DOI: 10.17951/ai.2016.16.1.12

ANNALES
UNIVERSITATIS MARIAE CURIE-SKLODOWSKA
LUBLIN – POLONIA

VOL. XVI, 1 SECTIO AI 2016

EXTENDING DQL WITH RECURSIVE FACILITIES

Marta Burzańska [1], Przemysław Krukowski[1], Piotr Wiśniewski [1]

[1] Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University in Torun

KEY WORDS: ORM, PHP, Doctrine, recursive queries, CTE.

ABSTRACT: Object Relational Mappings reduce a gap between Relational Databases and programming languages. However, only the simplest operations are covered by the ORM frameworks. Most facilities provided by DBMSs are not usable via ORM. Among such features are recursive queries, introduced in SQL:99 standard. This paper presents integration of Recursive Common Table Expressions with Doctrine Query Language - a part of Doctrine ORM framework for PHP.

1. INTRODUCTION

Hierarchical and graph structures can be found everywhere. Real-world examples are: bill-of-material, employee hierarchy, network of roads between cities. Commonly used version control systems represent direct acyclic graphs in which a single "checkin" can have from zero (the initial "checkin") to multiple parents (predecessors). To search through such structures, when the length of the search path is unknown - we need recursion. When building database application we may either use user defined stored server-side recursive functions, emulate recursion by sending database requests in each step of recursion, or utilize built-in recursive queries. Our intuition tells us that usually it would be best to use recursive queries, mainly due to DBMS's built-in optimization techniques. However, not all DBMSs implement recursive queries, despite their introduction in SQL:99 standard. What is more, each DBMS that supports recursive queries differs in their implementation from the standard schema [1].

Let us return to the problems of building database application. Nowadays, most applications are developed in an object-oriented language, but their backend is a relational database [1]. To improve the process of an application development, a common practice is to use Object-Relational Mapping (ORM). Programmers, that decide to use a popular external library gain time needed to write their own non-standard methods of data handling. They also get a tool that has been through-out tested by a number of users, which significantly increases safety and often results in improvements to memory management and query processing speed. However, the main benefit of the usage of ORM libraries is the speed of code development and code portability between different DBMSs. Examples of the most popular ORM solutions are: Hibernate (Java) [2], Django Models (Python)

[3], Entity Framework (.NET platform) [4], ActiveRecord (Ruby) [5] and Doctrine ORM (PHP) [6]. Unfortunately, they do not offer full capabilities of a professional DBMS. Most ORM frameworks provide additional data management capabilities through an alternative object query language. Usually it is similar to SQL, however it references objects and not records nor cells, and its data types are derived from the host language and not from a database. Examples of such a language are HQL (Hibernate Query Language) in Hibernate library and DQL (Doctrine Query Language) in Doctrine ORM. While those languages extend the base functionality of an ORM, they still lack the support for more advanced SQL language functionalities, including recursive queries.

In this paper we discuss an extension of the Doctrine Query Language with a new functionality - hierarchical data handling and processing of recursive queries. This is not a trivial issue. The solution must reflect the spirit of ORM - one notation unchanged regardless of the choice of the underlying DBMS. However, although the standard of recursive CTEs has been formulated over a dozen years ago, not all DBMSs implement them. Moreover, as we have already mentioned, various implementations may differ significantly from each other. The initial prototype introducing the concept of server-side recursion into a HQL language was presented in [7]. It added support for recursive Connect By queries available in the Oracle database. This paper extends previous studies with the support for recursive common table expressions (RCTE) and recursive queries emulator, based on the unrolling of queries algorithm, designed for DBMSs not supporting recursive CTE.

We have chosen Doctrine Project for our experiments based, among many other aspects, on PHP's popularity and Doctrine Project's specific design . It is a set of libraries expanding the functionality of PHP that allows for integration with relational and non-relational DBMSs. It splits its functionality between different layers of abstraction, which makes it easy to create a unified solution regardless of the differences between various DBMSs. One of its layers - the Database Abstraction Layer (DBAL) is specifically designed for quick implementation of database communication mechanisms for database. A programmer may utilize to connect and work with a database which does not have an appropriate PDO library, while maintaining uniform approach to data handling. An important element of the Doctrine ORM framework is its object-oriented query language DQL, built in resemblance to SQL. It does not refer to tables or records, but to the objects and relationships that link them. Doctrine is able to convert DQL query to a query in the SQL dialect of the currently connected DBMS.

This paper makes the following contributions:

- we propose an extension to DQL with recursive common table expression functionality,
- recursive queries computation method that allows applications to run such queries even if underlying DBMS does not support them,
- proof-of concept implementation of this extension with experimental results that prove the robustness of our idea.

This paper is organized as follows. In Section 2 we discuss querying recursive data structures using RCTE expressions. Section 3 describes the design of the proposed extension to DQL. In Section 4 we discuss recursive query emulation based on query

unrolling. Section 5 reports the details of the performance evaluation of our prototype implementation. Section 6 concludes.

2. RECURSIVE QUERIES

The research presented in this papers focuses on the problem of processing graph and hierarchical data structures. There is an abundance of real-life problems associated with such structures among which are: finding the communication links between two cities or finding routes based on information provided by the GPS systems, processing championships' scoreboards, corporate hierarchy or bill-of-material.

The first DBMS implementing recursive queries was Oracle (version 2). They were based on the Connect by statement and were used to recursively filter out rows in a selected (only one) table. Alternative version of a recursive query, based on common table expressions, has been introduced in 1997 in IBM's DB2 database. Their version has been added to the SQL standard ANSI SQL:99. This version has the following syntax:

```
WITH [RECURSIVE] cte_name [(column list)] AS
    ( seed_query
    UNION ALL
        recursion_query)
outer_query
```

For example, to traverse a bill-of-material structure we may utilize the following query:

```
WITH RECURSIVE included_parts (sub_part, part, quantity) AS
( SELECT sub_part, part, quantity
    FROM parts WHERE part = 'our_product'
UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part )
SELECT sub_part, SUM(quantity) as total_quantity
    FROM included_parts
    GROUP BY sub_part
```

Using this method one may traverse any hierarchical data without the a priori knowledge about the depth of the tree. This task is impossible to accomplish with the use of multiple subqueries or joins. And even with the knowledge about this tree's depth such queries quickly become gigantic and hard to maintain. The use of the recursive queries aids in gathering various data (like the tree's depth) in an organized fashion.

Nowadays most relational DBMSs implement RCTEs (despite the implementational differences). Even Oracle, which for many years provided recursion through the Connect By statement, implements RCTE construct. However, there is still a number of popular systems, like MySQL or MariaDB, which do not support such queries.

3. EXTENDING DQL WITH RECURSIVE CTE

As mentioned earlier, the DQL language does not support recursion. In order to query recursive data we have to write code fragments in DQL and process them using PHP's

recursive functions or loops. This paper adds support for recursive queries by making modifications to the Doctrine ORM library. Those modifications however, maintain backward compatibility with the basic version of the system - they do not affect the behaviour of queries other than RCTE. We have added the following rules to the DQL grammar:

```
RecursiveStatement ::= RecursiveClause RecursiveBody
SelectStatement RecursiveClause ::= "WITH RECURSIVE " \
    RecursiveFunctionName"(" \
        {RecursiveFunctionArguments} ") AS "
RecursiveBody ::= ( NonRecursiveTerm "UNION" | \
    "UNION ALL" RecursiveTerm )
NonRecursiveTerm ::= SelectStatement
RecursiveTerm ::= SelectStatement
RecursiveFunctionPathExpression ::= \
    RecursiveFunctionName "." \
    IdentificationVariable [ "." StateField ]
```

In the basic DQL grammar the initial non-terminal symbol is "QueryLanguage", which may be substituted with one of three non-terminal symbols depending on the first encountered keyword in a query:

- SelectStatement for SELECT keyword
- UpdateStatement for UPDATE keyword
- DeleteStatement for DELETE keyword

Here we have added a new non-terminal symbol "RecursiveStatement" replacing QueryLanguage symbol when lexer encounters the "WITH" keyword. The above-mentioned grammar rules regarding RecursiveStatement result in the following recursive query usage:

```
WITH RECURSIVE functionName(argument1, argument2...)
AS ( /*seed query*/
    SELECT valueList
        FROM aliasDefinitions
        [WHERE...] [GROUP BY ...] [HAVING ...] [ORDER BY...]
UNION [ALL]
/*recursive query*/
    SELECT valueList
        FROM aliasDefinitions
        [WHERE...] [GROUP BY ...] [HAVING ...] [ORDER BY...]
    )
/*outer query*/
SELECT valueList
    FROM aliasDefinitions
        [WHERE...] [GROUP BY ...] [HAVING ...] [ORDER BY...]
```

We have decided upon a form similar to its counterpart in SQL for several reasons. First of all, this form is clear - every step of recursive query processing is defined separately. In addition, one of the main features of the DQL is its similarity to SQL. In order to be consistent with the DQL ideology, we have to keep our construct compatible with the underlying SQL query. At the same time the authors were unanimous about the fact that

the preservation of the RCTE structure will make it easier for developers to work with those queries as they will be able to express directly their knowledge of SQL and the intended purpose of the query. We have decided to limit the structure to one base clause and one recursive clause, although some databases allow certain deviations from this rules.

Changes to the DBAL (Database Abstraction Layer) library responsible for communication with DBMSs are fairly small. They take into account the syntactic differences between database dialects and that for the MySQL the emulator of recursive queries (discussed in the next section) should be launched. For this purpose we have prepared the interface Doctrine\DBAL\Platforms\RecursivePlatform:

```
namespace Doctrine\DBAL\Platforms;
interface RecursivePlatform {
    public function withClause();
}
```

This interface is implemented by the database-specific classes (eg. Doctrine\DBAL\Platforms\OraclePlatform). Another problem was DQL's path expression restrictions. Such a path expression can contain only a single nesting (one dot). So it is not allowed to reference fields of a subobject: “c.parent.id”. For the purposes of handling recursive queries, this functionality has been changed. However, for paths that start with the alias pointing to the objects mapped in the Doctrine ORM, the double nesting is still not allowed. The modified path expression may reference the recursive function and its arguments, which may be both scalar values and mapped objects. Argument types of a recursive function are determined by the types of variables declared in the FROM clauses in seed and recursive subqueries. In the following exemplary query, both seed and recursive parts of the WITH query determine the type of the "d" variable introduced in the query's heading:

```
WITH RECURSIVE cat(d) AS
( SELECT c FROM Entity\Category c
    WHERE c.id = 1
    UNION ALL
    SELECT cr FROM Entity\Category cr, cat
    WHERE cr.parent = cat.d.id
)
SELECT cat.d FROM cat
```

4. RECURSIVE QUERIES EMULATOR

For the purpose of DBMSs that do not support recursive queries, we have created an emulator unrolling a recursive query to the linear form of multiple queries sent to the database server in a loop managed from within PHP.

The emulator class (Doctrine\ORM\Query\RecursiveEmulator) performs the task of arecursive DQL query execution in three steps:

1) evaluation of non-recursive subquery
2) evaluation of the recursive subquery
3) evaluation of the main query.

Before launching the emulator's "execute()" function the following steps have already been conducted:

1) DQL query lexical analysis (Doctrine\ORM\Query\Lexer)
2) DQL query syntax and semantic analysis (Doctrine\ORM\Query\Parser).

As a result of these steps, we obtain an AST (Abstract Syntax Tree) processed using three instances of the parser and three instances of the Doctrine\ORM\Query\SqlWalker containing the respective arrays of data created as a result of running the query. The most important of these arrays is the "recursiveArgumentsData". It should be noted that the final process of the conversion of the AST to an SQL query was not conducted. Calling the "walkRecursiveStatement()" method from the "SqlWalker" class would produce a full recursive SQL query, not necessarily supported by the intended database system. Therefore, during the emulation only a partial replacement of certain parts of AST is performed, following the procedure described below.

First the object of the Doctrine\ORM\Query\SqlWalker class changes the seed subquery of the With Recursive construct with an SQL Select query. What follows is this query's execution and retrieval of its results. At this stage the SqlWalker already knows the number and the types of the arguments of the DQL recursive query by analysing the objects from the SELECT clause of the seed query. We also know the relationships between DQL function arguments, SQL function arguments and the expressions from the SELECT clause of the seed query. Those data are passed to the emulator, which then may generate "on-the-fly" a mapping between a temporary table object and a temporary table within the relational DBMS.

The fact that the DBMS may refer to the temporary table through a SQL query has certain consequences. An array in PHP that stores operating data in a recursive step, must have its counterpart in the database system. Therefore the emulator, after processing non-recursive part of the query, receives information about the types of the arguments of the recursive function, and therefore also about the temporary table types. Thus, the next step is to form a suitable mapping table between the temporary PHP object and the temporary table in the database. This results in the creation of the instance of a Doctrine\ORM\Query\RecursiveEmulator\TemporaryTable type, which uses this mapping. Entering, changing, or deleting data in a PHP array maintained in this object also results in making appropriate changes in the temporary table in the database. In addition to both temporary structures, an instance of Doctrine\ORM\Query\RecursiveEmulator\Table is created. It gathers all the results produced in every recursive step. It is not linked to any structure in the database.

Third step is the translation of the recursive part of the DQL's RCTE statement into a single SQL SELECT query. In this new query we may find references the recursive function and its arguments. Therefore, the temporary table has been named reflecting the name of the function, and its column were named reflecting the function arguments' names. Also, the types of the columns match the types of arguments calculated by the SQL query, and "guessed" by the "MetadataGuesser" object. The SQL query will access the data stored in the temporary table. Both in the first and in the second step, if the seed and the recursive subquery are joined using "UNION" keyword, recurring results are removed from both the temporary array of results and the working temporary table. In the case of "UNION ALL" - they will be saved.

The fourth and final execution step is the processing of the outer Select query. Its result is stored using the Doctrine\ORM\Query\RecursiveEmulator\TemporaryTable object. Up to this point, all calculated data are stored in temporary tables of the relational DBMS. After the final query execution and results' retrieval, temporary tables and PHP arrays are removed to release the name of the recursive function and free resources.

5. PERFORMANCE

In this section we will show how the proposed solution affects the hierarchical data processing speed. Tests were performed on a desktop class computer.

The tests were repeated many times, and presented results are the average values. The prototypes were tested on two open-source database management systems MySQL and PostgreSQL, and on a selected popular commercial system, which we shall denote DBMS X. For each DBMS the tests were conducted with the same sets of data. The results for each DBMS are gathered in separate tables. In conducted tests the size of recursive data ranges from a few to 65,000 records. This set clearly shows the benefits of the proposed solutions for DBMSs implementing recursive queries.

For each DBMS the results are summarized in corresponding tables. The first column shows the number of records in the tree, the second column the tree depth, the third column the execution time of naive data querying (ie. every result becomes a source for a new query executed in a loop). The fourth column presents the recursive query execution time – for PostgreSQL and DBMS X databases, whereas for the MySQL the results of the recursive query emulation algorithm from Section 3. Column 5 shows the rate of acceleration resulting from the use of the recursive queries technique.

Table 1Results for MySQL database

Rec qty	Tree depth	Naive	Recursion	Ratio
15	4	0.018 s	0.06 s	333 %
156	4	0.055 s	0.13 s	236 %
16276	4	4.8 s	5.26 s	110 %
255	8	0.08 s	0.21 s	263 %
8191	13	3.06 s	2.36 s	77 %
65535	16	27.7 s	24.1 s	87 %

These tests show that in the case of a DBMS not supporting RCTEs, the unrolling of recursion brings little effect for larger data and slows down the parsing of the small data sets. More tests on using the recursive query unrolling can be found in [8]

The results for the PostgreSQL database, which supports recursive queries, present as following:

Table 2 Results for PostgreSQL database

Rec qty	Tree depth	Naive	Recursion	Ratio
15	4	0.02 s	0.08 s	400 %
156	4	0.11 s	0.11 s	100 %
16276	4	11.8 s	1.89 s	16 %
255	8	0.23 s	0.08 s	35 %
8191	13	8.67 s	0.73 s	8 %

Rec qty	Tree depth	Naive	Recursion	Ratio
65535	16	79.9 s	9.65 s	12 %

In PostgreSQL using recursion results in an important initial overhead, but soon with the increase in the size of the data we witness the increase in efficiency, which stabilizes at bigger data sets with a tenfold acceleration.

Table 3 Results for DBMS X

Rec qty	Tree depth	Naive	Recursion	Ratio
15	4	0.08 s	0.07 s	87 %
156	4	0.11 s	0.11 s	100 %
16276	4	15.2 s	1.76 s	11 %
255	8	0.29 s	0.10 s	34 %
8191	13	7.37 s	0.85 s	11 %
65535	16	62.0 s	11.3 s	18 %

In DBMS X the test results were similar to PostgreSQL, what allows us to assume that similar results would be obtained for other DBMSs implementing recursive CTEs

An interesting observation independent of the DBMS is the fact that the execution times seem to be insensitive to the depth of the hierarchical structures. Everywhere the results for the tree that contains 16,000 nodes with the depth of 4 were much slower than the time results for the query searching through 8000 nodes in the tree of the depth of 13.

6. CONCLUSIONS AND RELATED WORK

In this paper we have presented a proposal to extend DQL with recursive facilities based on SQL:99 Recursive Common Table Expressions. We have also presented the method of emulating such queries for Database Management Systems that do not support them. In order to encourage potential users we implemented a prototype mapper module that processes DQL queries enriched with WITH RECURSIVE clause. The result of performance tests conducted for this prototype emphasize the value of the proposed improvement. Compared to naive 3GL code we can achieve orders of magnitude improvement using our solution. Test have also shown that recursive queries are usually, but not always, the best choice for DBMSs that support them. Also our prototype requires the least modifications to the Doctrine framework. For the DBMSs that do not support recursive queries we provide the emulator to uphold one of the main postulates of the Doctrine ORM. According to it, each DQL query should be properly executed in each supported relational DBMS and return identical results. Unfortunately the costs of emulation (temporary table size, multiple requests to the database during recursive query execution) significantly impact the benefits of the usage of recursive queries. On the other hand the usage of recursive queries in comparison to emulator in DBMSs supporting RCTE gave significantly better execution times, mainly due to elimination of multiple database requests and better utilization of database built-in optimization methods.

This paper concludes the research conducted in the field of providing a programmer with benefits of recursive queries. Other research topics dealt with Oracle's Connect by constructs [7], enhancing different ORMs with RCTE supports [9,10], unrolling of recursive queries to support DBMSs without RCTE support [8], and benefits of utilizing

recursion in different cases [11,12] and with different data sets. We have also studied efficiency of execution of recursive queries in different DBMSs [1]. Interestingly, during the time the research took place we have witnessed growing demand for recursive query support [13,15] and enhancement of ORM capabilities [14]. Not only more DBMS support RCTEs than initially, but also SQL Alchemy ORM [16] implemented support for such queries in 2012 whereas our corresponding work dated 2010. Thus, we may conclude, that the research in this field was beneficial.

LITERATURE

1. Boniewicz A., Burzańska M., Przymus P., Stencel, K.: Recursive query facilities in relational databases: a survey, In DTA(2010), CCIS 118, 89-99.
2. Hibernate ORM, http://hibernate.org/orm/
3. Django Framework, https://www.djangoproject.com/
4. Entity Framework, https://msdn.microsoft.com/en-us/data/ef.aspx
5. Active Records, http://api.rubyonrails.org/classes/ActiveRecord/Base.html
6. Doctrine Project, http://www.doctrine-project.org/
7. Szumowska, A., Burzańska, M., Wiśniewski, P., Stencel, K.: *Extending HQL with plain recursive facilities*. In Morzy, T., Harder, T., Wrembel, R., eds.: ADBIS (2). Volume 186 of Advances in Intelligent Systems and Computing., Springer (2012) 265-272.
8. Boniewicz, A., Stencel, K., Wiśniewski, P.: *Unrolling SQL:1999 recursive queries.* In Kim, T.h., Ma, J., Fang, W.c., Zhang, Y., Cuzzocrea, A., eds.: Computer Applications for Database, Education, and Ubiquitous Computing. Volume 352 of CCIS (2012) 345-354.
9. Szumowska, A., Burzańska, M., Wiśniewski, P., Stencel, K.: *Efficient implementation of recursive queries in major object relational mapping systems*. FGIT 2011, LNCS 7105,78-89.
10. Wiśniewski, P., Szumowska, A., Burzańska, M., Boniewicz, A.: *Hibernate the recursivequeries - defining the recursive queries using Hibernate ORM*, In Eder, J., Bielikova, M., Tjoa, A.M., eds.: ADBIS (2). Volume 789 of CEUR Workshop Proceedings.,CEUR-WS.org (2011) 190-199.
11. Gawarkiewicz, M., Wiśniewski, P.: Partial aggregation using Hibernate. FGIT 2011, LNCS 7105, 90-99.
12. Boniewicz, A., Gawarkiewicz, M., Wiśniewski, P.: Automatic selection of functional indexes for object relational mappings system. International Journal of Software Engineering and Its Applications 7 (2013).
13. Joshi, A., Kukreti, S.: *Object Relational Mapping in Comparison to Traditional Data Access Techniques.* International Journal of Scientific \& Engineering Research, Volume 5, Issue 6, June-2014.
14. M. Sysak, B. Zieliński, P. Kruszyński, Ś. Sobieski & P. Maślanka. "*Static Integration of SQL Queries in C++ Programs*". In Advances in Databases and Information Systems, Springer International Publishing, 2014, pp. 126-138.
15. C.M. Gersen, "*ORM Optimization through Automatic Prefetching in WebDSL*". PhD Thesis. TU Delft, Delft University of Technology, 2013.
16. SQL Alchemy. http://www.sqlalchemy.org/