МАТЕМАТИЧКИ ВЕСНИК
46 (1994), 105–112

UDC 519.682
оригинални научни рад
research paper

A SOLUTION TO THE PARTS EXPLOSION PROBLEM IN (LISPKIT) LISP/SQL LANGUAGE

Nenad Mitić

Abstract. In this paper a new approach to resolving the parts explosion problem [1] will be presented. The parts explosion problem is well known as a problem that is beyond the capabilities of classical relational algebra. This problem can not be solved in a natural way in an SQL system by means of recursive calls, because SQL does not support recursion in a classical sense. Suggested solution is recursive and it is based on a different concept of a result of an SQL query embedded in a programming language.

1. Introduction

The parts explosion problem is defined in a following way [1]: Let a database contain a relation PARTS whose contents are a list of parts, defined as

```
CREATE TABLE PARTS
    (P# ...... NOT NULL,
    ...................., — other related data
    PRIMARY KEY ( P# ) );
```

and an other relation PP, in which component relationships between parts from the relation PARTS are stored. A pair of parts (P_x, P_y) appearing in a row of the table PP denotes that the part P_y is an immediate component of the part P_x. Such a relation would be defined as

```
CREATE TABLE PP
    (MAJORP# ...  NOT NULL,
    (MINORP# ...  NOT NULL,
    .........               , — other related data
    PRIMARY KEY ( MAJORP#, MINORP# ),
    FOREIGN KEY ( MAJORP# ) REFERENCES PARTS ...  ,
    FOREIGN KEY ( MINORP# ) REFERENCES PARTS ...  );
```

Parts from the PARTS table may have any number of immediate components. All of them are represented by the relation PP. Further on, due to simplicity, other attributes of the table PP, marked as "other related data", will not be considered (e.g. QUANTITY attribute, indicating how many instances of a given MINORP# are required as immediate components of a single instance of a given MAJORP#). For instance, let the table PP have the following contents:

MAJORP#	MINORP#
P1	P2
P1	P3
P1	P4
P2	P3
P2	P5
P3	P5
P3	P6
P4	P3
P4	P6
P7	P8
P7	P9

The parts explosion problem can be stated now as follows:

Find all the components of a given part (say part P1) to all levels.

Although the problem is stated in terms of parts and their components, it is of much more general applicability. The parts-to-parts structure PP can be regarded as a prototype for a wide class of such structures-family trees, organization charts, entity type hierarchies, and so on.

2. Tree-structured relations

A relation such as the parts-to-parts relation PP can be regarded as a collection of trees, in the following sense:

- Every part which is not an immediate component of another part is the root of some tree;
- All the immediate component P_y of a part P_x are direct descendants of the node P_x.
- Only parts of the relation PP can be nodes and leaves in the tree;

For example, according to the sample contents, the relation PP can be regarded as if it consisted of two such trees, one for part P1 and the other for part P7. Figure 1 shows the structure of the tree for the part P1.

Note that the tree on the Figure 1 is not necessarily a binary tree, and that a tree for any given part could contain duplicate nodes, in general (like the node P5).

It is well known that it is impossible to formulate a part explosion as a single expression in the relational algebra or relational calculus. In many languages [2], including in particular applicative SQL, it is possible but complicated to write a program which does the same thing. This program traverses each tree in a preorder (top to bottom, left to right) way, and, for the tree on the Figure 1, displays the following result:

P1, P2, P3, P5, P6, P5, P3, P5, P6, P4, P3, P5, P6, P6

The result obtained is much more readable if shown in a tabular form:

MAJORP#	MINORP#
P1	P2
P1	P3
P1	P5
P1	P6
P1	P5
P1	P3
P1	P5
P1	P6
P1	P4
P1	P3
P1	P5
P1	P6
P1	P6

Such a table cannot be used efficiently due to at least two reasons:

- it does not present a relation as it contains duplicate rows, and
- although it shows all the components of the part P1, it does not show its structure (e.g. level on which components P2 and P3 take part is not known)

It follows that a program which solves the parts explosion problem successfully should produce the result in a form of a table (that can be used in a relational database) representing the complete structure of a given part.

3. Resolving parts explosion problem using SQL

embedded in imperative programming languages

Resolving the parts explosion problem in imperative programming languages is reduced to resolving the tree traversal problem. As the algorithm of the tree traversal is recursive by nature, the given code will be (implicitly or explicitly) recursive too.

A natural way for resolving such a problem is designing a procedure whose parameters are:

- the part whose components are to be found, and
- the level on which the part is located.

In this procedure the cursor should be defined for fetching the parts that are immediate components of the parameter part. The same procedure is called, with increasing level, for every immediate component fetched. The procedure ends when the end of the list of immediate components of the part is reached. Parameters for initial procedure call are the part whose structure is to be defined, and the number 0, as the starting level.

This procedure is only theoretically correct because the SQL does not support the recursive cursor (e.g., it is not allowed to open more than one instance of the same cursor at the same time). In the SQL language, by definition, every attempt to open the cursor that is already opened fails with the SQLCODE that denotes an error. For that reason Date [1] elaborates two ideas as a possible manner to resolve such a problem:

- extending the cursor mechanism of embedded SQL (introducing "reopenable cursor"), and
- extending the function of SQL SELECT operation (introducing "tuple values" and treating the SQL SELECT as a tuple value)

Both ideas are only theoretical ones because of the way of using the SQL language in imperative programming languages.

Instead of it, it is possible to solve the problem by using the cursor which is closed every time an immediate component of some part is fetched. The cursor definition includes ordering by immediate components. It enables the cursor after reopening to skip all immediate components previously processed. For immediate component obtained the same procedure is called recursively with increasing level. The procedure that contains such a cursor finishes when the end of the list of immediate components of the part is reached. The argument for initial procedure call is the part whose structure is to be defined, and the number 0, as the starting level. The procedure is very inefficient: during the execution of this procedure, for every node X that has n descendants (there are n branches from the node X), such a cursor is opened and closed $n+1$ times.

4. Resolving parts explosion problem using Lisp/SQL

By using the embedded SQL in a functional programming language Lispkit Lisp, it will be possible to solve the parts explosion problem in simple and efficient way. The pure functional programming language Lispkit Lisp/SQL [3] represents the extension of the Lispkit Lisp language by the interface to DB2 Relational DataBase Management System. The interface constructed enables the usage of SQL relational query language. It has been implemented in Lispkit Lisp and PL/I languages, and presents a connection to DB2. It runs under MVS/ESA operating system on IBM/3090 machine.

As a basis for the interface constructed the mechanism for dynamic execution of SQL queries [4] is used. The usage of the dynamic instead of the static SQL, enables easier implementation of a number of SQL statements in the interface. Moreover, all the SQL statements that are supported by the dynamic SQL are implemented, except parameter markers. No differences exist in syntax of the implemented and SQL/DB2 versions. The host variables can be used in the same manner as in all the other host DB2 languages. Every host variable must have ':' as a prefix with no exception. The only condition that must be fulfilled is that a host variable name is equal to a name of some argument of Lispkit Lisp function in which that SQL query appears.

SQL query is present in a Lispkit Lisp program as an argument of the function EXECSQL. The value of this function is evaluated as follows:

- In case that given query does contain select statement, all rows from the result table will be placed (with necessary conversions) into the list that becomes the value of the function EXECSQL. Such a list consists of sublists which are the elements of the table that is generated by the execution of the query
- In case that given query does not contain select statement or the resulting table is empty, the value of the function EXECSQL is an empty list (NIL)
- In case that execution of the given query produces an error, a message will be generated and the result of the EXECSQL function will be set to an empty list (NIL).

As we deal with a pure functional programming language, host variables can not be used as locations in which the values obtained by a given query will be placed. Because SQL query is defined this way, and holding results is managed appropriately, there is no need for introducing the concept of cursor in Lispkit Lisp/SQL. Practically, designed SQL interface represents a mixture between an interactive SQL and SQL embedded in imperative programming languages. As in interactive SQL, the result of the SELECT operation could be the whole table. On the other side, data from such a table can be manipulated with in a program in the same manner as it is done in imperative programming languages. This new quality and the fact that Lispkit Lisp (as any other functional language) supports recursion, enables relatively simple coding of the program that solves parts explosion problem in a natural way.

The following Lispkit Lisp/SQL program, for a given part name as an argument, produces the table which shows its structure in detail:

```
(LETREC PPO
```

```
(PP0 LAMBDA(PART)
  (LET (IF (EQ NIL QUERY)
      (PRINTNL (QUOTE (PART has no components ))
      (TABLE_PRINT (PP1 PART QUERY 1))
    )
    (QUERY EXECSQL SELECT MINORP#
        FROM PP
        WHERE MAJORP# =:PART) ))

(PP1 LAMBDA(PART LIST LEVEL)
  (IF (EQ LIST NIL)
    NIL
    (APPEND (PP2 PART (CAR (CAR LIST)) LEVEL)
        (PP1 PART (CDR LIST) LEVEL) )))

(PP2 LAMBDA(PART SUBPART LEVEL)
  (LET (IF (EQ NIL QUERY)
    (CONS (CONS (KEY) (CONS PART (CONS SUBPART (CONS LEVEL NIL))))
          NIL)
    (CONS (CONS (KEY) (CONS PART (CONS SUBPART (CONS LEVEL NIL))))
          (PP1 PART QUERY (ADD LEVEL 1)) ))
    (QUERY EXECSQL SELECT MINORP#
      FROM PP
      WHERE MAJORP# =:SUBPART) )))
```

The function PRINTNL prints the argument in a new line, while TABLE_PRINT prints (especially structured) list in a form of the table. The value of the KEY function is an integer equal to the number of callings (executions) of the KEY function in the program.

By execution of the SQL query in the function PP0, the list of all immediate components of the part that is an initial argument of the function is produced. The initial part, the list obtained, and a number 1 (denoting the first level) represent input arguments of the function PP1. The function PP1 calls the function PP2 for each element of the list (second input argument). The function PP2 evaluates immediate components of the element. In case an argument of the function PP2 has its own immediate components, the function PP1 is called recursively. Otherwise, the value evaluated becomes the value of the function and execution stops.

The result of the execution of the previous program with P1 as an input argument is:

Numb	Majorp#	Minorp#	Level
13	P1	P4	1
12	P1	P6	2
11	P1	P3	2
10	P1	P6	3
9	P1	P5	3
8	P1	P3	1
7	P1	P6	2
6	P1	P5	2
5	P1	P2	1
4	P1	P5	2
3	P1	P3	2
2	P1	P6	3
1	P1	P5	3

5. Comments on the functional solution

The solution obtained by the previous Lisp/SQL program satisfies the listed conditions entirely:

- it shows the complete structure of the part desired on all levels. The meaning of column Level is the level on which the part Minorp# is the component of the part Majorp#
- it represents a relation that can be used further on in a relational database. Adding the column Numb enables defining a primary key of the relation and keeping obtained data in the database.

Moreover, by analyzing the result obtained (the contents of the table), structure of a tree defined by the original relation PP can be reconstructed entirely. The tree is formed using the following algorithm:

- Define the depth of the tree as the value of the attribute Level in the row with the smallest value of Numb (in the previous example the depth is 3)
- Process rows in an increasing order of values of the attribute Numb
- Associate data (Minorp#) contained in the row with the smallest value of the attribute Numb, with the "low-left" leaf in the tree
- Processing row by row, place other nodes at the appropriate level in the tree (determined by value of the Level attribute in that row). Associate such a node with value of Minorp# attribute in the processed row. If the value of the Level attribute decreased (with respect to the value of the Level attribute in the previously processed row), connect that node with all unconnected nodes placed one level deeper (in the previous example such a row is "3 P1 P3 2", and previously processed rows are "1 P1 P5 3" and "2 P1 P6 3"). Then, if necessary repeat this step in the algorithm.

- When the value 1 for the Level attribute in a row is reached, it denotes that construction of a subtree is completed. The Minorp# value in the row is an immediate component of the given part. The remaining data (if exist) form other subtrees.
- Associate a root of the overall tree with value of the Majorp# and connect it with all the subtrees previously constructed.

In the Lisp/SQL program, the SQL query is executed exactly once for each node in the tree. It represents a great decrease of demand for resources with respect to the method described in [2]. The program is simple to use by the end user. It runs both in interactive and batch mode. Although it is based on dynamic SQL, no problem with performance has been noticed while the program has been tested.

6. Summary

This paper proposes a method for solving parts explosion problem. An algorithm is designed and implemented in SQL integrated in a pure functional programming language Lispkit Lisp. It solves the given problem efficiently and in a natural way. It is simple and easy to use by end user. The efficiency and simplicity of the solution are reached due to the way of integrating SQL into Lispkit Lisp language. The integration is done by using the dynamic SQL, and the language obtained possesses attractive features of both interactive SQL and SQL embedded in imperative programming languages. The fact that integrated system does not need the concept of a cursor for processing tables, enables the recursion to be used as a natural way for solving the problem.

7. Acknowledgments

I would like to thank Gordana Pavlović-Lažetić who have read and commented on previous draft of this paper.

REFERENCES

[**1**] C. J. Date, *A Note on the Parts Explosion Problem*, in *Relational Database: Selected Writings*, Addison-Wesley 1986

[**2**] C. J. Date with Colin J. White, *A Guide to DB2* (3 ed.), Addison-Wesley 1989

[**3**] Nenad Mitić, *Integracija SQL/DB2 u Lispkit Lisp jeziku*, XXXVII konferencija ETAN-a, Beograd 1993, sveska 8, pp. 103-108

[**4**] IBM, DB2 V2 Release 2: *Application Programming and SQL Guide*, SC26–4377–1

(received 18.10.1994.)

Faculty of Mathematics, University of Belgrade, Studentski trg 16, 11000 Beograd, Yugoslavia
xpmfm23@yubgss21.bg.ac.yu