A File Structure Supporting Traversal Recursion

P.-A. Larson and V. Deshpande

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

ABSTRACT

Traversal recursion is a class of recursive queries where the evaluation of the query involves traversal of a graph or a tree. This limited type of recursion arises in many applications. In this report we investigate a simple file structure that efficiently supports traversal recursion over large, acyclic graphs. The nodes of the graph are sorted in topological order and stored in a B-tree. Hence, traversal of the graph can be done in a single scan. Nodes and edges can also be inserted, deleted, and modified efficiently.

1. Introduction

Recursive queries over trees and graphs arise in many applications. The Bill of Materials problem is the canonical example but similar queries occur, for example, in planning and scheduling (task networks), genetics (inheritance), and transportation and communication (routing problems). Recent work on recursive queries has focused mainly on the problem of coupling recursive Horn clause languages like PROLOG to relational databases. Bancilhon and Ramakrishnan [BR] give an excellent overview of recursive query processing strategies. Supporting a general recursion mechanism seems like an overkill: performance is often poor even for a moderately large database and few applications appear to require full-blown recursive capability. The problem is to find a limited class of recursive queries that is sufficient to handle the needs of most applications and that can be efficiently supported.

Rosenthal et al. [RHDM] introduced a limited type of recursion called traversal recursion where the recursion consists of the traversal of a graph. The computations performed while traversing the graph differ from query to query. In this paper we study a file structure that efficiently supports one-pass traversal of a directed, acyclic graph. The basic idea is simple: store the nodes in topological order in a B-tree. All successors (predecessors) of an initial set of nodes can then be found simply by scanning forward (backward) in the file.

2. Traversal Recursion

This section gives a brief, informal overview of traversal recursion. A more extensive discussion can be found in [RHDM]. All examples will be based on the example graph in Figure 1.

The common element in traversal recursion is the traversal of a graph from a given set of initial nodes. Each node and edge is assumed to have some associated information, called node labels and edge labels, respectively. The output from the recursion is another graph containing a set of nodes and edges derived from the underlying graph. The node and edge labels of the derived graph are computed as the graph is traversed. Rosenthal et al. [RHDM] distinguish between two types of traversal recursion: recursion with path aggregation and recursion with path enumeration.

Recursion with path aggregation will produce an output graph consisting of a subset of the nodes and edges in the underlying graph, possibly with different node and edge labels. If a node is reached along more than one path, only one output node is created. In recursion with path enumeration, a node is processed once on every path. The derived graph forms a tree and new node and edge labels are computed as the recursion proceeds.

This work was supported by Cognos Inc., Ottawa and the Natural Science and Engineering Research Council of Canada.
Electronic mail: {palarson, vdeshpande}@{waterloo.edu, uwaterloo.ca}

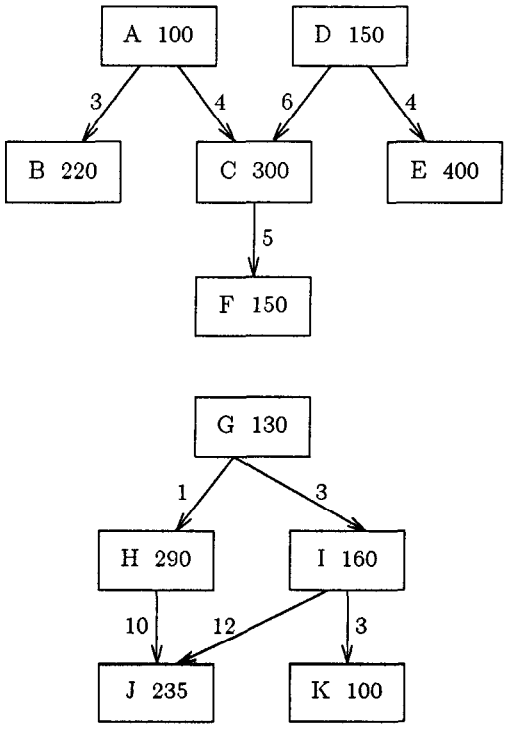

Figure 1: Example graph

Example:
Assume that the nodes of the graph in Figure 1 represent parts and that the edges refer to the components of a part. The edge labels indicate the number of "lower" components required in one "higher" component. Now consider the query: for each part required to make 10 units of part A and 20 units of part D, find the number of parts needed. This query is an example of traversal recursion with path aggregation. The number of units needed of some part can only be computed when the number of units needed of all its immediate predecessors has been computed. When a part has more than one immediate predecessor, the number of units needed is the sum of the requirements computed from all its predecessors. The resulting graph, which contains no edges, is shown in Figure 2.

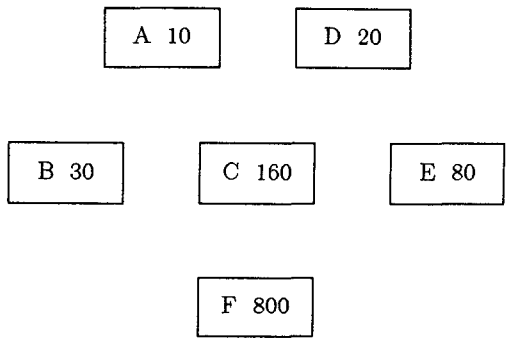

Figure 2: Result of the the example query.

3. File Organization

The design of a storage structure for some set of data must take into account what operations we want to perform on the data. Our objective is to design a storage structure for (large) directed acyclic graphs that efficiently supports the following operations:

- finding all successors (predecessors) of a given set of nodes
- retrieval of nodes and edges
- updating (insertion, deletion, modification) of nodes and edges

Naive storage of a graph may result in very poor performance. Banerjee et al. [BKKG] studied a file organization for directed acyclic graphs involving different possible clustering sequences. Each clustering sequence ensures that all descendants of a node are fetched in a single forward scan of the file. They do not allow for arbitrary insertion and deletion of edges, or for arbitrary deletion of nodes. In particular, the insertion of arbitrary edges forces extensive changes in the clustering sequence, which we have handled. In a relational database, a graph would typically be represented by two relations: one relation containing the nodes and one containing the edges. A two-relation representation of our example graph is shown in Figure 3.

Retrieval and updates of nodes and edges are straightforward when using this representation. The cost depends on how the relations are stored but it can be made quite low. The main drawback is the cost of traversal. Two basic traversal strategies are repeated joins and traditional graph search. Other strategies are described in [BR].

Repeated joins:
This strategy is also known as Naive Evaluation. In this approach, the result of our example query would

Nodes

Node Key	Node Data
A	100
B	220
C	300
D	150
E	400
F	150
G	130
H	290
I	160
J	235
K	100

Edges

Start Node	End Node	Edge Data
A	B	3
A	C	4
D	C	6
D	E	4
G	H	1
G	I	3
C	F	5
H	J	10
I	J	12
I	K	3

Figure 3: Two-relation representation of the example graph.

be computed by first selecting nodes A and D from the Nodes relation and then performing a sequence of join pairs. The two joins consist of joining the result obtained so far with the Edge relation over NodeKey and StartNode and then joining the result with the Node relation over NodeKey and EndNode. In essence, a pair of joins obtains the immediate successors of the nodes already in the result. This approach is efficient when the initial set of nodes is relatively large and the longest path from an initial node to a successor is short. Under those circumstances, the number of joins required is small and each join "pulls" in many new nodes.

Traditional graph search:
Traditional graph traversal algorithms (depth-first search, breadth-first search, level search) developed for in-core graphs can be used for graphs on external storage as well. For this approach to be reasonable, the two relations must be organized in such a way that we can quickly retrieve a node given its NodeKey value and all edges with a given value for StartNode. This can be achieved, for example, by storing the nodes in a hash file and the edges in a B-tree using (StartNode, EndNode) as the primary key. This approach is efficient when the initial set of nodes is small and the number of immediate successors (fanout) is small. Note, however, that it typically requires several disk accesses per node retrieved.

Traversal can be done by a one-pass scan if the nodes are stored in topological order. The nodes of a graph are in topological order if, for every node x, all successors of x occur after x in the ordering. There are typically several valid topological orderings of a graph. (For more details on graphs and graph algorithms, see [RND].) If the graph is acyclic, a topological order always exists. To speed up traversal by exploiting this property, we propose a storage structure that consists of two files: a main file and an index file.

Main file

Order Key	Node	Successors	Predecessors
#10	A 100	(#40,3),(#50,4)	
#20	D 150	(#50,6),(#60,4)	
#30	G 130	(#70,1),(#80,3)	
#40	B 220		(#10,3)
#50	C 300	(#90,5)	(#10,4),(#20,6)
#60	E 400		(#20,4)
#70	H 290	(#100,10)	(#30,1)
#80	I 160	(#100,12),(#110,3)	(#30,3)
#90	F 150		(#50,5)
#100	J 235		(#70,10),(#80,12)
#110	K 100		(#80,3)

Index

Node Key	Order Key
A	#10
B	#40
C	#50
D	#20
E	#60
F	#90
G	#30
H	#70
I	#80
J	#100
K	#110

Figure 4: The example graph stored using the proposed organization.

1. *Main file*
   Both the nodes and the edges of the graph are stored in the main file. A record consists of a node, its outgoing edges (successor edges), and its incoming edges (predecessor edges). Each node (record) is assigned an artificial *ordering key* (OrderKey) and the records are stored in a B-tree based on this key. The ordering keys are assigned in such a way that the records will be stored in topological order. An edge is represented by the ordering key of its end node (start node) and the

data associated with the edge. Every edge is stored twice, once as a successor edge and once as a predecessor edge. Note that records will be of variable length.

2. *Index*
   The index keeps track of the association between the NodeKey value of a node and its assigned OrderKey value. A record simply consists of a (NodeKey, OrderKey) pair. The file is organized on NodeKey, either as a hash file or a B-tree. Given the NodeKey value of a node, we can quickly look up its OrderKey value in the index.

Figure 4 shows how our example graph would be stored using this organization. The topological ordering used is one obtained by a level-order search of the graph. Records are shown as being stored two to a page; in practice it would be much higher, especially in the index.

4. File Creation

To create the main file we must assign the ordering keys, build the records, and load the B-tree. To create the index, we must extract (NodeKey, OrderKey) pairs and load the index file. In this section we outline a procedure for creation of the main file. We assume that the graph is initially given as two sequential files: a node file and an edge file, with the fields shown in Figure 3.

Assigning the ordering keys amounts to a topological sort of the nodes. For this we propose to use a traditional graph search algorithm, combined with a compact representation of the graph in main memory. Here is a basic algorithm for topological sorting.

```
Procedure TopologicalSort( V : set of
nodes;  E : set if edges)

AUX : set of nodes ;

AUX := empty set ;
for each node y in V do
  if y has no incoming edges in E then
    add y to AUX ;
    delete y from V ;
  endif
endfor

while AUX is not empty do
  x := some node from AUX ;
  for each edge p in E having x as its
  start node do
    y := end node of p ;
    delete edge p from E ;
    if y now has no incoming edges in
    E then
      add y to AUX ;
      delete y from V ;
    endif
  endfor
  output x ;
  delete x from AUX ;
endwhile

if V is not empty then error
( the graph contains a cycle ) ;

end ;
```

To implement this algorithm, we must decide how to organize AUX and how to store nodes and edges. If AUX is implemented as a stack, the algorithm will perform a depth-first search. If it is implemented as a queue, the search will be breadth-first. For the algorithm to run efficiently, we must be able to rapidly find all the outgoing edges of a node, and rapidly decide whether a node has any (unprocessed) incoming edges. A data structure proposed by A. B. Kahn [Ka] in 1962 achieves this and, in addition, is very compact. The structure is illustrated in Figure 5. Only the data inside the boxes is actually stored; the data outside the boxes was included in the figure to simplify the explanation of the structure.

Nodes

	Edge Count	First Edge
A	0	1
B	1	-1
C	2	3
D	0	4
E	1	-1
F	1	-1
G	0	6
H	1	8
I	1	9
J	2	-1
K	1	-1

Edges

	End Node	Stop Flag
A→B	2	0
A→C	3	1
C→F	6	1
D→C	3	0
D→E	5	1
G→H	8	1
G→I	9	1
H→J	10	1
I→J	10	0
I→K	11	1

Figure 5: Compact storage of the example graph for topological sorting

Assume that all nodes have been numbered from one up and that the edges are also expressed using node numbers. First sort the nodes in ascending order on node number and sort the edges in ascending order on the node number of the start node of the edge. The graph can be stored in main memory using two arrays: one for nodes and one for edges. The first entry in the node array corresponds to node one, the second to node two, etc. All edges having the same start node are stored as a block in the edge array. The field EndNode contains the node number of the end node of the edge. A stop flag indicates the last edge in a block of edges having the same start node. The field EdgeCount counts the number of incoming edges. The field FirstEdge points to the first outgoing edge of the node. Some designated value (here shown as -1) is used for indicating that the node has no outgoing edges. The node table is filled in as the edge table is built.

By using this representation, large graphs can be stored in main memory. An entry in the node table requires no more than six bytes (two for EdgeCount, four for FirstNode). An edge needs only four bytes (using the sign bit as the StopFlag). A graph with one million nodes and one million edges can be stored in 10 Megabytes. The main loop of the sort algorithm will also be very fast. Given a node number, we can immediately find all its outgoing edges by retrieving the pointer from the node table. We then loop through the outgoing edges (stopping as soon as we find StopFlag set) decrementing the edge count of the node pointed to by the edge. If the edge count of a node becomes zero, we immediately add the node to the set AUX. The running time of the algorithm is linear in the number of edges and the number of nodes. (The observed CPU time for a graph of 5000 nodes and 15000 edges was around 12 sec on a 16 MHz Intel 80386 processor.)

The topological sorting algorithm is the core of a procedure for building the main file and the index. However, considerable pre- and postprocessing is required. A complete procedure for building the main file and the index is outlined below. This rather complex procedure is necessary when the graph does not fit in main memory.

```
Procedure CreateFile( V : node file,
E : edge file )

Phase 1: (Assign Node# values)
 Sort V on NodeKey, creating two
 output files: V' := file of (Node#,
 NodeData), VS := file of (Node#,
 NodeKey). The records in V' and VS
 are assigned Node# values in
 ascending order, starting from one.

Phase 2: (Convert NodeKey values to
Node# values in the edge file)
 Sort E on EndNode, creating file T.
 Merge-join VS and T (over NodeKey
 and EndNode), creating file T'
 of (StartNode, EndNode#, EdgeData).
 Sort T' on StartNode, creating file S.
 Merge-join VS and S (over NodeKey
 and StartNode) creating file S'
 of (StartNode#, EndNode#, EdgeData).
 Sort S' on EndNode#, creating file
 S''.

Phase 3: (Topological sort)
 Scan S' building the internal
 representation of the graph. Run the
 topological sorting algorithm. Use
 the field FirstEdge to record the
 OrderKey assigned to a node. Keep in-
 core node table, delete edge table.

Phase 4: (Build the main file)
 Scan V', S' and S'' simultaneously
 building the main file. Create one
 main file record for each record
 in V' obtaining the required
 information as follows: Node# and
 NodeData from V', successor edges
 from the fields EndNode# and EdgeData
 of a sequence of records in S', and
 predecessor edges from the fields
 StartNode# and EdgeData of a sequence
 of records in S''. Translate all
 Node# values to OrderKey values using
 the in-core table built in phase 3.

Phase 5: (Build the index)
  Scan VS building the index file.
  Use the in-core table built in
  phase 3 to translate Node# values
  to OrderKey values.

End.
```

Finally, some remarks about assigning ordering values. In the experiments reported in section 7, ordering keys were 32 bit integers. When the topological ordering was computed, ordering key values were assigned at equal intervals over the range representable by 32 bit integers. As a node was inserted or moved and a new ordering key value had to be assigned, it was chosen halfway between two existing ordering key values. If there are many new nodes inserted between two existing ones, we may be unable to find a free ordering key value because of the limited precision provided by a fixed-length representation. This can be overcome by representing ordering keys by variable length strings, which provide unlimited precision.

5. Retrieval Operations

The proposed file organization efficiently supports the following retrieval operations: random retrieval of a node or an edge, complete scanning of all nodes and edges, and finding all successors (or predecessors) of a given set of nodes.

Random retrieval:
To retrieve a node given its NodeKey value, we first look up the corresponding OrderKey value in the index. Using this value, we can then retrieve the desired record from the main file. To retrieve an edge given values for StartNode and EndNode, we first look up the OrderKey values corresponding to the StartNode and EndNode values. Using the OrderKey value of StartNode, we then retrieve the record from the main file and extract the desired edge from the set of successor edges stored in the record. Note that we can also rapidly find all the incoming or outgoing edges of a node. They are all stored in the record corresponding to the node.

Complete scanning:
A complete scan of all nodes trivially translates into a complete scan of all the records in the main file. Note that the records will be obtained in sorted order on OrderKey. A complete scan of the edges can be done by scanning all records in the file and extracting all successor edges.

Finding all successors (predecessors):
This is the basic operation in traversal recursion and the raison d'ètre for the file organization. The basic algorithm for finding all successors follows. It makes use of a priority queue for keeping track of which nodes to visit. The operation "GetFirst" retrieves the element with the lowest OrderKey value from the queue and also deletes it from the queue.

```
Procedure FindSuccessors
( M : set of NodeKey values )

Q : priority queue of OrderKey values ;

for each x in M do
  y := OrderKey value corresponding to
  (retrieved from the index) ;
  put y on Q ;
endfor

while Q is not empty do

  y := GetFirst( Q ) ;
  r := get record with OrderKey y from
  the main file ;
  for each s in the successor set of r
    do put s on Q ;
  endfor

  output r ;

endwhile

End.
```

This basic algorithm can be modified for various purposes by including additional processing. Instead of just outputting the record, further processing can be done at that point. Not all successor edges have to be placed on the queue; a subset may be selected by some condition. Additional information can be carried in the entries of the priority queue. (This would be required to evaluate the example query in section two.) Whether or not the queue permits duplicate entries makes an important difference. If duplicate entries are allowed, the same node may be reached several times, as required by recursion with path enumeration [RHDM].

6. File Maintenance

There are six basic file maintenance operations: insert, delete, or modify either a node or an edge. By modification we mean modification of the data associated with the node or edge, not modification of the NodeKey values. We assume that a node to be updated is identified by its NodeKey value and an edge by its StartNode and EndNode values. It turns out that five of the six operations are straightforward and fast. The exception is insertion of an edge. Insertion of an edge may render the current topological ordering invalid and force a rearrangement of some records. We will briefly comment on each operation; an algorithm sketch is provided only for

insertion of an edge.

Modification of a node:
Using the NodeKey value of the node, look up its OrderKey value in the index. Then retrieve the required record from the main file, modify and rewrite it.

Deletion of a node:
A node to be deleted may still have incoming and outgoing edges, which must also be deleted. Look up the OrderKey value of the node in the index. Then retrieve the required record from the main file and save the OrderKey values of its immediate successors and its immediate predecessors. Remove the record from the file. For each immediate predecessor record, delete the (successor) edge pointing to the deleted record. For each immediate successor record, delete the (predecessor) edge pointing to the deleted record. Finally, delete the node from the index file.

Insertion of a node:
A new node can be assigned any (free) ordering key value as there cannot be any edges incident on the node.

Modification of an edge:
Assuming that an edge is identified by the NodeKey values of its start node and end node, we first have to look up the OrderKey value of the start node in the index. We can then retrieve the record corresponding to the start node of the edge, modify the information associated with the edge, save the OrderKey value of the end node of the edge, and rewrite the record. Using the saved OrderKey value, we then retrieve the record corresponding to the end node and modify that copy of the edge information.

Deletion of an edge:
The procedure is analogous to the procedure for modifying an edge.

Insertion of an edge:
This is the most expensive operation because we may have to move records, assign them new ordering keys, and modify edges. This occurs when the new edge invalidates the current topological ordering. An outline of the algorithm follows.

```
Procedure InsertEdge
( X->Y : edge from X to Y )

Look up the ordering keys of X and Y
in the index. Denote these by
X.K and Y.K, respectively.

Case 1: (Forward edge)
If X.K < Y.K then

  Add the edge X->Y as a successor
  edge to the record with ordering key
  X.K.

  Add the edge X->Y as a predecessor
  edge to the record
  with ordering key Y.K.

endif

Case 2: (Backward edge, current
ordering invalid)
If X.K > Y.K then

  Retrieve the record with key Y.K and
  all its successors with an ordering
  key less then or equal to X.K.
  Call this set of nodes M.

  If X is in M then terminate without
  inserting the edge. (Inserting it
  would cause a cycle in the graph.)

  Assign every record R in M a
  new ordering key R.K' such
  that R.K' is between X.K and the
  ordering key of the record
  immediately after X. Record the
  following information in a
  temporary table T: NodeKey of R,
  old ordering key (R.K), and
  new ordering key (R.K').

  Scan the records in M, computing two
  sets of (old) ordering keys: S and P.
  S is the ordering keys of all
  successor edges of records in M.
  P is the ordering  keys of all
  predecessor edges of records in M.

  For each record R in M do

    Remove R from the main file
    (using the old ordering key).
    Change the ordering key of R to
    the new value (found in T).

    For every successor and
    predecessor edge in R with
    an (old) ordering key value
    between X.K and Y.K, change
    the ordering key to the new
    value found in T.
```

```
        Reinsert the modified version of R
        into the file.

    endfor

    For every entry in P such that
    R.K < X.K, retrieve the record and
    for every successor edge with a key
    between X.K and Y.K, change
    it to the new value found in T,
    and rewrite the record.

    For every entry in S such that
    R.K > Y.K, retrieve the record
    and for every predecessor edge with
    a key between X.K and Y.K,
    change it to the new value found in
    T, and rewrite the record.

    For every entry in T, retrieve the
    corresponding record in the index,
    change the ordering key to the new
    value, and rewrite the record.

endif
End.
```

7. Preliminary Performance Results

To get some indication of the performance of the proposed file structure, we ran several experiments on randomly generated graphs. We were particularly interested in the cost of finding all successors of a node, the cost of inserting an edge and how they were affected by the topological ordering used.

To run the experiments, we implemented the file structures and operations for file creation, finding all successors, and edge insertion. Both the main file and the index were implemented as B-trees (stored in main memory). Random acyclic graphs were generated as follows. The nodes were first allocated and assigned keys from 1 to n. To generate an edge, we generated two random numbers in the range 1 to n: one selected the start node and the other one selected the end node. To ensure that the graph remained acyclic, the lower number was taken to be the start node and the higher number the end node. Hence, there were no backward edges and the graph was guaranteed to be acyclic. The nodes and the edges were then input (in random order) to the algorithm for topological sorting explained in section 4. We tried two different topological orderings: breadth-first (AUX implemented as a queue) and depth-first (AUX implemented as a stack).

To estimate the cost of finding all successors of a node, we randomly selected a start node, retrieved all its successors and collected statistics. This was repeated 20 times and averages computed. The experiment was then repeated using four different page sizes for the B-trees. The results are reported in Tables 1 to 4.

	Page size			
	256	512	1024	2048
Records/data page	4.20	9.35	19.61	38.46
Index fanout	7.04	15.87	29.41	58.82
Data pages/ successor	.485	.273	.162	.102
Index pages/ successor	.166	.093	.047	.035
Total pages/ successor	.651	.366	.209	.137

1000 nodes, 3000 edges, one component, breadth-first ordering. 106 successor nodes on average, queue size 47.3 nodes.

Table 1: Cost of retrieving all successors of a node

The results in Table 1 are for a graph containing 1000 nodes and 3000 edges. The topological ordering was generated by a breadth-first traversal. The average number of successors of a node was 106 nodes. The maximum size of the priority queue used during traversal was 47.3 nodes (averaged over the 20 traversals). These two statistics do not depend on the page size.

The cost of finding all successors of a node was measured by the number of data pages and the number of index pages read per successor node retrieved. As shown in Table 1, the cost decreased rapidly with the page size. (The absolute page size is unimportant; what matters is the number of records per data page and the fanout of the B-tree index.) Even with data pages holding less than 5 records each, the number of disk reads was less than one per successor node. When data pages held close to 40 records each, the cost decreased to less than 0.15 disk reads per successor node.

Topological orderings produced by a breadth-first search do not automatically preserve locality, that is, nodes in the same independent component may not be clustered together. Topological orderings produced by a depth-first search tend to preserve locality and cluster nodes from the same component

into a few groups. To verify this behaviour, we repeated the same experiments but using depth-first ordering. To our surprise, the result were no better than the results in Table 1. This appears to be caused by the high ratio of edges to nodes (3:1). When the graph is highly connected and contains only one or a few components, there seems to be no difference between the two topological orderings.

In subsequent experiments, we generated graphs with 100 independent components. In essence, this was done by generating 100 different graphs, each containing 50 nodes, and then supplying the union of these graphs as input to the topological sort. The results from these experiments are shown in Tables 2 and 3. (The decrease in successor nodes is expected.) These results confirmed the hypothesis that depth-first ordering achieves a better clustering than breadth-first ordering and improves the retrieval performance.

	Page size			
	256	512	1024	2048
Records per data page	3.72	8.33	17.54	35.97
Index fanout	6.90	14.05	27.03	56.18
Data pages per successor	.621	.527	.473	.443
Index pages per successor	.962	.782	.593	.459
Total pages per successor	1.583	1.309	1.066	.902

5000 nodes, 15000 edges, 100 components, breadth-first ordering. 25.05 successor nodes on average, queue size 11.7 nodes.

Table 2: Cost of retrieving all successors of a node

We also ran several experiments measuring the cost of edge insertion and the effect on retrieval performance of modifications to the graph. For each experiment, we first created a random graph as described above, determined the ordering keys, and created the main file and index file. The records were inserted into the main file in random order to create a stable B-tree. This was followed by a series of modification cycles, where each cycle consisted of 100 insert/delete operations followed by 50 random searches. An insert/delete pair consisted of inserting a randomly generated edge and deleting a randomly selected edge. (Edges that would have created a cycle were rejected.) The insertions were done in this way

	Page size			
	256	512	1024	2048
Records per data page	3.73	8.35	17.48	35.71
Index fanout	6.69	14.16	25.64	49.50
Data pages per successor	.447	.255	.154	.100
Index pages per successor	.250	.200	.142	.130
Total pages per successor	.697	.455	.296	.230

5000 nodes, 15000 edges, 100 components, depth-first ordering. 25.05 successor nodes on average, queue size 11.65 nodes.

Table 3: Cost of retrieving all successors of a node

to keep the graph statistically unchanged, i.e. the same number of nodes and edges.

The first observation was that retrieval performance seemed not to be affected by edge insertions and deletions. Even after several thousand insert/delete pairs, the cost of retrieving all successors of a node remained about the same; no consistent trend was observed. This was true both for depth-first and breadth-first orderings. Hence, we concluded that the edge insertion algorithm does not cause a deterioration in retrieval performance.

The statistics for edge insertion are summarized in Table 4. The figures are averages over 1000 insertions. As expected, the number of disk accesses decreases rapidly when the page size increases. The fact that more than 60% of the edges were forward edges is probably caused by some backward edges being rejected, i.e. those that would have created a cycle. "Records moved" is the average number of records per insertion that had to be moved and assigned new ordering keys. "Records changed" is the average number of records per insertion whose successor or predecessor edges had to be modified.

8. Concluding Remarks

Overall, we consider the preliminary performance results encouraging. However, our experiments are far too limited for any firm conclusions. Further work is needed in two areas: (experimental) analysis of the performance for a wider range of parameters and investigation of heuristics for obtaining better topological orderings. Various characteristics of the graph (ratio of edges to nodes, number and size of independent components) seem to

	Page size			
	256	512	1024	2048
Records per data page	2.47	5.61	12.47	25.30
Data page reads/writes	6.82	5.50	4.40	3.98
Index page reads/writes	11.40	9.92	8.36	6.34
Percent forward edges	61.4			
Records moved	1.45			
Records changed	10.87			

5000 nodes, 15000 edges, 100 components, 500 edge insert/delete pairs.

Table 4: Statistics for edge insertions.

have a significant effect on the performance. There are typically many valid topological orderings, some better than others for traversal purposes. At this point, we do not know how the ordering affects the performance. Is there a wide difference between the performance obtained by using "standard" topological orderings and that obtained by an optimal ordering? How much room is there for improvement? The results in [ES] indicate that finding an optimal topological ordering is likely to be NP-complete for most "reasonable" cost measures. It seems that we have to concentrate on finding simple heuristics that produce good, but not necessarily optimal, topological orderings.

References

[BKKG] J. Banerjee, W. Kim, S. Kim and J.F. Garza, Clustering a DAB for CAD Databases, IEEE Transactions on Software Engineering, Vol. 14, No. 11, 1988, 1684-1699

[BR] F. Bancilhon and R. Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies, Proc. SIGMOD Conf. on Management of Data, 1986, 16-52

[ES] S. Even and Y. Shiloach, NP-completeness of Several Arrangement Problems, Technical Report No. 43, Weizmann Inst. of Science, Rehovot, Israel, 1975

[Ka] A. B. Kahn, Topological Sorting of Large Networks, Comm. of the ACM, Vol. 5, 1962, 558-562

[RHDM] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications, Proc. SIGMOD Conf. on Management of Data, 1986, 166-176

[RND] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, NJ, 1982