Journal of Information Technology Education **Volume 3, 2004**

Teaching Structured Design of Network Algorithms in Enhanced Versions of SQL

Bert de Brock
University of Groningen, Groningen, The Netherlands

e.o.de.brock@bdk.rug.nl

Executive Summary

From time to time developers of (database) applications will encounter, explicitly or implicitly, structures such as trees, graphs, and networks. Such applications can, for instance, relate to bills of material, organization charts, networks of (rail)roads, networks of conduit pipes (e.g., plumbing, electricity), telecom networks, and data dictionaries. Algorithms on such data structures often require recursion or iteration where the number of repetitions is unknown a priori. Such algorithms are usually implemented in a third generation language (3GL) and, therefore, are typically "record-at-a-time". A vast amount of theoretical work on recursive queries in logical languages (and related problems in research prototypes) is available, but these "extensions" typically are not available in commercial database management systems. Hence, they do not directly help the database developer "in the field" who has available only "ordinary" SQL with a few enhancements.

Extensions of SQL with assignments and "control of flow" constructions such as the **while**-loop enable database developers to manage and solve such graph problems more completely and compactly on a 4GL-level in their daily work. Such SQL-extensions have existed for some time in several commercially available database management systems. Incorporating this 4GL-approach in the educational field constitutes a challenge as well as an opportunity, as we show in this paper. We also illustrate various classical aspects of algorithm design at 4GL-level.

In this paper we elaborate on the idea of graph algorithms on 4GL-level. In the Introduction we give a simple criterion to recognize in a general way whether such network structures are "hidden" in our data. We start with the "standard" recursive graph problem of the computation of the set of all paths in a graph. We show that the computation of the paths themselves can easily be extended with the computation of additional path properties. Such algorithms essentially operate differently from the algorithms on 3GL-level. This paradigm shift from 3GL to 4GL constitutes an important educational attention point.

It turns out that intuition regarding the correctness (and the termination) concerning these subtle "set-at-a-time" algorithms sometimes falls short. Therefore, we also pay special attention to the correctness and termination of the algorithms (using invariants). Actually, this combines some educational themes from different disciplines in computer science, namely programming (correctness, termination, invariants) and databases (4GL, stored procedures), in an elegant and useful manner.

One of the advantages of our uniform 4GL-approach is that it makes the practical development of ad hoc queries in such (recursive) application areas considerably easier, i.e., both simpler and faster, than a mixed 3GL/4GL-approach, using 3GL host languages with embedded SQL. As a consequence, it facilitates both the development and management of information systems in those application areas.

Even on 4GL-level the student can influence the efficiency of the graph algorithms. Therefore, we present some incremental improvements on our algorithms, all successively leading to better results.

All programs turn out to be rather compact; they consist of only a small number of SQL-statements. This clearly contributes to the transparency of the structure of the algorithms and the maintainability of the software, and therefore makes it also very suitable for educational purposes.

Keywords: Advanced database methods, extended relational applications, network algorithms, transitive closure, recursive queries, invariants, 3GL versus 4GL, paradigm shift, SQL

Introduction

In several database (and other) applications, structures such as trees, graphs, and networks play a prominent role; see for instance Aho, Hopcroft, and Ullman (1983), Houtsma and Apers (1992), Küng, Wagner, and Wöβ (1995), and Ullman (1989) as well as their references among (many) others. There are many well-known examples of such application areas: bills of material, genealogical trees, organization charts, holding structures, networks of railroads, networks of conduit pipes, and telecom networks, to name a few. But also many other application areas contain, often implicitly, such structures. Within computing science itself we also encounter those structures quite frequently, for example in data dictionaries, in deductive databases, in software configuration management, and in CASE-tools (e.g. ER-diagrams). Consequently, these structures occur in several courses in a CS curriculum. We will show how this can be used as an educational opportunity and combine some classical themes of programming with databases.

A tree, graph, or network can be considered as a "picture" with "nodes" and "edges", informally speaking. Usually, the nodes and the edges are "labeled" with additional data as well. For some classes of examples we indicate in Table 1 what the pictures, the nodes, and the edges represent in those cases. (The students could be invited to add some classes of examples themselves.)

Table 1: Some application areas of trees, graphs, and networks

Picture	**Node**	**Edge**
bill of material	(intermediate) product	use
organization chart	function/employee	hierarchical relationship
genealogical tree	person	parent-child relationship
decision tree	decision node	option
road map	city	road
(working) group	actor	communication channel
ER-diagram	entity	relationship

As an illustration, Figure 1 gives a concrete instance of a bill of material (or BOM) for an imaginary manufacturer of office furniture, taken from De Brock (1995). We note that a bill of material constitutes a central part of MRP-systems for instance. The nodes represent products (with product number and description), the edges indicate which products occur as a direct part in which products, and the edge labels tell us in which numbers they occur as a direct part, e.g. product 11297 (bolt + nut) occurs 24 times as a direct part in desk 87384 (and via the six drawers with number 44660 it also occurs 6 x 6 times as an indirect part in desk 87384).

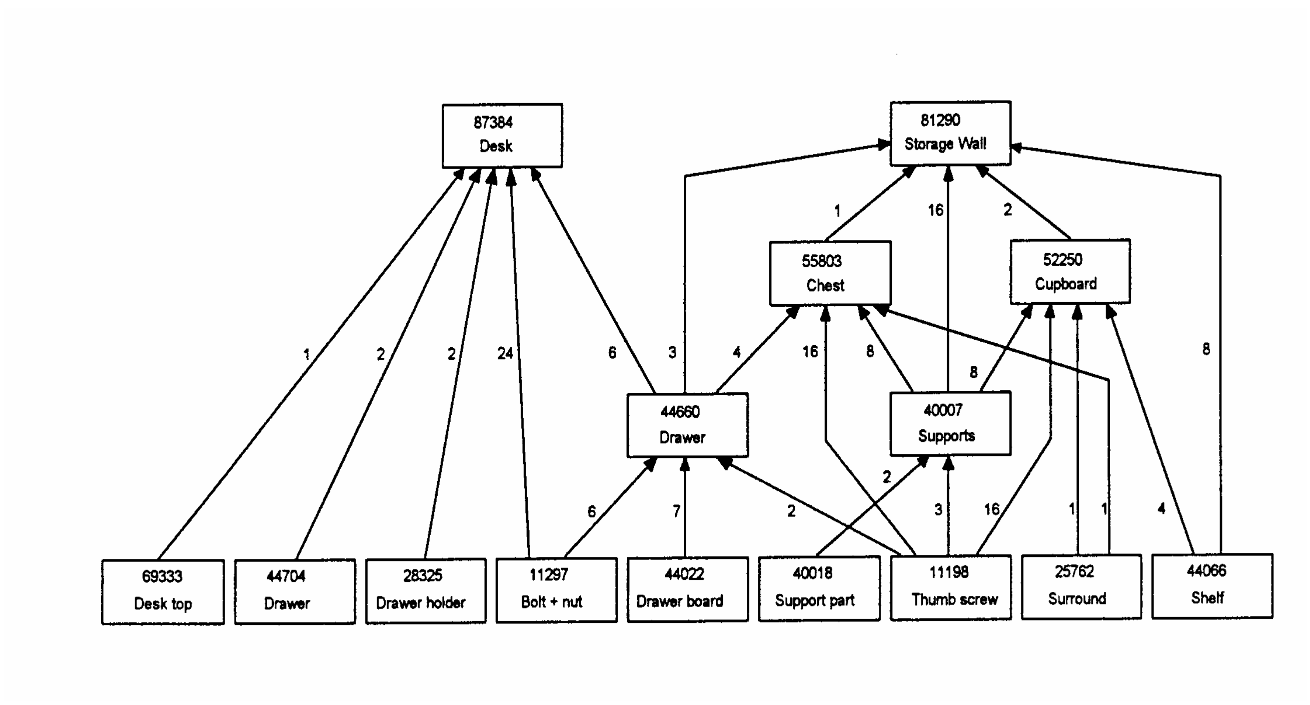

Figure 1: A bill of material

We can represent our bill of material from Figure 1 by means of two tables, the table PROD (products, or nodes) with (at least) the attributes PNR (product number) and DESCR (description), and the table EDGES with (at least) the attributes BNODE (begin node), ENODE (end node), and NUM (number of pieces). The key of the table PROD is {PNR} and the (composite) key of the table EDGES is {BNODE, ENODE}. In the table EDGES, both BNODE and ENODE are foreign keys, each referring to PNR in the table PROD. In Figure 2 we represent a part of each table.

PROD		
PNR	DESCR	...
87384	Desk	
69333	Desk top	
44704	Drawer	
28325	Drawer holder	
11297	Bolt + nut	
44660	Drawer	
.	.	
.	.	

EDGES			
BNODE	ENODE	NUM	...
69333	87384	1	
44704	87384	2	
28325	87384	2	
11297	87384	24	
11297	44660	6	
44660	87384	6	
.	.	.	
.	.	.	

Figure 2: Representation of a bill of material by means of tables

Here we already note that ad hoc querying for management applications in areas like production management, for instance, is often laborious due to the recursive character of many queries (e.g., computing pack lists, total assembly times, and the like). But before we return to this problem, we will first give an example of an even more general network. This network, represented in Figure 3, consists of 23 nodes and 37 edges.

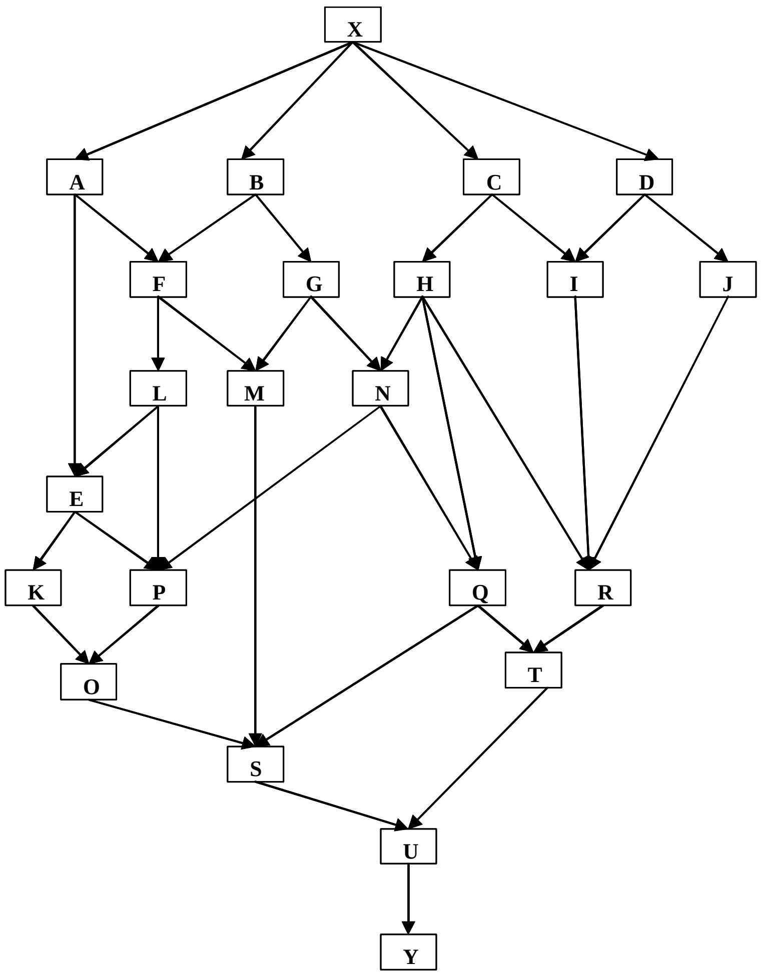

Figure 3: A network (without cycles)

Sometimes, network structures are only implicitly present in databases. This raises the question how students can recognize in a general way whether such network structures are "hidden" in their data. A simple criterion, illustrated by Figure 4, is the following: a data model with two different referential integrities from a given entity E to a given entity N can be an indication that the N-occurrences can be considered as nodes and the E-occurrences as edges between those nodes, and hence that N together with E in fact represent a network. (It would be an interesting exercise to check existing database schemes for these situations; one might get surprisingly new insights in those data structures.)

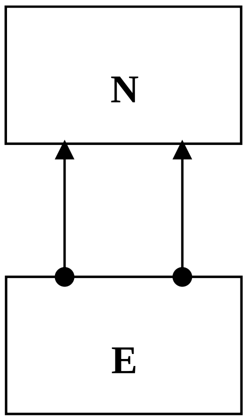

Figure 4: A simple criterion for the presence of graphs

Algorithms on such data structures often ask for recursion or iteration with an unknown number of repetitions because the depth of the tree or the "diameter" of the network can be arbitrarily large. As an illustration of the consequences of this point, note that for a given table representing a genealogical tree, we can retrieve all children or all grandchildren of a person using a fairly simple SQL-statement. However, it becomes much harder in SQL when we want to retrieve *all descendants* of a person. In general, the problem is that it is easy in SQL to retrieve all paths of a fixed length (such as length 3), but not to retrieve all paths of arbitrary length (such as, "Give all paths"). Traditionally, such algorithms will be programmed in a 3GL using recursion or iteration; see the classic Aho, Hopcroft, and Ullman (1983) for example. Those programs are then typically "record-at-a-time"; in case of iteration, one record will be treated in the (inner) **while**-loop and in case of recursion, one record will be treated in the (inner) recursive call. A classic example of the latter case is backtracking in trees.

Much theoretical work has been done on recursive queries, transitive closure algorithms and strategies as well as their implementation, e.g., in deductive database systems; see for instance Ullman (1989) and its many references. This research usually concentrates on (extensions to) Datalog and other research languages like NAIL! (Morris, Ullman, & Van Gelder, 1986) and LDL (Naqvi & Tsur, 1989), as well as their theoretical foundations, research prototypes, and their optimization. However, this deductive database research did not lead to widely used commercial database systems. Hence, this research does not directly help the database developer “in the field”, who usually only has available “ordinary” SQL with few enhancements. Therefore, it is our primary purpose to provide feasible and realistic solutions regarding such graph problems for students who might be going to work in the field with a commercially available database management system (e.g., as future database designers or application developers).

We can show that enhanced versions of SQL with assignments and “control of flow” constructions such as **while** that are commercially available make it possible to solve such graph problems completely on 4GL-level. We point out that this contributes to the transparency of the structure of the software and that this also influences the design as well as the implementation of (recursive) constraints, queries, and transactions. Officially, such language constructs were planned to occur in the SQL3-standard for the first time (see Cannan & Otten, 1992), but they have already existed in practice for some time in several commercially available RDBMS's. Well-known examples of such enhanced SQL-versions are TRANSACT-SQL of Sybase and SQL Server, INGRES/SQL of Ingres, and PL/SQL of Oracle.

We will work out the idea of graph algorithms on 4GL-level using the computation of all paths in a graph, known in the literature as the computation of the so-called *transitive closure* of the graph. This is an instance of a so-called *all-pairs paths problem*; see for example Aho, Hopcroft, and Ullman (1983). If the graph represents a genealogical tree, for instance, then this reduces to the computation of *all* descendants of *all* persons, and in case of a bill of material this reduces to the generation of *all* parts - direct as well as indirect - of *all* products. This problem models as it were many other graph algorithms. We also show that the computation of the paths themselves can be extended with the computation of additional path properties in a simple and modular way.

Our approach makes the development and implementation of ad hoc queries in such recursive application areas considerably simpler and faster in practice, which in turn facilitates the development of information systems in those application areas. (The author of this paper already wrote such graph algorithms in TRANSACT-SQL (of Sybase) at the end of the eighties, during an internal project at Philips Research. Now that more and more existing and new RDBMS's appear with such enhanced versions of SQL, the time has come to dedicate an educational paper to this topic.)

Even on 4GL-level we can influence the efficiency of our graph algorithms. As an illustration we treat three successive improvements of our algorithms, each clearly leading to successively better results, hence offering a nice example of incremental (4GL) program refinement.

It turns out that intuition on the correctness of these new “set-at-a-time” algorithms sometimes falls short. This also holds for the question whether the algorithms do terminate, due to the possible presence of cycles in the graph. For this reason, we also have to give some considerations regarding the termination and correctness of the algorithms. For graphs that might contain cycles, we will present alternative versions for each of our four algorithms. This leads to a modular variety of alternatives.

Our goals with this paper are not only to present these algorithms as such but, above all, to present a suitable and almost necessary manner to treat such complex questions in SQL: first the students ought to carry out the analysis and design at a high and compact level, namely in terms of operations on sets, before they plunge into the details of the SQL-code of the algorithms! This message adds to the educational value of this paper.

The structure of the paper is as follows. Section 1 starts with a few necessary basic notions and notations regarding paths. In Section 2 we design, analyze, and redesign the algorithms in general terms of operations on sets. Only after we have finished all analysis and redesign, we turn to the (practically applicable) implementation of all our algorithms in SQL with a few enhancements (in Section 3). The paper ends with conclusions, and an appendix offering an elaborate example.

1. Basic Notions

We can start with the introduction of only a few basic notions and notations regarding paths. A path is a sequence of nodes; sequences are written as $<x_1; x_2; \ldots; x_n>$. For example, the path p1 from the "node" 11297 via 44660 to 55803 in Figure 1 will be denoted by <11297; 44660; 55803>. The "one step path" p2 from 55803 to 81290 can therefore be denoted by <55803; 81290>. We will denote the begin node of a path p by first(p), the end node by last(p), and the path length, i.e., the number of steps (or edges) in p, by length(p). So, if $p = <x_1; x_2; \ldots; x_n>$ then $\text{first}(p) = x_1$, $\text{last}(p) = x_n$, and $\text{length}(p) = n - 1$.

As an example, for p1 = <11297; 44660; 55803> we obtain first(p1) = 11297, last(p1) = 55803, and length(p1) = 2.

Since the end node of path p1 is also the begin node of path p2, or formally, since last(p1) = first(p2), we can "glue" those two paths together, thus making a longer path. We will denote that path by p1 & p2:

p1 & p2 = <11297; 44660; 55803; 81290>.

In general, if $p = <x_1; x_2; \ldots; x_n>$ and $s = <y_1; y_2; \ldots; y_m>$, and last(p) = first(s), i.e., $x_n = y_1$, then

$p \,\&\, s = <x_1; x_2; \ldots; x_n; y_2; \ldots; y_m>$.

When we have two path *sets* R and S, then $R \oplus S$ denotes the set of all possible "glue results" of a path from R followed by a *consecutive* path from S. Formally:

$$R \oplus S = \{\, r \,\&\, s \mid r \in R \text{ and } s \in S \text{ and last}(r) = \text{first}(s) \,\}$$

With only these notations at hand the instructor is ready to treat our graph algorithms.

2. Analysis and Design of the Algorithms

Before rushing into the details of the implementation of the graph algorithms (computing the set of all paths in a graph) into SQL, the student first has to (learn to) analyze and design the algorithms at a higher and more compact level, namely in terms of operations on sets.

We presuppose the existence of a given table EDGES in which the data on the edges is already available. On behalf of our algorithms we have to introduce two auxiliary tables: a result table PATHS, which will contain all "old" paths we found *thus far*, and a table NEWP with the same structure (i.e., with the same type T), which will contain all paths we "just" found; this will be part of our invariants (see below). At this moment we do not yet need to go into the details of that structure T.

2.1 Graphs without Cycles

In this section we assume that we are sure beforehand that the graph contains no cycles. This means that our algorithms do not need to check whether they "are running around in circles". In Section 2.2 we will consider the case that the graph might contain cycles.

2.1.1 Combining paths with edges

Abstractly, one of the simplest algorithms for computing the set of all paths in a graph looks as follows (in pseudo-code):

```
1    var PATHS, NEWP: T;
2    PATHS := ∅;
3    NEWP  := EDGES;
4    while NEWP ≠ ∅
5    begin PATHS :=  PATHS ∪ NEWP;
6          NEWP  := (PATHS ⊕ EDGES) - PATHS
7    end
```

Explanation. Line 1 contains the declaration of the auxiliary tables PATHS and NEWP, each of type T.

Lines 2 and 3 initialize these variables: PATHS becomes empty and in NEWP we put all paths of length 1.

As long as the set NEWP still contains some elements (according to line 4), we add this set to PATHS (in line 5), and subsequently (in line 6) we will let NEWP consist of all paths we found thus far extended with one edge, but leaving out from this result all paths we already found thus far.

In this algorithm we have the following invariant: If the **while**-loop in this algorithm is executed, say, n times, the table PATHS will contain all paths in the graph of which the path length is at most n. Hence, the total number of times the **while**-loop will be executed is equal to the path length of the longest path in the graph. In our bill of material of Figure 1 the longest paths have path length 3. The path <11297; 44660; 55803; 81290> is an example of such a longest path. In our network of Figure 3 the longest paths have path length 9. Here, an example of such a longest path is the path <X;A;F;L;E;K;O;S;U;Y>.

At this point the students could already be asked to think about an improvement of this algorithm.

2.1.2 Combining paths with each other

We obtain a first improvement on the foregoing algorithm by combining in line 6 the set of all found paths with *itself* (and not with EDGES), and - again - leaving out from this result all paths we already found thus far. Line 6 will then be replaced by:

```
6b          NEWP := (PATHS ⊕ PATHS) - PATHS
```

Now we have the following invariant: If the **while**-loop in this algorithm is executed (n + 1) times, the table PATHS will contain all paths in the graph of which the path length is at most 2^n. The following trivial scheme indicates which maximum path lengths will already be reached in the first seven loops of the **while**-statement. (Note that in many of the application areas mentioned in Table 1, a depth of 64 is already very much in practice.)

while-loop	1	2	3	4	5	6	7
maximum path length	1	2	4	8	16	32	64

Figure 5: Maximum path lengths in each **while**-loop

In our bill of material of Figure 1, for example, the longest paths have path length 3, which means that we have obtained all paths as soon as the **while**-loop has been executed three times. In our network of Figure 3 the longest path length is 9, which means that in this case we have obtained all paths as soon as the **while**-loop has been executed five times. We note that in general the total number of times that the **while**-loop will be executed is only 1 plus the 2-log (rounded off upwards) of the length of the longest path in the graph!

Although the operation PATHS ⊕ PATHS may seem to be more complex than the operation PATHS ⊕ EDGES, the new algorithm as a whole will be an improvement.

At this point the students could again be asked whether they can come up with an improvement of the algorithm.

2.1.3 Generating only new paths

Just like the first algorithm, the second algorithm suggests that we also generate in passing the paths we found earlier (but ignore them immediately), since a literal translation of line 6b into SQL would yield something of the following general form:

```
DELETE FROM NEWP

INSERT INTO NEWP
(SELECT ...
 FROM    PATHS p1, PATHS p2
 WHERE  ... )
EXCEPT
(SELECT * FROM PATHS)
```

Fortunately we can distinguish new paths from “old” paths (i.e., paths found earlier) in a simple way, namely by means of their path length. For that matter, note that if the **while**-loop in the second algorithm is executed (n + 1) times, the table PATHS contains all paths of which the length is at most 2^n. Hence, in PATHS ⊕ PATHS only the paths longer than 2^n are new. In order to keep track of that maximum length, we introduce a variable MAXL in our next algorithm. This (third) algorithm generates only new paths:

```
1      var    PATHS, NEWP : T;
              MAXL : integer;
2      PATHS := ∅;
3      NEWP  := EDGES;
       MAXL  := 1;
4      while NEWP ≠ ∅
5      begin PATHS := PATHS ∪ NEWP;
6c           NEWP  := { p | p ∈ PATHS ⊕ PATHS and length(p) > MAXL };
             MAXL  := 2 * MAXL
7      end
```

Explanation.

The integer variable MAXL starts with the value 1 and doubles in each execution of the **while**-loop.

Immediately before the **while**-loop as well as at the beginning and the end of each execution of the **while**-loop, the following relationships hold between our variables:

(B1) PATHS contains all paths p with $\text{length}(p) \leq \frac{1}{2}\,\text{MAXL}$

(B2) NEWP contains all paths p with $\frac{1}{2}\,\text{MAXL} < \text{length}(p) \leq \text{MAXL}$

In other words, these statements are *invariant* under execution of the **while**-loop.

The **while**-loop ends as soon as NEWP = ∅.

2.1.4 Generating each new path only once

Again we can ask the students whether they see room for further improvement of this algorithm. If they don't, we can give them the following hint: Although we generate only *new* paths now, we do generate each new path *multiple times*. For instance, the path <11297; 44660; 55803; 81290> in Figure 1 is generated by the combination of the path <11297; 44660> with the path <44660; 55803; 81290> as well as by the combination of <11297; 44660; 55803> with <55803; 81290>.

In order to prevent *multiple* generation of a new path, we can compose each new path in a standard manner: the first component will be a path with the (as yet) maximum length MAXL and the second component will be a path of arbitrary length. In this manner we generate only new paths and we will generate each new path only once. Note that this is the minimal effort we have to do anyway. In order to express this idea formally, we have to write out the set PATHS ⊕ PATHS, using the definition of the "⊕"-operation introduced in Section 1. We will obtain our fourth algorithm by replacing line 6c in our former algorithm by:

```
6d      NEWP := { p1 & p2 | p1 ∈ PATHS and p2 ∈ PATHS and
                            last(p1) = first(p2) and length(p1) = MAXL }
```

2.2 Graphs That Might Contain Cycles

In Section 2.1 we assumed that we knew beforehand that the graph contained no cycles, implying that our algorithms did not need to check whether they were "running around in circles". When we drop this assumption, we have to check on cycles during execution. Therefore we have to strengthen the condition in line 4 of our four algorithms as follows:

```
4b      while NEWP ≠ ∅ and
            { p | p ∈ NEWP and first(p) = last(p) } = ∅
```

In words: we continue as long as NEWP still contains paths *and* for none of these paths the begin node is equal to the end node. Here the teacher can point out that this refinement (of line 4) is independent of the refinements made in Section 2.1 (which were all in line 6); indeed a nice illustration of the principle of "separation of concerns".

2.3 From Labeled Edges to Labeled Paths

The next (educational) step is to go from labeled edges to labeled paths. The edges in trees, graphs, and networks are often labeled with additional data. Figure 1 gives an example; there the edges are labeled with an integer (indicating the *multiplicity* of a direct part in a product, i.e., the number of pieces as a direct part in that product).

These labels usually occur as additional attributes in the "edge table", as in Figure 2.

Example. If Figure 3 represents a network of roads from north to south, we can imagine all kinds of labels (or attributes, or properties) that could be attached to the edges. For instance:

(a) the distance (in km.)
(b) the expected traveling time
(c) the maximum gradient percentage
(d) the maximum vehicle height
(e) the maximum vehicle weight
(f) the maximum vehicle width
(g) toll passage indication (Boolean)
(h) suitability indication for caravans (Boolean)
(i) the route description

At this point the instructor could ask the students for additional examples of possible attributes. (Beforehand, the instructor could leave out some of the attributes above.)

Often these edge properties can be lifted/extended/promoted to path properties. We can show that (and how) our algorithms for the computation of the paths themselves can easily be extended with the computation of such additional path properties. In order to be able to do so we have to know

(R0) which additional path properties we want to compute,

(R1) the value of each property for a one step path
(which is usually equal to the value of that property for the underlying edge), and

(R2) the value of each property for the composition p1 & p2 of two paths p1 and p2
(usually in terms of the value for p1 and the value for p2).

As an illustration of this point, the instructor can (ask to) work out part (R2) for (some of) the road properties mentioned in the example above and/or for the multiplicity of direct subparts in our BOM-example from Figure 1. In Figure 6, we denote the values of the property for p1 and for p2 by val(p1) and val(p2), respectively. Concatenation of strings will be denoted by ⎮.

	Edge property	**Value for the path p1 & p2**
(a)	distance	val(p1) **+** val(p2)
(b)	expected traveling time	val(p1) **+** val(p2)
(c)	max. gradient percentage	**max(**val(p1)**,** val(p2)**)**
(d)	max. vehicle height	**min(**val(p1)**,** val(p2)**)**
(e)	max. vehicle weight	**min(**val(p1)**,** val(p2)**)**
(f)	max. vehicle width	**min(**val(p1)**,** val(p2)**)**
(g)	toll passage indication	val(p1) **or** val(p2)
(h)	suitability for caravans	val(p1) **and** val(p2)
(i)	route description	val(p1) ⎮ **'; then '** ⎮ val(p2)
	multiplicity	val(p1) ***** val(p2)

Figure 6: Value of some properties for path compositions

So, in general the value of a property for a one step path p is usually equal to the value of that property for the underlying edge e, and the value for p1 & p2 is some function f of the values val(p1) and val(p2):

(R1)	val(p)	= val(e)
(R2)	val(p1 & p2)	= f(val(p1), val(p2))

In terms of our algorithms in Section 2.2,

- part (R0) will be used to adapt the type in line 1,
- part (R1) will be used to adapt line 3, and
- part (R2) will be used to adapt line 6 (and its variants).

3. Implementation of the Algorithms in SQL

In order to show the students the practical applicability of the approach in this paper, we will sketch how the foregoing abstract algorithms can be implemented in the commercially available enhancements of SQL we mentioned earlier.

In this section we assume some familiarity with SQL; in particular, the SELECT, INSERT, DELETE, and UPDATE statements are assumed to be familiar. We can refer to Date (1995) or De Brock (1995), for instance, for further background reading.

3.1 Implementation of our Third Algorithm

The instructor can start with the implementation of our *third* algorithm, see Section 2.1.3 (i.e., the algorithm containing line 6c); by means of our hints in Section 3.2 the student should then be able to "assemble" the code of the first two algorithms, those with line 6 and 6b respectively (in stead of line 6c).

We assume the existence of a table EDGES with the attributes BNODE (begin node) and ENODE (end -node) of type String. (When these attributes are of another type, for example Integer, then we might add a type conversion in the SELECT-part of the first INSERT-statement in the algorithm below.)

Now the time has come to go into the details of the structure of the auxiliary tables PATHS and NEWP. These tables contain four attributes (PATH, BNI, ENE, and LEN), which we will explain below.

The attribute PATH is meant to contain the path itself, represented by the successive nodes, separated by commas, but as yet *without* the end node of the path. (Otherwise, this end node would have to be deleted again when we would glue two paths together.) The UPDATE statement at the end of the program will finally add the end node to each path.

For the sake of convenience, the attribute BNI contains the begin node of the path separately (although it is in principle reconstructible from the attribute PATH).

The attribute ENE contains the end node of the path, which is, as we know, not represented in the attribute PATH.

Also for the sake of convenience, the attribute LEN contains the length of the path (although this is in principle reconstructible from the attribute PATH as well).

The CREATE TABLE statement below is, strictly speaking, not a genuine SQL statement; in practice, this statement has to be written out in two separate CREATE TABLE statements.

The statement "SELECT @MAXLEN = 1", with the integer variable @MAXLEN, is our SQL-representation of the assignment-statement "MAXL := 1" from Section 2.1.3.

In some commercially available SQL-enhancements, each of our algorithms can be "packed in" as a so-called (stored) procedure. The variable @MAXLEN can then be declared as a local variable of that procedure. We will illustrate these points in the Appendix.

The implementation of our third algorithm now looks as follows:

```
CREATE TABLE   PATHS,  NEWP

(PATH    String,             | path excluding the end node (= ENE)
 BNI     String,             | begin node of the path (inclusive)
 ENE     String,             | end node of the path (exclusive)
 LEN     Integer)            | path length

INSERT INTO NEWP
SELECT x.BNODE          as PATH,
       x.BNODE          as BNI,
       x.ENODE          as ENE,
       1                as LEN
FROM   EDGES x

SELECT @MAXLEN = 1

WHILE  EXISTS(SELECT * FROM NEWP)
BEGIN  INSERT INTO PATHS
       SELECT * FROM NEWP

       DELETE FROM NEWP

       INSERT INTO NEWP
       SELECT p1.PATH + ',' + p2.PATH  as PATH,
              p1.BNODE                 as BNI,
              p2.ENODE                 as ENE,
              p1.LEN + p2.LEN          as LEN
       FROM   PATHS p1, PATHS p2
       WHERE  p1.ENE = p2.BNI
          AND  p1.LEN + p2.LEN > @MAXLEN

       SELECT @MAXLEN = 2 * @MAXLEN
END

UPDATE PATHS p
SET PATH = p.PATH + ',' + p.ENE
```

3.2 Implementation of our Fourth Algorithm

In order to prevent multiple generation of a new path, we composed each new path from a path with the (as yet) maximum length MAXL as the first component and a path of arbitrary length as the second component; see Section 2.1.4. This led to the replacement of line 6c by line 6d; in our SQL-translation this will lead to the replacement of the line “AND p1.LEN + p2.LEN > @MAXLEN” by

```
          AND  p1.LEN = @MAXLEN
```

The student will now be able to “assemble” the code of the first two algorithms, those with line 6 and 6b respectively (in stead of line 6c); we already presented the form of the necessary SQL-statement during the discussion of line 6b in Section 2.1.3.

3.3 Performance Considerations

We recall from Section 2.1 that from the second algorithm on, the total number of times that the **while**-loop will be executed is only 1 plus the 2-log (rounded off upwards) of the length of the longest path in the graph. In each **while**-loop only one join is performed. Moreover, in the third algorithm we generate only new paths and in the fourth algorithm we also generate each new path only once (the minimal effort we have to do anyway).

The CREATE TABLE statements need not be part of the algorithm itself (as we did). For performance reasons, they can be kept outside the algorithm by introducing them once beforehand, and starting the algorithm with emptying the tables PATHS and NEWP:

```
DELETE FROM PATHS

DELETE FROM NEWP
```

Only as an illustration of the (relative) efficiency improvements that are possible, we note that in our test example from Figure 3 the execution time gained a factor 15 by the second improvement (generating only new paths). Subsequently, by the third improvement (generating each new path only once), the execution time still gained another factor 4 in this test case. A formal or experimental proof of the absolute or relative performance benefits of each of the algorithms is outside the scope of this paper, but could be added by the instructor as an interesting exercise for the students.

3.4 Implementation in Case of Possible Cycles

In order to check on cycles during execution, we have to strengthen the condition in the **while**-loop, according to the condition given in line 4b in Section 2.2. Therefore, in our earlier SQL-translations the line

"WHILE EXISTS(SELECT * FROM NEWP)"   has to be replaced by

```
"WHILE EXISTS(SELECT * FROM NEWP)
   AND NOT EXISTS(SELECT * FROM NEWP p  WHERE p.BNI = p.ENE)"
```

We note that in the foregoing text each fragment of the form "EXISTS(SELECT * FROM ...)" can be replaced by "0 < (SELECT COUNT(*) FROM ...)". We will illustrate these points in the Appendix.

3.5 Implementation of Labeled Paths

In principle, each additional path property we want to compute, will add

- an attribute Y to the type T to be chosen,
- a line of the form "x.Y as Y" to the SELECT-part of the SQL-translation of line 3, and
- a line of the form "f'(p1.Y, p2.Y) as Y" to the SELECT-part of the SQL-translation of line 6 (and its variants), where f'(p1.Y, p2.Y) is the SQL-translation of f(val(p1), val(p2)) mentioned in Section 2.3.

We note that standard SQL does not contain the type Boolean directly. Therefore, we simulate it by a user-defined type (or "domain") called Bool, consisting of the integers 0 (for "false") and 1 (for "true"). In SQL2 (see Cannan & Otten, 1992) this can be done as follows:

```
CREATE DOMAIN Bool AS INTEGER

CHECK(VALUE = 0 OR VALUE = 1)
```

In this case, the Boolean expression "A **and** B" translates to min(A,B) and "A **or** B" translates to max(A,B). We will illustrate all these points in the Appendix.

In your favorite RDBMS, the syntax for domains might be slightly different; if your RDBMS does not support user-defined types yet, you might be able to add a CHECK-clause to the type declaration of the intended Boolean attribute.

We also note that standard SQL does not contain the operations min(a,b) and max(a,b), for the minimum and maximum of two given numbers a and b, which we used in our examples in Section 2.2. Note, however, that

$$\min(a,b) = (a + b - \text{abs}(a,b))/2$$

$$\max(a,b) = (a + b + \text{abs}(a,b))/2$$

where abs(a,b) denotes the absolute value of a – b. The right hand sides of these expressions do have a counterpart in many SQL-implementations.

Conclusions

Commercially available enhancements of SQL with assignments and "control of flow" constructions such as **while** constitute the educational challenge to teach these topics in a clean and clear manner, retaining the lessons learned in earlier programming courses (top down design, modular design, correctness and termination considerations, invariants) as well as in the earlier parts of the database course. Enhanced versions of SQL with assignments and "control of flow" constructions make it possible to express (seemingly recursive) algorithms on trees, graphs, and networks completely on 4GL-level. This clearly contributes to the transparency of the structure and the maintainability of the software. As a consequence, our approach makes the development and implementation in commercially available database management systems of ad hoc queries in such (recursive) application areas considerably simpler and faster for application developers, which in turn facilitates the development and management of information systems in those application areas. As such, it may constitute a useful theme in a CS course on software development in a database environment.

A simple criterion for students to recognize whether network structures are "hidden" in their data is that a data model with two different referential integrities from an entity E to an entity N can be an indication that the N-occurrences can be considered as nodes and the E-occurrences as edges between those nodes, and hence that N together with E in fact represent a network.

Even on 4GL-level the students can influence the efficiency of their graph algorithms in a substantial way.

We also paid special attention to the correctness and termination of the algorithms, by using invariants.

All programs turn out to be rather compact: they consist of only a few SQL-statements. This makes our programs surveyable, easily adaptable, and very suitable for educational as well as practical purposes.

Our goal with this paper was not only to present these algorithms as such but (above all) to offer a suitable (and almost necessary) manner to treat complex queries in SQL: first the students ought to carry out the analysis and design at a high and compact level, namely in terms of (operations on) sets, before they plunge into the details of the SQL-code! This is our general experience in teaching the development of complex queries for more than 20 years, and this message will hold even stronger in the context of (even more complex) iterative queries. The method has not yet been formally evaluated from a student perspective in this new context of complex iterative queries, so it would be interesting future work to carry out such an evaluation. The reader is invited to apply the method and to evaluate it in his or her own classroom.

Acknowledgements

The author is grateful to Herman Balsters and the reviewers for their comments on earlier versions of this paper.

References

Aho, A.V., Hopcroft, J.E. & Ullman, J.D. (1983). *Data structures and algorithms.* Reading, MA: Addison-Wesley.

Cannan, S.J., & Otten, G.A.M. (1992). *SQL, the standard handbook.* London: McGraw-Hill.

Date, C.J. (1995). *An introduction to database systems.* Reading, MA: Addison-Wesley.

de Brock, E.O. (1995). *Foundations of semantic databases.* London: Prentice Hall International Series in Computer Science.

Houtsma, M.A.W., & Apers, P.M.G. (1992). Algebraic optimization of recursive queries. *Data & Knowledge Engineering, 7*, pp. 299-325.

Küng, J., Wagner, R., & Wößβ, W. (1995). A rule driven transformation processor for bill of material data. *DEXA'95*, *Lecture Notes in Computer Science 978*, pp. 545-553.

Morris, K., Ullman, J. & van Gelder, A. (1986). Design overview of the NAIL! System. Proceedings of the 3rd International Conference on Logic Programming, *Lecture Notes in Computer Science 225*, pp. 554-568.

Naqvi, S., & Tsur, S. (1989). *A logical language for data and knowledge bases.* New York: Computer Science Press.

Ullman, J.D. (1989). *Principles of database and knowledge-base systems. Vol. II.* Rockville, MD: Computer Science Press.

Appendix: An Elaborated Example

This appendix contains an example to illustrate several points we made earlier in this paper. In particular, we will work out the SQL implementation

– of the road network example mentioned in Section 2.3, with six representative additional edge labels:

distance, maximum gradient percentage, maximum vehicle height, toll passage indication,

suitability indication for caravans, and the route description,

– for the case that the network might contain cycles (see Section 2.2, line 4b, and Section 3.4),

– in the form of a (stored) procedure (see Section 3.1), for which we chose the name *Pathfinder*,

– for the case that we generate each new path only once (see Section 2.1.4, line 6d, and Section 3.2),

– with CREATE TABLE statements for PATH and NEWP outside the procedure itself (Section 3.3), and

– with the EXISTS-version replaced by the COUNT(*)-version (see Section 3.4).

We start with the CREATE TABLE statements for PATH and NEWP before we present the procedure itself:

```
CREATE TABLE  PATHS, NEWP

(PATH    String,            | path excluding the end node (= ENE)
 BNI     String,            | begin node of the path (inclusive)
 ENE     String,            | end node of the path (exclusive)
 LEN     Integer,           | path length

 DIST    Integer,           | distance (in km.)
 GRAD    Integer,           | maximum gradient percentage
 MHEI    Integer,           | maximum vehicle height
 TOLL    Bool,              | toll passage indication
 CVAN    Bool,              | suitability indication for caravans
 DESC    String)            | route description
```

In Section 3.5 we noted that we could use the equalities

$$\min(a,b) = (a + b - \text{abs}(a,b))/2$$

$$\max(a,b) = (a + b + \text{abs}(a,b))/2$$

in order to express the minimum and maximum of two given numbers directly in SQL. For convenience sake, we will not write this out in the code below.

We continue with the stored procedure itself:

```
CREATE PROCEDURE Pathfinder AS
DECLARE  @MAXLEN  Integer
BEGIN
  DELETE FROM PATHS
  DELETE FROM NEWP

  INSERT INTO NEWP
  SELECT x.BNODE      as PATH,
         x.BNODE      as BNI,
         x.ENODE      as ENE,
         1            as LEN,
         x.DIST       as DIST,
         x.GRAD       as GRAD,
         x.MHEI       as MHEI,
         x.TOLL       as TOLL,
         x.CVAN       as CVAN,
         x.DESC       as DESC
  FROM   EDGES x

  SELECT @MAXLEN = 1

  WHILE  0 < (SELECT COUNT(*) FROM NEWP)
     AND  0 = (SELECT COUNT(*) FROM NEWP p WHERE p.BNI = p.ENE)
  BEGIN  INSERT INTO PATHS
         SELECT * FROM NEWP

         DELETE FROM NEWP
```

```
            INSERT INTO NEWP
            SELECT p1.PATH + ',' + p2.PATH          as PATH,
                   p1.BNODE                         as BNI,
                   p2.ENODE                         as ENE,
                   p1.LEN  + p2.LEN                 as LEN,
                   p1.DIST + p2.DIST                as DIST,
                   max(p1.GRAD, p2.GRAD)            as GRAD,
                   min(p1.MHEI, p2.MHEI)            as MHEI,
                   max(p1.TOLL, p2.TOLL)            as TOLL,
                   min(p1.CVAN, p2.CVAN)            as CVAN,
                   p1.DESC + '; then ' + p2.DESC    as DESC
            FROM    PATHS p1, PATHS p2
            WHERE  p1.ENE = p2.BNI
                AND  p1.LEN = @MAXLEN

            SELECT @MAXLEN = 2 * @MAXLEN
  END

  UPDATE PATHS p
  SET PATH = p.PATH + ',' + p.ENE
END
```

From this point on it is relatively easy for students to exercise all kinds of “path related” queries; e.g., the query for the distance table for all toll free north-south routes that are suitable for caravans can now be expressed as follows:

```
SELECT  p.BNI,  p.ENE,  min(p.DIST)
FROM     PATHS p
WHERE  p.TOLL = 0  AND  p.CVAN = 1
GROUP   BY BNI, ENE
```

Biography

Bert de Brock is an associate professor of Information Technology at the University of Groningen since 1993 and a professor of Business Intelligence at the Hanze University Groningen since 2003. He received a M.Sc. in Mathematics at the University of Groningen in 1979 and a Ph.D. in Computing Science at the University of Technology in Eindhoven in 1984. From 1985 to 1990 he worked at Philips Research on the PRISMA-project (Parallel Inference and Storage Machine) and the ECHO-project (Electronic Case Handling in Offices). In 1990 he and one of his former colleagues started a company in the areas of IT-consultancy, post-academic education, and analysis, design, and construction of (tailor-made) information systems for customers. His research interests include databases, information systems, business intelligence, and bioinformatics. He is the author of the book Foundations of Semantic Databases (Prentice Hall International Series in Computer Science, 1995).