Parallel Algorithms for the Generalized Same Generation Query in Deductive Databases

Nabil Arman
Faculty of Computer Science
Palestine Polytechnic University
Hebron, Palestine
narman@ppu.edu

Journal of Digital Information Management

ABSTRACT: *The intelligence of traditional database systems can be improved by recursion. Using recursion, relational database systems are extended into knowledge-base systems (deductive database systems). Linear recursion is the most frequently found type of recursion in deductive databases. Deductive databases queries are computationally intensive and lend themselves naturally to parallelization to speed up the solution of such queries. In this paper, parallel algorithms to solve the generalized fully and partially instantiated forms of the same generation query in deductive databases are presented. The algorithms use special data structures, namely, a special matrix that stores paths from source nodes of the graph representing a two-attribute normalized database relation to all nodes reachable from these source nodes, and a reverse matrix that stores paths from any node to all source nodes related to that node.*

Categories and Subject Descriptors

C.2.1 [Network architecture and design]: Distributed networks**; H.3.4 [**Systems and Software**]:** Information networks

General Terms
Peer to peer networks, Content search

Keywords: Deductive Databases, Linear Recursive Rules, Same Generation Query, Parallel Databases

Received 13 Dec. 2006; Reviewed and accepted 12 Jan. 2007

1. Introduction

The development of efficient algorithms to process recursive rules and queries within the context of large database systems has recently attracted a large amount of research efforts due to the important role of recursive rules in improving the intelligence of database systems and extending them into knowledge-base systems [1,2,3,4,5,6,7,8]. One of the main features of these intelligent database systems, namely deductive databases, is their ability to define recursive rules and to process queries on them directly.

Many queries in deductive databases, including the same generation query, have data requirements that may run into terabytes. Handling such large volumes of data at an acceptable rate is difficult, if not impossible, using single-processor systems. In fact, a set of commercial parallel database systems, such as Teradata DBC series of computers have demonstrated the feasibility of parallel database queries. As a matter of fact, the set-oriented nature of database queries naturally lends itself to parallelization [9].

In deductive databases, most recursive rules appear in a simple form in which the rule's head appears only once in the body [3]. In general, this type of logic rules are called **linearly recursive**. A same generation (*sg*) rule is a linearly recursive rule of the following form:

$$sg(X_1, X_2,..., X_n) :- par(Y_1, X_2)\ par(Y_2, X_2),...,$$
$$par(Y_n, X_n) :- sg(Y_1, Y_2,... Yn)$$

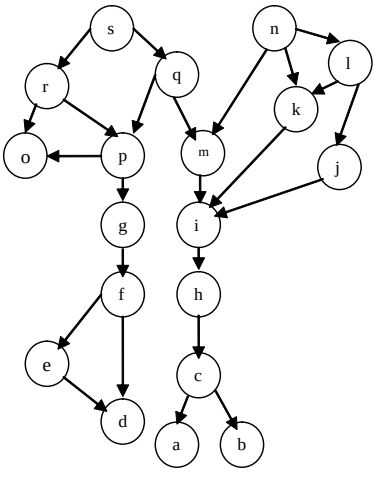
where "*par*" is an extensional (base) predicate and "*sg*" is an intentional database predicate. Within the context of deductive databases, the extensional database predicate "*par*" is defined by a two-attribute normalized database relation with very many tuples as shown in Figure 1 (a) [3,4]. Another common view for the base relation is represented by a directed graph, as shown in Figure 1 (b). For every tuple $<x,y>$ of the base relation, there exists, in the corresponding graph, a directed edge from node *x* to node *y*. The nodes in such a graph are the set of distinct values in the two columns of the base relation (i.e., the domain). To generate solutions from the above recursive rule, another non-recursive rule, the exit rule, which defines the predicate "$sg(X_1, X_2,..., X_n)$" must exist. This non-recursive rule is given by:

$$sg(X_1, X_2,..., X_n) :- par(Y_1, X_2)\ par(Y_2, X_2),...,$$
$$par(Y_n, X_n)$$

A query on a predicate that is defined by the recursive and the exit rule is called a same generation query. This query is a headless rule of the following form:

$$sg(X_1, X_2,..., X_n)$$

X	Y
s	r
s	q
r	o
r	p
q	p
q	m
p	o
p	g
n	m
n	k
n	l
m	g
m	i
l	k
l	j
k	i
j	i
i	h
h	c
g	f
f	e
f	d
e	d
c	a
c	b



(b) GRAPH FORM

Figure 1. THE BINARY RELATION "par" IN (a) TABLE FORM (b) GRAPH FORM

A query typically involves a predicate symbol with some variable arguments, and its meaning or answer is the different constant combinations that when bound (assigned) to the variables, can make the predicate true. In general, an *n*-place unit query, such as the above one, may have different forms depending on the instantiation status of the variables [5]. In this article, we propose a parallel algorithm for solving the generalized fully instantiated form of the same generation query, i.e., a query that has the form:

$$sg(C_1, C_2,..., C_n)$$

where $C_1, C_2,..., C_n$ are constants representing nodes in the graph. The order of the arguments is irrelevant since "*sg*" is a symmetric relation. Let the instantiated set of nodes (ISN) be {}, then the answer of such a query is either TRUE if are at the same generation (i.e., the set of nodes are on the same level of a family tree), or FALSE if are not at the same generation (i.e., the set of nodes are not on the same level of a family tree).

	0	1	2	3	4	5	6
0	s	r	o				
1			p	0,2			
2				g	f	e	d
3						2,6	
4			1,2				
5			m	2,3			
6			q	i	h	c	a
7							b
8	n	5,2					
9		k	6,3				
10		l	9,1				
11			j	6,3			

	0	1	2	3	4	5	6
0	o	r	s				
1		p	0,1				
2			q	0,2			
3	d	e	f	g	1,1		
4					m	2,2	
5						n	
6		3,2					
7	a	c	h	i	4,4		
8					k	5,5	
9						1	5,5
10					j	9,5	
11	b	7,1					

0,0	0,1	0,2	1,2	1,3	2,3	2,4	2,5	2,6	3,5	4,1	4,2	5,2	5,3	6,3	6,4	6,5
s	r	o	p	0,2	g	f	e	d	2,6	q	1,2	m	2,3	i	h	c

6,6	7,6	8,0	8,1	9,1	9,2	10,1	10,2	11,2	11,3
a	b	n	5,2	k	6,3	l	9,1	j	6,3

Figure 3. The Matrix as linear array

In addition, the paper presents another algorithm for solving the generalized partially instantiated form of the same generation query, i.e., a query that has the form:

$$:- sg(X_1,X_2,...,X_i, c_{i+1}, c_{i+2},...,c_n)$$

where $X_1,X_2,...,X_i$ are the uninstantiated variables whose values are to be determined and $c_{i+1}, c_{i+2},...,c_n$ are constants representing nodes in the graph. The order of the arguments is irrelevant since "*sg*" is a symmetric relation. Let the uninstantiated set of nodes (USN) be {} and the instantiated set of nodes (ISN) be {}, then the answer of such a query is the set of nodes with a cardinality of *i* that are of the same generation as (i.e., the set of nodes that are on the same level of a family tree with $c_{i+1}, c_{i+2},...,c_n$).

2. The structure used in the algorithms

The structure used in the algorithms is a special matrix. This structure has been used in computing the transitive closure of a database relation [5], and in developing a sequential algorithm for the generalized form of the partially instantiated same generation query in deductive databases [4]. This matrix structure has been compared with other graph representation schemes. The comparison has shown that the matrix representation has more information than the other schemes [10,11]. In this matrix, the rows represent some paths in the graph starting from the source nodes to the leaves. Basically, depth-first search is used to create the paths of the graph. Instead of storing every node in all paths, the common parts of these paths can be stored only once to avoid duplications. If two

$P_1 = <a_1,a_2,...,a_n, b_1,b_2,b_m>$ and

$P2 = <a_1,a_2,...,a_n, c_1,c_2,c_m>$

have the common parts $<a_1,a_2,...a_n>$, then P_1 and P_2 can be stored in the two consecutive rows of the matrix as $<a_1,a_2,...,a_n,b_1,b_2,...b_m>$ and < — n *empty entries* — , $<c_1,c_2,...c_l>$, where the first *n* entries of the second row are empty. To prevent the duplicate storage of the nodes in the matrix, a different technique is used; for the first visit to the node, it is entered into the matrix and the coordinates of its location is recorded. On subsequent visits, instead of entering the node itself, its coordinates are entered into the matrix (a pointer to the already stored node). In this way, only a single copy of each of the graph's nodes is guaranteed to be entered in the matrix. Moreover, there will be only one entry (either a node or a pointer) in the matrix for each edge in the graph. In Figure 2 (a), the matrix representation of the graph given in Figure 1 (b) is presented. In that graph, there are 25 edges, and in its matrix representation there are 25+2 =27 nonempty entries in the matrix (another two entries for the nodes *s* and *n*). An important advantage of this matrix structure is that it stores a path from each node to all the source nodes that can reach the node. In the implementation of this sparse matrix, the empty entries are not stored explicitly. The matrix can be stored sequentially row by row as shown in Figure 3. For each row, storing the column number of its first non-empty entry and the sequence of non-empty entries in the row is sufficient. Thus, the size of the stored matrix is much smaller than the original relation and matrix. After the special matrix form is created, a (reverse) matrix, which is the matrix representation of the reverse graph, is generated using the reverse graph. Let $G=(V,E)$ be a graph, where V is a finite set of vertices/nodes and E is a finite set of arcs/edges such that each arc e in E is associated with an ordered pair of vertices/nodes v and w, written as $e=(v,w)$, then the reverse graph RG is given by $RG=(V,E')$ where V is a finite set of vertices/nodes (the same set of vertices of the original graph) and E' is a finite set of arcs such that each arc e' in E' is associated with an ordered pair of vertices w and v, written by $e'=(w,v)$ for each $e=(v,w)$ in E. The reverse matrix representation generated from the graph in Figure 1 (b) is the matrix given in Figure 2 (b). An important advantage of this matrix structure is that it stores paths from every node to the (s). For solving the same generation query, we are interested in the parents and ancestors of a certain node and not in its descendants and this information can be extracted easily from the reverse matrix (and not from the original matrix). Therefore, we need the reverse matrix representation. The reverse matrix can also be stored sequentially row by row as explained for the original matrix. In fact, there is no need even to store the whole matrix structure, because storing the row beginnings, row ends, the entries stored at the row ends, and matrix coordinates of the nodes is sufficient. This is due to the fact that we are interested in the path lengths and not in the stored nodes themselves, from the reverse matrix structure.

3. The Generalized same generation query parallel algorithms

The evaluation of the generalized same generation queries can be parallelized using intraoperation parallelism. The processing of these queries can be speeded up by parallelizing the execution of many individual operations involved in the solution of these queries. To simplify the explanation and presentation of the algorithms, it is assumed that there are *n* processors, P_1, …,P_n, and *n* disks D_1, …,D_n, where disk D_i is associated with processor P_i.

A benefit of the matrix structure is that it stores paths from the source nodes to all nodes reachable from these source nodes. This means that the nodes in the matrix are clustered on the source nodes of the graph, i.e., starting from any source, all nodes reachable from that source can be accessed. The reverse matrix structure stores paths from each node to all source nodes related to that node, which means that the nodes in the reverse matrix are clustered on the leaves of the graph, i.e., starting from any node, all source nodes related to that node can be accessed. As mentioned before, there is no need even to store the whole matrix structures, because storing the row beginnings, row ends, the entries stored at the row ends, and matrix coordinates of the nodes is sufficient. This is due

to the fact that we are interested in the path lengths and not in the actual paths and the stored nodes themselves. These structures, which are small, in size, when compared to the original structures, are replicated across all the processors.

The replicated structures can be used to solve the generalized fully instantiated same generation query. In solving such a query, the parallel algorithm proceeds as follows:

(1) Starting from each node c_i in the instantiated set of nodes (ISN), a processor P_i computes all path lengths to all relevant source nodes using the replicated structures generated from the reverse matrix structure, where each processor works on one node. During this computation, only the row beginnings and ends are used. After that, these path lengths are sorted locally by the processors in ascending order, according to the source nodes and lengths, and duplicate paths are removed.

(2) The source nodes obtained from the above step are partitioned in a round-robin technique across all the processors. Taking each source node and using the replicated structures generated from the forward matrix structure, all nodes having the same path lengths are determined by each processor. Let this set of nodes in the result be (RS) for a certain path length. In this step, only the row beginnings and row ends are also used in the computation of the paths.

3) Having all nodes (RS) collected in step (2), the algorithm checks whether all nodes in ISN are in the result i.e., ISN $\subseteq$ RS or not. If ISN RS , then the query returns TRUE, otherwise, the query returns FALSE.

The parallel algorithm for the generalized fully instantiated same generation query can be summarized as follows:

Procedure
Parallel_Generalized_Fully_Instantiated_Same_*Generation_Query()*

begin
Distribute the nodes in ISN to the available processors using a round-robin scheme
Starting from c_i in ISN, P_i computes all path lengths to all relevant source nodes using the reverse matrix structure
Each processor P_i sorts, locally, path lengths in ascending order
Each processor P_i remove duplicate paths
find source nodes that are common to all nodes in ISN using all Processors P_i
Partition source nodes using round-robin technique
Starting from s_j, P_i collects the nodes RS that are of length l from s_j
if ISN$\subseteq$ RS then
return TRUE and exit
if all relevant source nodes are considered then
return FALSE
end

Preliminary results and a draft version of this algorithm were published in [12].

The replicated structures can also be used to solve the generalized partially instantiated same generation query. In solving such a query, the parallel algorithm proceeds as follows:

(1) Starting from each node c_i in the instantiated set of nodes (ISN), a processor P_i computes all path lengths to all relevant source nodes using the replicated structures generated from the reverse matrix structure, where each processor works on one node. During this computation, only the row beginnings and ends are used. After that, these path lengths are sorted locally by the processors in ascending order, according to the source nodes and lengths, and duplicate paths are removed.

(2) The source nodes obtained from the above step are partitioned in a round-robin technique across all the processors. Taking each source node and using the replicated structures generated from the forward matrix structure, all nodes having the same path lengths are determined by each processor. Let this set of nodes in the result be (RS) for a certain path length. In this step, only the row beginnings and row ends are also used in the computation of the paths.

(3) Having all nodes (RS) collected in step (2), the algorithm makes sure that all nodes in ISN are in the result i.e., ISN RS. In addition, the number of nodes in RS-ISN should be greater than or equal to the number of nodes in USN

(i.e., |RS-ISN| $\geq$ |USN|). The result of the query will consist of all combinations of the nodes in the set RS-ISN.

The parallel algorithm for the generalized partially instantiated same generation query can be summarized as follows

Procedure
Parallel_Generalized_Partially_Instantiated_Same_Generation_Query()
begin
Distribute the nodes in ISN to the available processors using a round-robin scheme
Starting from c_i in ISN, P_i computes all path lengths to all relevant source nodes using the reverse matrix structure
Each processor P_i sorts, locally, path lengths in ascending order
Each processor P_i remove duplicate paths
find source nodes that are common to all nodes in ISN using all Processors P_i
Partition source nodes using round-robin technique
Starting from s_j, P_i collects the nodes RS that are of length l from s_j
if ISN$\subseteq$ RS and |RS-ISN|$\geq$ |USN| then
the result of the query will consist of all combinations of the nodes in RS-ISN
end

Preliminary results and a draft version of this algorithm were published in [13].

The path lengths will be sorted because the algorithms collect all the nodes in the same generation with the given node in a single step. For example, if a certain node has a set of path lengths $\{l_1, l_2, \ldots l_k\ l_1 <, l_2 < \ldots l_k\}$ from the selected query node, then all nodes that are reachable from that source node with these path lengths are collected in a single step. The duplicate paths are removed because they do not add new nodes to the solution set.

The intelligence of the algorithms is exhibited by the approach they use to answer the queries. The algorithms consider only the relevant part of the database/graph, i.e., it considers only the set of nodes that are somehow relevant to the instantiated part of the query (the nodes in ISN). In addition to that, the algorithms jump from one node to another, skipping many nodes on the paths of the underlying graph, since they only use the row beginnings and row ends of the matrices in the computation of the paths rather than the nodes of the graph themselves.

Depth-first search based techniques, such as the magic-sets rule rewriting technique and the counting technique [6], consider all source nodes of the graph. Starting from each source node, all nodes of the graph reachable from that node will be considered, even though such nodes may not be related to the instantiated set of nodes in the query. Our algorithms, on the other hand, determine the set of relevant source nodes by starting from one of the nodes in the instantiated set using the reverse graph.

Another important aspect is the benefit obtained from the parallelism of the queries' solutions. As mentioned before, intraoperation parallelism is used to speed up the execution of the algorithm.

If the number of nodes in ISN is less than the number of processors, then |ISN| number of processors are used. If the number of processors is greater than |ISN|, the nodes of ISN are assigned to processors in a round-robin scheme. If |ISN| is equal to the number of processors, then node c_i is assigned to P_i.

Example: Assume there are three processors P_1, P_2, and P_3, with three disks D_1, D_2, and D_3 respectively. For the graph in Figure 1 (b), the answer of the generalized fully instantiated same generation query

: - *sg(j,i,g)*

is computed as follows.

(1) The algorithm starts from the instantiated arguments and distributes the nodes *j,i,g* to processors P_1, P_2, and P_3 respectively. P1 starts with *j* and computes all path lengths to all relevant source nodes. P_2 and P_3 do the same for nodes *i* and *g* respectively. These paths are sorted and duplicates are removed locally on each of the processors. Thus, this step generates one path of length 2 from *j* to source node *n* using processor P_1. This step also generates two paths of length 2 and 3 from *i* to *n* using processor P_2. Finally, the step generates a path of length 3 from *g* to *n* and a path of length 2 from *g* to *s* using processor P_3.

(2) From the above step, the algorithm determines that *n* is the only relevant source node (the source node *s* is not considered in the computation since it is not common to all nodes in ISN). Therefore, the algorithm starts from *n* and uses the forward matrix structure to determine all nodes with path lengths of 2 from source node *n*. When a node of path length 2 is reached, it is recorded and the search continues until all relevant parts of the graph is traversed up to path lengths of 2 (the search terminates at this point for the current path of the graph since nodes with lengths greater than 2 are irrelevant in answering the query) or until leafs are encountered. The set of nodes in the result is RS={*g,i,k,j*}.

(3) Since ISN $\subseteq$ RS *(i.e.,* {*j,i,g*}{*g,i,k,j*}), then the answer of the query is TRUE.

The answer of the generalized partially instantiated same generation query

:- *sg(j,i,g,X)*

is computed as follows.

(1) The algorithm starts from the instantiated arguments and distributes the nodes *j,i,g* to processors P_1, P_2, and P_3 respectively. P_1 starts with *j* and computes all path lengths to all relevant source nodes. P_2 and P_3 do the same for nodes *i* and *g* respectively. These paths are sorted and duplicates are removed locally on each of the processors. Thus, this step generates one path of length 2 from *j* to source node *n* using processor P_1. This step also generates two paths of length 2 and 3 from *i* to *n* using processor P_2. Finally, the step generates a path of length 3 from *g* to *n* and a path of length 2 from *g* to *s* using processor P_3.

(2) From the above step, the algorithm determines that *n* is the only relevant source node (the source node *s* is not considered in the computation since it is not common to all nodes in ISN). Therefore, the algorithm starts from *n* and uses the forward matrix structure to determine all nodes with path lengths of 2 from source node *n*. When a node of path length 2 is reached, it is recorded and the search continues until all relevant parts of the graph is traversed up to path lengths of 2 (the search terminates at this point for the current path of the graph since nodes with lengths greater than 2 are irrelevant in answering the query) or until leafs are encountered. The set of nodes in the result is RS={*g,i,k,j*}.

(3) Based on the above, *X=k* is the only answer of the query.

4. Performance evaluation of the parallel algorithms

To determine the performance of the parallel algorithms, simulations of the algorithms were performed for random database relations with 4000 tuples of 5 different outdegree values from 1 to 5. A cluster of 3 PCs running a Linux operating system is used as a parallel machine. The implementation language is C with an MPI package. The sequential generalized same generation query algorithms were tested for 100 randomly generated queries (fully instantiated and partially instantiated). The same 100 randomly generated queries were answered using the parallel versions of the generalized same generation query algorithms. The time taken to answer these queries was determined for the sequential algorithms as well as for the parallel algorithms. These numbers were plotted for different outdegrees of the randomly generated relations as shown in Figure 4.

As can be noticed, the speedup obtained from parallelizing the generalized same generation query algorithms is sublinear. For example, when the outdegree of the graph is 5, the time for the sequential version is about 76 seconds and the time for the parallelversion is about 31 seconds. The speedup is 76/31=2.45 and that is close to the number of processors/PCs used in the simulations, which is 3. This indicates that the use of parallel algorithms for these queries is justifiable.

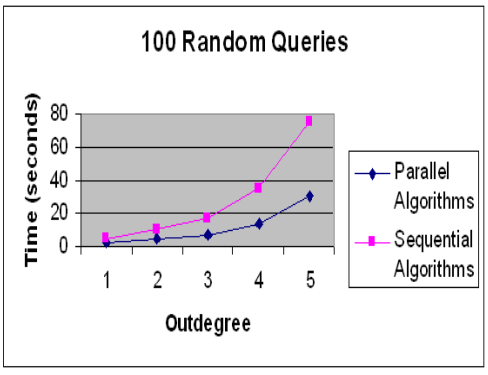


Figure 4. Performance for Parallel vs. Sequential Generalized Same Generation Query Algorithms

5. Conclusion

This paper presents parallel algorithms to solve the generalized fully instantiated same generation query and the generalized partially instantiated same generation query in deductive databases. The algorithms exhibit some intelligence by focusing on the relevant portion of the graph/database rather than considering all source nodes of the graph. In addition, the algorithms use intraoperation parallelism to execute many steps in parallel. The algorithms uses special data structures, namely, a matrix representation of the graph, representing the two-attribute normalized database relation, and a reverse matrix representation of the reverse graph. A simulation study demonstrates that the speedup obtained is sublinear. Therefore, the parallelization of the generalized same generation query algorithms is necessary, to make use of the parallel systems.

References

[1] Banchilon, F., Maire, D., Sagiv, Y., Ullman, J (1986). , Magic Sets and other Strange Ways to Implement Logic Programs, Proc. 5th ACM Symp. On Principles of Database System, 1986, pp. 1-15.

[2] Elmasri, R., Navathe, *S (2004). Fundamentals of Database Systems*, Addison Wesley.

[3] Qadah, G., Henschen, L., Kim, J (1991). Efficient Algorithms for the Instantiated Transitive Closure Queries, *IEEE Transactions on Software Engineering*, 17 (3).

[4] Arman, N (2003). An Intelligent Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases, Proceedings of the 4th International Arab Conference on Information Technology (ACIT'2003), p. 224-228, December 20-23, 2003, Arab Academy for Science and Technology, Alexandria, Egypt.

[5] Toroslu, I.,Qadah, G., Henschen, L (1994). An Efficient Database Transitive Closure Algorithm, *Journal of Applied Intelligence,* 4, pp. 205-218.

[6] Ullman, *J (1989)Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1989.

[7] Young, C., Kim, H., Henschen, L., Han, J (1992). Classification and Compilation of Linear Recursive Queries in Deductive Databases, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 1, 1992.

[8] Arman, N (2004). An Efficient Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases, *The International Arab Journal of Information Technology*, 1(1) 142-146.

[9] Silberschatz, A., Korth, H., Sudarshan, *S (2005). Database System Concepts*, McGraw Hill.

[10] Arman, N (2005). Graph Representation Comparative Study, Proceedings of the 2005 International Conference on Foundations of Computer Science (FCS'05), June 27-30, 2005, Las Vegas, USA.

[11] Arman, N (2005). Graph Representation: Comparative Study and Performance Evaluation, *Journal of Information Technology*, 4 (4) 465-468.

[12] Arman, N (2006). A Parallel Algorithm for the Generalized Fully Instantiated Same Generation Query in Deductive Databases," *In*: Proceedings of the 4th International Multiconference on Computer Science and Information Technology (CSIT2006), ISBN: 9957 - 8592 - 0 –X. Vol. 1, pp. 281-288, April 5-7, 2006, Applied Science University, Amman, Jordan.

[13] Arman, N (2006). A Parallel Algorithm for the Generalized Partially Instantiated Same Generation Query in Deductive Databases, *In:* Proceedings of the 2006 International Conference on Information and Knowledge Engineering (IKE'06), June 26-29, 2006, Las Vegas, USA.

Nabil Arman received his BS in computer science with high honours from Yarmouk University, Jordan in 1990, his MS in computer science from The American University of Washington DC, USA in 1997, and his PhD from the School of Information Technology and Engineering, George Mason University, Virginia, USA in 2000. Currently, he is an associate professor at Palestine Polytechnic University, Hebron, Palestine. His research interests include database and knowledge-base systems, and algorithms.