node or edge labels computed for this subgraph

Example 1 — Reachability The simplest form of recursion is computing reachability That is given a graph and an initial node "A" determine the set of nodes reachable from A More generally determine the reachable subgraph (both nodes and edges)

Example 2 — Recursively Defined Functions The next level of complication allows recursively defined functions to be computed during traversal of a nonreconvergent graph The value of a function may depend on values computed at the previous derived node In Figures 1a-b edge labels are interpreted as lengths The recursion equation defines the set of nodes reachable from A and for each such node n the computed function dist(n) The recursively derived graph would be as shown in Figure 1b

Reconvergent Graphs

Now consider the reconvergent graph in Figure 2a It includes nodes {F G H} that are on multiple paths from A A function such as dist(x) computed according to the equation in Figure 1a is no longer well defined since values computed along the different paths may conflict We distinguish two ways of extending the definition of recursion to accommodate the reconvergent case — Enumeration and Aggregation

Example 3 — Path Enumeration in Reconvergent Graphs When a node is encountered on two different paths enumeration recursions treat it as a distinct entity each time The derived graph is a tree as shown in Figure 2b (The graph would be a forest if there were multiple initial nodes) Node F G or H in the underlying graph each underlies two nodes in the derived graph There is now no ambiguity in computing values of dist(x) since each derived node corresponds to a single path in the underlying graph as shown in Figure 2b

Example 4 — Path-Aggregation in Reconvergent Graphs Path Aggregation recursions combine the different values computed on different paths For example the equation below aggregates lengths of the alternative paths by selecting the maximum (as in a task scheduling application) The derived graph is shown in Figure 2c

dist(x) = 0 if node x is the initial node "A"
otherwise Max{dist(y) + length(e)
| e is an edge (y x)}

Example 5 — Aggregation in Cyclic Graphs Consider the recursion equation in Example 4 but over a graph that includes a directed cycle (v1 v2 vm) No set of values for dist() can satisfy dist(v1) < dist(v1)+len(v1 v2) < dist(v2) < < dist(v1) The combination of cycles and arithmetic has caused the recursion to have no solution

Now suppose we change the aggregation operation from Max to Min Assuming edge lengths are positive the equation (below) has a unique solution at nodes reachable from A namely dist(x)= length of shortest path from A

dist(x) = 0 if node x is the initial node "A
otherwise Min{dist(y) + length(e)
| e is an edge (y x)}

Note that the solvability of the recursion (and later the algorithm selected) depends on the difficulty of the graph properties of the edge lengths the aggregation function and the concatenation function (+ in these examples) that combines an edge length with the length of an existing path

Example 6 — Enumeration in Cyclic Graphs By definition enumeration does not check whether a node has been previously encountered Thus the recursion will generate an infinite number of nodes unless a termination condition (e g distance < 20) is specified

3 A Formal Outline of Traversal Recursion

For purposes of this paper the result of a recursion is defined as the solution (least fixed point) of a recursion equation over an *underlying graph* The underlying graph denoted UG is a labeled digraph with the usual semantics UNode and UEdge are the nodes and edges of UG respectively A function Outedges(UNode) is defined that returns the UEdges leading out of UNode Functions Beginnode(UEdge) and Endnode(UEdge) are defined that return the beginning and ending nodes respectively of UEdge

The result of the recursion will be a *derived graph* in general the derived graph s topology and labelling functions will differ from those of the underlying graph (see Figures 1 2)

Enumeration and aggregation recursions correspond to equations whose forms differ slightly These are defined in Sections 3 1 and 3 2 Section 3 3 discusses recursion over cyclic graphs Section 3 4 shows how to express the various forms of traversal recursion by Horn clauses

3 1 Traversal Recursion with Path Enumeration

The general form of an enumeration recursion equation is

(1) DResult = DInitial union Traversal_Step(DResult UG)

where

UG is the *underlying graph*

DResult is the unknown in the equation and denotes a set of *derived* (result) nodes and edges

DInitial (the base step of the recursion) is an expression that does not reference the unknown and denotes a set of derived nodes

Traversal_Step() denotes a function that operates on a set of

derived nodes and edges, and on UG, to produce a set of derived nodes and edges, and node labels. More precise conditions are given below.

The recursion equation says, in effect, that d is in DResult iff d is in DInitial or (exists d' in DResult where d = Traversal_Step({d'}, UG)). Traversal_Step satisfies conditions 1 and 2 below:

1. The total functions

```
Underlying(DNode) --> UNode
Underlying(DEdge) --> UEdge
```

are defined for DNodes and DEdges, respectively, in DResult.

2. Traversal_Step(DResult,UG) computes its output by traversing the outedges of the underlying nodes of the DNodes of DResult, i.e. by evaluating Outedges(Underlying(DResult)). The computation of DResult includes the computation of the Underlying functions and the node labels for DResult. Specifically, Traversal_Step is of the form:

```
For each d in DNode
  For each UEdge in Outedges(Underlying(d))
   {Compute dn in DNode with functions:
     Underlying(dn) := Endnode(UEdge)
     label(dn) := con{label(d),label(UEdge),
                      label(Underlying(dn))}
    Compute de in DEdge with functions:
     Underlying(de) := UEdge
     Beginnode(de) := d
     Endnode(de) := dn
    Where  traversal_predicate(d,UEdge,Underlying(dn))
```

Con (concatenation) is a function that computes a node label based on supplied argument labels. Traversal_predicate is a condition that tests whether the traversal of a particular edge should take place.

For the recursion in Figure 2b, Traversal_Step has the form:

```
For each d in DNode
 For each UEdge in Outedges(Underlying(d))
 {Compute dn in DNode with functions:
   Underlying(dn) := Endnode(UEdge)
   dist(dn) := dist(d) + len(UEdge)}
  Compute de in DEdge with functions:
   Underlying(de) := UEdge
   Beginnode(de) := d
   Endnode(de) := dn
 }
```

If UG is acyclic one can obtain the solution by traversing UG in topological order. A simple induction shows that the solution is correct and unique.

The recursion can easily be extended to generate edge labels for the DEdges. A more significant extension would be to provide for the computation to terminate as soon as a result entity satisfying a given termination predicate was generated. This interrupt facility would speed computations for shortest paths between a specified pair of points, ancestor predicates, and least common ancestors.

Traversal recursions of the form of Equation 1 are called *enumeration recursions*. Recursions on nonreconvergent graphs may all be expressed as enumeration recursions.

Recall from Section 2 that each path from an initial node to any other node will yield a derived node. Hence, the result of an enumeration recursion on a reconvergent acyclic graph can contain distinct nodes whose labels and functions (including Underlying) are identical.

3.2 Traversal Recursion with Path Aggregation

The other form of traversal recursion, *aggregation recursion*, *combines* label and function values computed by Traversal_Step along all edges into an underlying node. Aggregation recursion includes Reachability and also more complex aggregations such as optimal path problems or summations from subtrees. Aggregation recursions extend equation 1 by adding an aggregation operator:

```
(2) DResult =  N_aggr{DInitial union
        TraversalStep(DResult,UG) |
        group_by UNDERLYING(DResult) }
```

We call a set of DResult instances "matching" if they have the same value for the Underlying function. N_aggr denotes a function that maps a set of matching DResult instances to a single DResult instance, and aggregates the labels of the matching DResult instances to produce a single aggregate label. The result of N-aggr is a graph isomorphic to a subgraph of the underlying graph, but possibly with different labels. Hence, equation 2 can be viewed as a recursive computation of labels.

If UG is acyclic, Equation (2) may be solved inductively, like equation 1.

3.3 Traversal Recursion over Cyclic Graphs

Graphs with cycles can create problems in Traversal recursion, because the recursion equation (still Equation (1) for enumerations, equation (2) for aggregations) may not have a solution.

Enumeration recursions will be unsolvable unless Traversal_Predicate() is present and terminates each path.

Aggregation computations generate only nodes and edges corresponding to those in the original graph, but in cyclic graphs we must determine whether the derived labels are well defined. The theory of path algebras [CARR79] provides conditions under which a recursion is equivalent to a well defined computation of labels of a set of paths in the graph.

A *path label* is given by a *concatenation* (con) function applied to all its edge and node labels. The label of a set of paths is given by a *path-aggregate* (P_aggr) function applied to the labels of all paths in the set.

A *path algebra* is a structure (P, P_Aggr, con) such that:
- P is a set of path labels.
- P_Aggr is idempotent, commutative, and associative
- con is associative and distributive over P_Aggr
- P contains an identity for con and an identity for P_Aggr.

Examples of structures (P, P_Aggr, con) that form path algebras include (P, Min, +) for P = positive integers or reals (the ordinary shortest path problem), (P, Max, Min) for the same P (the "bottleneck" shortest path problem), (P, OR, AND) for Boolean P (reachability problems), and (P, Union, Intersection) for sets P.

[CARR79] shows that if UG is acyclic and (P, P_Aggr, con) is a path algebra, then the Least Fixed Point of equation 2 is well defined. The solution may be interpreted as the aggregation of

labels of all paths from D_Initial to each other node. That is, the solution can be computed by the following formula:

```
DNodelabel(dn) =
 P_Aggr{ con{ label(u) |u is an edge or node in p}
   | p is a simple path from un0 to un,
     un0 is in Underlying(DInitial),
     and un is in Underlying(dn)}
```

(In a simple path, each node appears at most once).

[CARR79] also shows that, if the path algebra satisfies a *cycle nonnegativity* condition for graph UG, then the Least Fixed Point of equation 2 is given by the above formula even if UG is cyclic, and can be found by iteration and many other algorithms. Moreover, if an additional *edge nonnegativity* condition also holds, Dijkstra's algorithm will produce the solution. All the path algebras above satisfy both edge and cycle nonnegativity. Nearly all applications satisfying cycle nonnegativity also satisfy edge nonnegativity.

3.4 Horn Clause Formulations of Traversal Recursion

Figure 3 shows how traversal recursions can be expressed in the style of "regular" Horn clause recursions [BANC85], augmented with some additional operators. We have simplified the notation by assuming that each pair of nodes is connected by at most one edge.

To express label computations, Horn clauses must be augmented with function symbols (con). Aggregation of information from different paths requires a "group-by" facility, i.e., functions symbols (P_Aggr) whose arguments are sets. Enumeration requires a function that creates new nodes. In Figure 3, new node identifiers are created out of path names, which are formed by concatenating identifiers of nodes along the path.

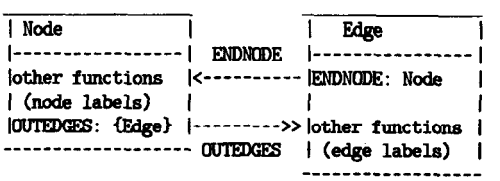
4. Traversal Recursion over Databases

This section describes the application of traversal recursion to databases. The discussion is formulated in terms of the DAPLEX functional data model [SHIP81]. The basic constructs of this model are *entities*, representing objects in the real world, and *functions*, representing attributes of those objects or relationships between them. We express recursions as queries over the database entities and functions that represent the graph. The result of a recursive query is an entity-set or sets, just like the result of any other DAPLEX query. The following subsections describe the encoding of a labelled graph in database terms, followed by examples of each of the subclasses of traversal recursion over sample databases.

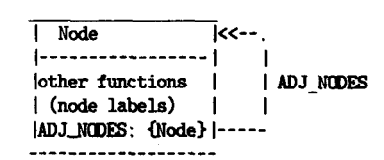
4.1 Defining a Graph within a Database

We describe two straightforward encodings of labelled graphs in terms of entities and functions. Any structure in a database that matches those forms may be treated as a graph.

First, a structure within the database may be declared as a graph by specifying two entity sets and two functions as shown below. Other functions of the node and edge entities constitute the labels.



Alternatively, a structure in a database may be declared as a graph by specifying a single entity set to model nodes, and a single function to model edges:



4.2 Recursive Queries over a Database

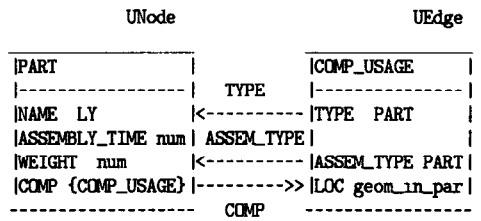
A traversal recursion over a database is a query that produces a derived graph. The derived graph is represented by entity sets and functions as above. Distinct derived node entities may have the same labels, since TraversalStep does not include duplicate removal. Entity identifiers must be generated for the derived entities. The creation of derived entities (with new entity identifiers) is automatic and poses no problem in entity-oriented data models such as DAPLEX. In the relational model, however, nodes and edges will be represented by tuples, and surrogates must be explicitly generated to supply keys of these tuples; otherwise derived nodes that correspond to the same underlying node might agree on all attributes, and hence be indistinguishable.

The user interface supplies some predefined functions (whose detailed definition depends on whether the recursion is enumeration or aggregation). The functions will be computed if referenced by the user. These include UNDERLYING (giving the UNode or UEdge entity underlying each DNode or DEdge), PREDECESSORS and SUCCESSORS (of each DNode), SELECTED_PREDECESSOR (on the optimal or chosen path), INITIAL_NODE (on enumerations, tells the node in DInitial from which the recursion that produced each DNode started), LEVEL, MIN_LEVEL, and MAX_LEVEL (which count edges in paths from DInitial), and PATH_ID (a string that uniquely identifies the path from DInitial that was traversed by the recursion).

4.3 An Example of Path Enumeration Recursion in a Database

The Parts Explosion problem from CAD/CAM was the original motivation for this class. In a design database, a part's design is described only once. For each usage of a component in the part, there is a usage entity that references the design entity for the component, and contains information about that particular usage, e.g. the location of the particular usage of the component within the part (for example, a wing may have two engines and the engines have different locations).

Consider the part (design) hierarchy defined over PART and COMP_USAGE entities as shown below.



The OUTEDGES function is COMP(PART) the ENDNODE function is TYPE(COMP_USAGE)

A recursive query might derive entities D_PART of the following form from an initial Unode part0

D_PART	
UNDERLYING	underlying part entity
DPRED	predecessor instance in D_PART
NAME	name of underlying part
ASSEMBLY_TIME	assembly time for underlying part
LOCN_IN_END_ITEM	0 if UNDERLYING is part0 else transform(LOC_IN_END_ITEM(DPRED), LOC(COMP_USAGE))

There will be a derived entity D_PART for each usage of an underlying PART entity in the assembly represented by part0 For example if part0 is the end-item airplane and a wing is used twice in the airplane and an engine is used twice in the wing the derived entities will include four engine nodes one for each path from air plane to engine Derived entities include the function LOC_IN_END_ITEM obtained from information in each derived predecessor and in the corresponding underlying usage entity

4 4 An Example of Path Aggregation Recursion in a Database

The same part (design) hierarchy defined over PART and COMP_USAGE entities described above might be used in an aggre gation recursion to derive entities of the form

H_PART	
UNDERLYING	underlying part entity
DPRED	predecessor instance in D_PART
NAME	name of underlying part
ASSEMBLY_TIME	assembly time for underlying part
DAY_NEEDED	0 if UNDERLYING is part0, else min(DAY_NEEDED(x) - ASSEMBLY_TIME(x) x in DPRED)

In this case an underlying node entity (PART) instance gen erates at most one derived entity instance even if the part has mul tiple usages The derived entities H_PART include the function DAY_NEEDED obtained by aggregating (min) information over the set of derived predecessors (DPRED) of the entity

4 5 An Example of Recursion over a Cyclic Graph in a Database

An example of Traversal recursion over a cyclic graph is given below It is based on a graph of the following CITY and ROAD entities

The OUTEDGES function OUTROADS(CITY) includes 1 way roads leading out of the city

The recursion derives PATH entities with a function SHORTEST_MILEAGE which represents the length of the shortest path from the beginning city c0 to each other city that is reachable from c0 through OUTROADS The derived PATH entities are of the form

PATH	
ROOT	starting city c0
DEST	destination city (underlying node)
PRED	PATHs from c0 to some node pre- ceding DEST
SHORTEST_MILEAGE	0 if DEST is c0, else min(SHORTEST_PATH(x)+LENGTH(ROAD) START_CITY(ROAD)=DEST(x) and DEST_CITY(ROAD)=DEST and x is in PRED(PATH))

5 Query Processing — Overview

This section illustrates via examples our approach to processing Traversal recursion queries (For details see [ROSE85]) First we describe the classes of algorithms for executing recursive queries Then we show how to integrate the processing of recursive queries into a conventional DBMS query processor architecture

5 1 Algorithms

We propose three classes of algorithms main memory computations iteration and one pass traversal

The *main memory strategy* is to read all relevant information (the underlying graph) into main memory data structures The remaining subquery is processed entirely in main memory with no further database access Either iteration or traversal may be used in main memory This approach requires only one or two separate database queries to retrieve the graph and may achieve just one access to each page containing relevant information However it must read the entire graph not just the subset reachable from DIni tial via edges satisfying Traversal_Predicate Furthermore the memory requirement is large

Iteration (breadth first search) is essentially the classical strateg described in [AHO79] using the differencing method [BANC8(BAYE85] to eliminate some recomputations The strategy is

```
DNode = DInitial
While (previous step created or changed DNodes)
    {
    Create new DNodes by traversing out of
      UNodes that underly DNodes created or
      changed at the previous step
    }
```

The computational problem introduced by aggregation is that all members of the "group" must be available before a new instance can be computed and output

An extension of this basic strategy can handle aggregation the second time a UNode is reached the old label of its corresponding DNode must be aggregated with the label computed by the new traversal If the label changes it is considered newly-created for the next step of the iteration The algorithm will terminate if the graph is acyclic or if the recursion equation is a cycle-nonnegative path algebra Figure 4 shows the steps at which the iteration algorithm reaches each node in a maximum path length problem on an acyclic graph in which all edges are of equal length [ROSE85] shows how the algorithm can be simplified for reachability problems problems where path length is given by the number of edges in a graph and other special cases

The advantages of Iteration are that it reads in many nodes with each database query that it is likely to be implemented anyhow to support deductive recursions and that it requires no special information about the recursion

In most engineering applications recursive computations are handled by *one-pass graph traversals* A *One-Pass order* is an ordering on the derived entities such that the recursively computed functions depend only on the earlier entities not on later ones If a 1-pass order can be found the functions can be computed by traversing the graph in that order

If the underlying graph is acyclic any topological ordering is a one-pass order In cyclic graphs no topological ordering exists and the recursions may be unsolvable Most of the the solvable cases seem to be variants on shortest path problems which can be solved efficiently by Dijkstra s shortest path algorithm [AHO75] We will consider Dijkstra s algorithm as a traversal which includes its own computation of a 1-pass order (in order of shortest distance from the root)

Figure 5 shows a 1-Pass traversal of the graph of Figure 4 The strategy accesses many fewer nodes than Iteration If we assume either one node per page accessed or that the nodes are stored in topological order (suitable for a one-pass traversal) traversal outperforms iteration The advantages of Traversal over Iteration are

1 Iteration propagates values before all paths have been included in the aggregation hence each derived label may be updated multiple times In the worst case N*L/2 changes are made to DNode where N is the number of derived nodes and L is the number of edges in the longest path Traversal algorithms do not propagate uncompleted aggregates

2 Iteration does not permit the optimizer to control the order in which nodes are accessed so it cannot take advantage of storage layout For example the application might know that Parts are stored sorted by Weight A one-pass traversal that visits Parts in stored order will access each page just once

3 In enumerations one can traverse simultaneously from all DNodes with the same UNDERLYING node Iteration will not process them simultaneously unless all paths from DInitial to those nodes contain the same number of edges

The problems in using one-pass traversal lie in determining legal traversal order and in the larger number of database queries issued (one per DNode) We assume that some direct access structure permits UNodes to be retrieved from the entity identifier in the function EndNode(UEdge)

5 2 Query Processing Architecture

Conventional DBMS query processors have four stages

1 semantic analysis which parses the query and accesses metadata to associate each name in the query with an object in the schema producing an internal representation of the query such as an operator DAG

2 logical transformation which uses metadata and simplification rules to transform the operator DAG into a "better" one and performs view substitutions

3 strategy selection which generates alternative implementation strategies evaluates the alternatives and finally selects one for execution producing a program that can be executed against stored files

4 execution-time support which includes creating temporaries to hold intermediate results

Our goal was to incorporate the processing of recursive queries without radically modifying this architecture There are two reasons why this is desirable First the architecture is well understood and has stood the test of implementation Second even recursive queries may include non-recursive computations having the same architecture makes it easier to optimize the whole query

We now illustrate how to enhance each stage of the query processor to handle recursive queries The following query is used as a running example

```
Find NAME(H_PART)
 where DAY_NEEDED(H_PART) > -60
```

The operator DAG produced by semantic analysis is shown in Figure 6 It contains an operator node for aggregation recursion (similarly we can introduce an operator for enumeration recursion) this node points to the template also shown in Figure 6 that captures all the information about the recursion needed by the optimizer This information comes either from the query or from metadata The template specifies the underlying and derived entity types the OUTEDGES and ENDNODE functions the beginning nodes of the traversal the computations (both recursive and non recursive) of the derived functions and various assertions (about the properties of the graph and the arithmetic operations) that can be used by the optimizer

Logical transformation produces the operator DAG of Figure 7 Functions that are not needed for subsequent processing are crossed out from the recursion template (this will translate to a projection operation later) Second the optimizer uses the monotonicity of DAY_NEEDED to move the selection condition DAY_NEEDED > 60 *inside* the template as a restriction on the edge function This means that instead of first computing *all* H_PART entities and then

restricting them the query processor can perform the restrictions *during* the traversal Monotonicity assertions can be used to restrict traversals (as in this example) to truncate traversals and to limit output (as discussed in [ROSE84]) This transformation goes beyond the transformations of [AHO79] since it depends on meta data about graph properties

The choice of a strategy for implementing the recursion operator depends upon various factors constraints on the graph (reconver gent acyclic or cyclic) form of recursion (aggregation or enumera tion) properties of functions and arithmetic operators data organiza tion on disk and presence of auxiliary traversal aids Because detailed cost models are difficult to construct for recursive queries the strategy selector uses a collection of rules (heuristics) to choose a good implementation strategy Examples of such heuristics are

- if the graph fits in main memory and there are no restrictions on the edges traversed then choose the main memory strategy
- if the form of recursion is Reachability or Shortest Path where all edges are the same length then choose the iteration strategy
- if the graph has a monotonic function f then choose a one-pass traversal strategy in which the next node to visit is the entity with the smallest value of f

The third rule is used in selecting a strategy for the aggregation recursion operator occurring in the operator DAG of Figure 7 (See [ROSE85] for a complete list of algorithm-selection heuristics)

Finally optimization is completed with a puff of black smoke - Traversal_Step and DInitial are themselves queries and may be shipped to the query optimizer This permits them to be recursively defined and makes the join ordering and detailed physical modelling facilities of the optimizer available for their optimization The opera tor DAG for these subqueries is included in the global operator DAG as shown in Figure 8

6 Conclusions

Traversal recursions are tractable practically important and poorly supported by most recent research Most recursions in engineering can be expressed as traversal recursions when the data is already in the database (e g part hierarchies) a recursive query facility can provide an immediate payoff [HEIL85]

We have categorized traversal recursions into enumerations and aggregations This categorization

- helps in determining convergence
- allows the user interface to define automatically derived functions that capture "structural" information
- highlights useful features absent from many Horn clause formula tions
- is exploited in choosing an algorithm

We have shown how traversal recursion can be integrated into a conventional DBMS query processing architecture minimizing the amount of special code needed for recursive optimization Metadata about operators and graphs can be exploited to provide very efficient processing

Future work includes

- Investigation of a cost model for selecting among alternative implementation strategies for traversal recursion
- Investigation of the interaction between the DBMS environment and the application program environment and especially of how to stage the temporary data structures used by all three implemen tation algorithms between main memory and disk
- Investigation of access structures that efficiently support traver sals
- Integration of Traversal Recursions with tactics for optimizing deductive recursions over complex sets of Horn clauses

Acknowledgements

The Authors would like to thank Jack Orenstein Upen Chakra varthy and Dave Evans for their contributions to improving the presentation of the ideas in this paper

7 References

[AHO76]
Aho A J Hopcroft and J Ullman *Design and Analysis of Computer Algorithms* Addison Wesley Reading MA 1976

[AHO79]
Aho A and J Ullman "On the Universality of Data Retrieval Languages " *ACM Symp on Principles of Programming Languages* 1979

[BANC86]
Bancilhon F "Naive Evaluation of Recursively Defined Relations " in M L Brodie and J Mylopoulos (eds) *On Knowledge Base Management Integrating Artificial Intelligence and Database Technologies* Springer Verlag 1986

[BAYE85]
Bayer R "Query Evaluation and Recursion in Deductive Database Systems " 1985

[CARR79]
Carre B *Graphs and Networks* Oxford University Press New York NY 1979 ch 3

[CHAK82]
Chakravarthy U S J Minker and D Tran "Interfacing Predicate Logic Languages and Relational Databases " *Proc of the 1st Int l Logic Programming Conf* France Sept 1982

[CHAN82]
Chandra A and D Harel "Horn Clause Queries and Generalization " *ACM SIGACT-SIGMOD Symp on Principles of Database Systems Conf* 1982

[CLEM81]
Clemons E "Design of an External Schema Facility to Define and Process Recursive Structures " *ACM Trans on Database Systems* Vol 6 No 2 June 1981 pp 81-92

[CLOC81]
Clocksin W F and C S Mellish *Programming in PROLOG* Springer-Verlag New York NY 1981

[DAYA85]
Dayal U et al "Probe - A Knowledge-Oriented Database System Preliminary Analysis" Technical Report CCA 85-03 Computer Corporation of America July 1985

[DAYA86]
Dayal U and J M Smith "PROBE A Knowledge Oriented Database Management System" in M L Brodie and J Mylopoulos (eds) *On Knowledge Base Management Integrating Artificial Intelligence and Database Technologies* Springer-Verlag 1986

[HEIL85]
Heiler S I and A Rosenthal "G Whiz A Visual Interface for the Functional Model with Recursion" *Proc of the 11th Conf on VLDB* 1985

[HENS84]
Henschen L J and S A Naqvi "On Compiling Queries in Recursive First Order Databases" *Journal of the ACM* Vol 31 No 1 Oct 1984 pp 47 85

[IOAN85]
Ioannides Y "A Time Bound on the Materialization of some Recursively Defined Views" *Proc of the 11th Conf on VLDB* 1985

[JAME82]
James G and W Stoeller "Operations on Tree Structured Tables" X3H2-26-15 Standards Committee Working Paper 1982 pp 81 92

[JARK84]
Jarke M J Clifford and Y Vassiliou "An Optimizing PROLOG Front-end to a Relational Query" *Proc of the ACM SIGMOD Int l Conf on Management of Data* 1984

[JARK85]
Jarke M Linnemann V and Schmidt J "Data Constructors On the Integration of Rules and Relations" *Proc of the 11th Conf on VLDB* 1985

[MERR84]
Merrett T *Relational Information Systems* Reston Publishing Reston VA 1984 ch 5 2

[OREN86]
Orenstein J "Spatial Query Processing in an Object Oriented Database system" *Proc ACM SIGMOD Int l Conf on Management of Data* 1986

[ROSE84]
Rosenthal A S I Heiler and F Manola An Example of Knowledge Based Query Processing in a CAD/CAM DBMS" *Proc of the Tenth Int l Conf on VLDB* 1984 pp 363-370

[ROSE85]
Rosenthal A "Traversal Recursions" Probe Project Working Paper Computer Corporation of America 1985

[SHIP81]
Shipman D "The Functional Data Model and the Data Language DAPLEX" *ACM Transaction on Database Systems* 6 1 March 1981 pp 140-173

[ULLM85]
Ullman J "Implementation of Logical Query Languages for Data bases" *ACM Trans on Database Systems* 1985

[YOKO84]
Yokota H et al "An Enhanced Inference Mechanism for Generating Relational Algebra Queries" *ACM SIGACT-SIGMOD Symp on Principles of Database Systems* 1984

[ZLOO75]
Zloof M "Query By Example *Proc of the NCC44* May 1975

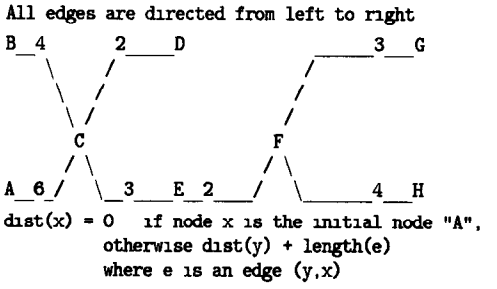


Figure 1a A Nonreconvergent Graph

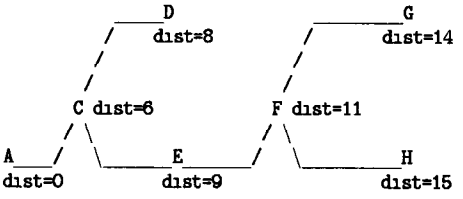


Figure 1b The Derived Graph

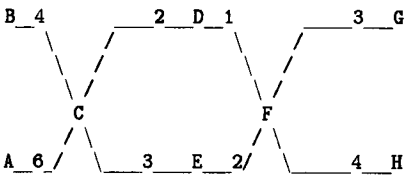


Figure 2a A Reconvergent Graph

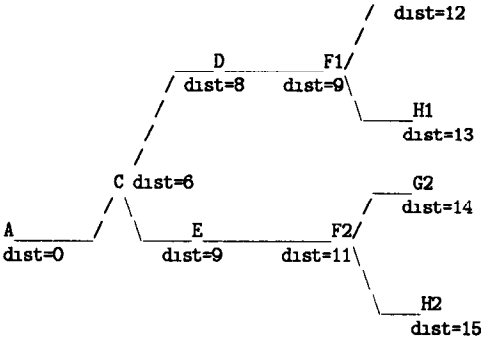


Figure 2b Path-Enumeration

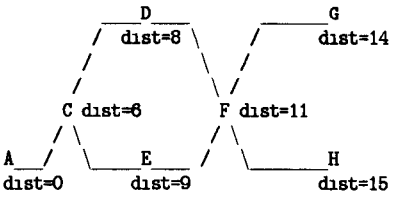


Figure 2c Path-Aggregation

E(x y L) is the assertion of an edge from node x to node y with label L

R(x y L) indicates that the recursion initialized with x derives a node y with label L

Reachability

E(x y L) --> R(x y) R(x y) ^ E(y z L) -> R(x z)

Nonreconvergent graphs

E(x y L) - > R(x y L)
R(x y L1) ^ E(y z L2) ^ Traversal_Predicate(L1 L2) ^
(L = con(L1 L2)) - >
R(x z L)

Aggregation

T(x y L) denote the values to be aggregated i e the result of TraversalStep

E(x y L) - > T(x y L)
R(x y L1) ^ E(y z L2) ^ Traversal_Predicate(L1 L2) ^
(L = con(L1 L2)) -->
P(x z L)

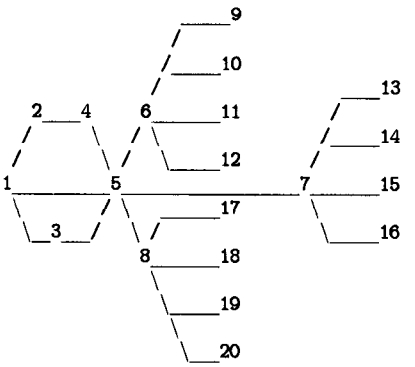
(L = P_aggr{L"| T(x y L")}) ^ T(x y L) - > R(x y L)

Enumeration

A path s node sequence is used as its NAME P(x y L NAME) denotes that a path NAME with label L exists between nodes x and y NEWNODE is a function that creates a new node for each new value of its arguments

E(x y L) ^ (NAME=[x y]) -> P(x y L NAME)
P(x y L1 NAME1) ^ E(y z L2) ^ (L = con(L1 L2)) ^
Traversal_Predicate(L1 L2) ^
(NAME = append(NAME1 [z])) -->
P(x z L NAME)
P(x y L NAME) ^
(n=NEWNODE(y NAME)) -> R(x L n)

Figure 3 Horn Clause Formulation of Traversal Recursion



Layout on disk

Page	A	B	C	D	E
	1 2 3 4	5 6 7 8	9 10 11 12	13 14 15 16	17 18 19 20

Compute maximum length paths from 1 where all edges have equal length and are directed rightward
Assume edges into a node are stored in the same page as the node

Iteration algorithm

Iteration	Nodes Retrieved	Pages Accessed
0 (Initial)	1	A
1	2,3,5	A,B
2	4 5,6-8	A B
3	5,6-8, 9-20	B,C,D,E
4	6-8, 9-20	B,C,D,E
5	9-20	C,D,E

Iteration Totals 52 nodes retrieved, 16 pages accessed
With a 1-page buffer, ~11 pages accessed

Figure 4 Iteration Algorithm

(Name of) Node Finalized	Nodes retrieved	Pages accessed by Traversal_Step
(initialize)	1	A
1	2 3 5	A,B
2	4	A
3	5	B
4	5	B
5	6-8	B
6	9-12	C
7	13-16	D
8	17-20	E

Traversal Totals 19 nodes retrieved, 5 pages accessed

Figure 5 One-Pass Traversal Algorithm (using order 1 2)

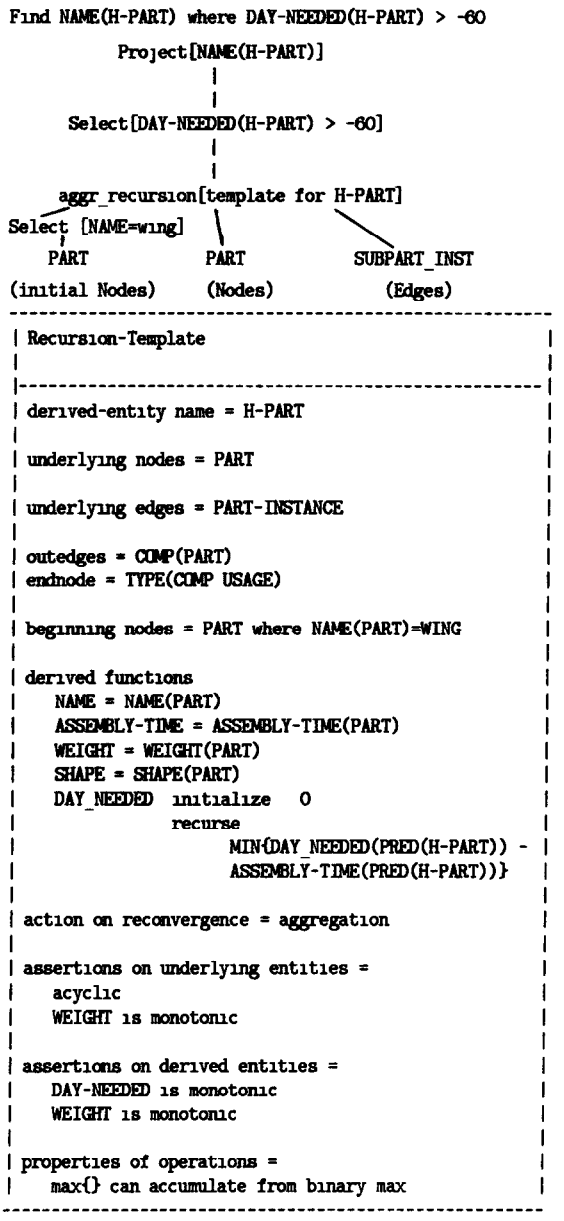


Figure 6 Result of Semantic Analysis

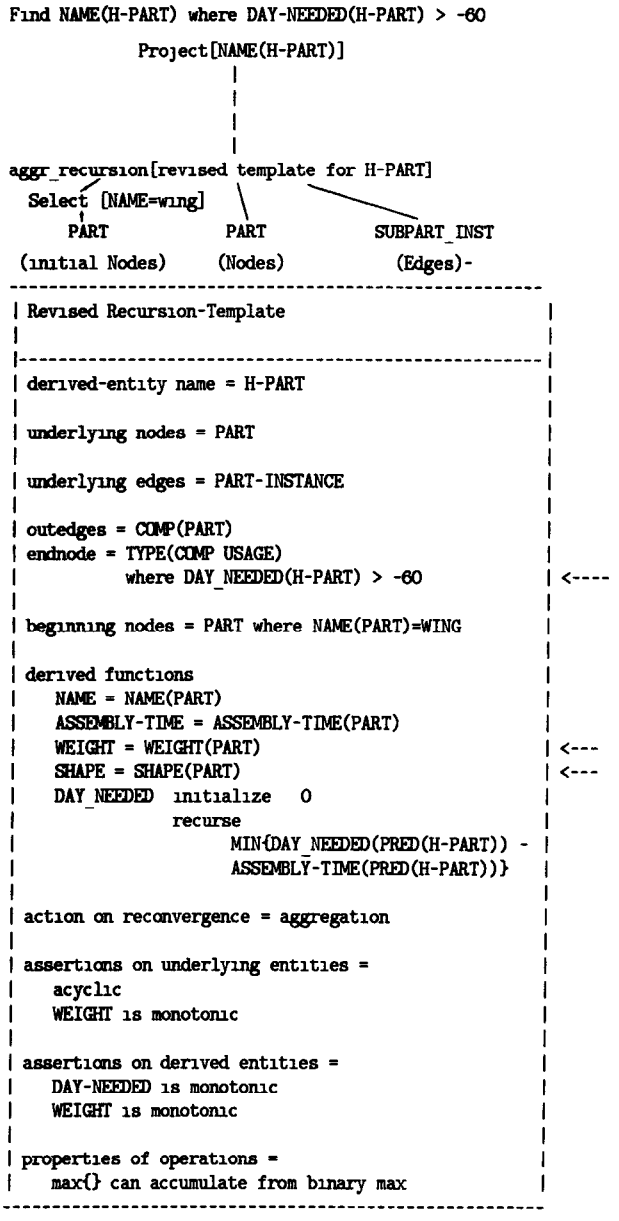


Figure 7 Result of Logical Transformations

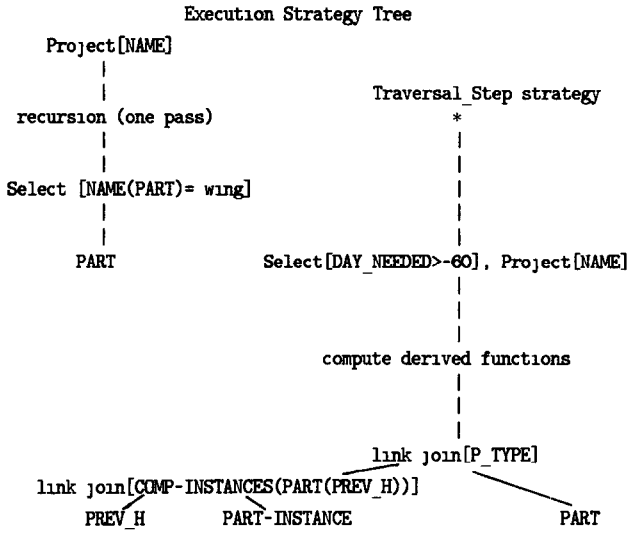


Figure 8 Execution Strategy Tree