Implementing Common Table Expressions for MariaDB

Galina Shalygina
Mathematics and Mechanics Department
Saint Petersburg State University
Saint-Petersburg, Russia

Boris Novikov
Mathematics and Mechanics Department
Saint Petersburg State University
Saint-Petersburg, Russia

***Abstract*—Common Table Expressions (CTEs), introduced in the SQL Standard 1999, are similar to subroutines in programming languages: they can be referenced from multiple places in a query and may refer to themselves, providing recursive queries. Although many RDBMSs implemented this feature, but not in full. Until recently MariaDB did not have this feature either. This article describes CTE overall and looks through some interesting cases that are implemented in MariaDB only, like non-linear recursion and mutual recursion. It also compares optimizations for non-recursive CTEs across different RDBMSs. Finally, the results of experiments comparing computation of recursive CTEs for MariaDB and PostgreSQL are presented.**

Keywords—common table expressions; optimization; MariaDB;

I. Introduction

In spite of the fact that first SQL Standard appeared in 1986, there was lack of standard recursive constructions for over a decade from this period. Nevertheless, as it was essential to write hierarchical queries in some cases, several DBMSs introduced their own recursive constructions. One of the most popular is CONNECT BY presented by Oracle in 1980's [3]. And still now, even after standard recursive construction common table expression (CTE) was officially introduced, there are many researches on presentation of recursive queries, especially on using Object-Relational Mapping to make recursion [6], [7], [8], [9].

First attempt of CTE was in the SQL Standard 1999 [1] and from this version of Standard CTE specification didn't change until the latest considered version SQL:2008. First implementation of CTEs dates back to 1997, to RDBMS DB2. Later there was a period of stagnation and only in 2003-2005 other RDBMSs started to implement CTE [2]. Nowadays CTE are supported by most major RDBMSs, but still none of them support all CTE features that are described in the SQL Standard. Not so long ago MariaDB introduced its own implementation of CTE on that one of the authors has worked with other MariaDB developers. This realization includes features that have never been introduced by other RDBMS, like mutual recursion and non-linear recursion. A little later MySQL Labs included in their code an implementation of CTE as well.

As soon as CTE is defined in a query, this CTE may be used several times in the same query, thus the role of CTEs is similar to that of subroutines in imperative programming languages.

The SQL Standard requires that results of the query execution are the same as if CTEs would be executed only once during the query processing. This requirement suggests a straightforward implementation of CTE that computes CTE as a separate query and materializes results in a temporary table. However, this implementation might result in a poor performance. It can happen in the case of CTE defining a large number of records in a query with additional conditions on CTE data restricting it to a single row.

An efficient implementation of CTE, even for non-recursive queries, is a non-trivial task.

These days more and more benchmarks include CTE usage. To compare, TPC-H 1999 Benchmark has no tests using CTE, while in TPC-H 2011 38 of 100 queries contain CTE [5]. To provide fast and low-cost CTE computation a lot of attention was given to researches on optimization techniques for CTE. Articles [10], [11], [12] provide such techniques for recursive CTEs. MariaDB also introduced its own optimization technique for non-recursive CTEs, that also concern non-mergeable views and derived tables.

In this article we will discuss the implementation of CTE due to MariaDB Server special characteristics. We will also overview different optimization techniques for non-recursive CTEs. Non-recursive CTEs are handled like derived tables, but recursive ones are a much more difficult case. Moreover, they are computed in a different way.

Firstly, it should be checked if recursion is linear (if there is no special variable set to work with non-linear recursion). Secondly, all mutual recursive CTEs are defined, if there are any. At the same time all Standard restrictions on CTEs are checked. Lastly, an important process of checking when recursion should be stopped takes place on the execution stage. As regards optimization techniques for non-recursive CTEs, we describe most popular of them with proper examples and compare different RDBMSs approaches.

The contributions of this work include:

- Techniques for efficient implementation of several special cases of CTEs
- Techniques for implementation of mutual recursion of CTEs
- An implementation of both non-recursive and recursive CTEs in MariaDB

This paper is organized as follows: Section 2 discusses non-recursive CTEs in general and Section 3 describes recursive ones. Also Section 3 shows how computation of recursive CTEs in MariaDB goes and looks through different recursion cases and how recursion stops. Section 4 compares optimizations in different RDBMSs and Section 5 presents the results of experiments on two RDBMSs MariaDB and PostgreSQL. In Section 6 conclusion is made.

II. Non-recursive CTE

CTE can be introduced as temporary result set, that is defined within the scope of a single SELECT, INSERT, UPDATE, DELETE or CREATE VIEW statement. In MariaDB CTE can be defined only in SELECT or/and CREATE VIEW statements.

Each definition of non-recursive CTE consists of obligatory *WITH* keyword, CTE header, optional column list and query specifying this CTE.

```
WITH expression_name [( column_name [,...] )]
AS <CTE_query_specification>
```

Non-recursive CTEs can be called 'query-local views' and are similar to derived tables. As well as derived tables they aren't stored in the database. This means that CTEs live only for the duration of the query where they were defined. However, unlike derived table, they can be referenced multiple times in the same query. Instead of redefining the same derived table every time CTE can be used with the aim of making query more readable.

In MariaDB after CTE identification all references on non-recursive CTEs are formed as references on derived tables. On further phases of contextual analysis, optimization and execution non-recursive CTEs are handled as derived tables. The only thing that should be checked is if there are any renamed columns in the CTE definition because derived table doesn't have such an opportunity.

III. Recursive CTE

The greatest advantage that using CTE provides is that it can reference to itself so the recursive query can be made.

Each definition of recursive CTE consists of obligatory *WITH RECURSIVE* keyword, CTE header, optional column list and seed and recursive parts specifying this CTE. Both seed part and recursive part can be defined by several SELECT statements and they should be joined by *UNION* or *UNION ALL* operations.

```
WITH RECURSIVE
expression_name [( column_name [,...] )]
AS ( [<seed_part>] UNION [ALL]
     <recursive_part> ) [,...]
```

A. Computation

At the first step all the components of a seed part are computed. At all further steps a recursive part is computed using result received from the previous step. The Standard requires that only linear recursion can be used. This means that on each step of recursion only those records can be used that are received from the previous step and have never been received before. In the case of linear recursion the process of computation of recursive CTE stops when there are no new records collected.

In MariaDB computation of recursive CTE goes according to the following scenario:

On the preparatory stage dependency matrices for all CTEs that are used in the query are built to find out recursive CTEs. Also on this stage it is checked if there are enough seed parts for recursive CTEs.

On the stage of the contextual analysis the structure of temporary tables where results will be saved is defined using a seed part query. Several temporary tables are created: table where a final result will be stored, table for the new records and tables for each reference to the recursive CTE. Further all Standard restrictions are checked on this stage.

Lastly, on the execution stage CTE is executed in cycle using temporary tables defined on the previous stage. At the beginning these temporary tables contain the result of execution of the seed part. On each iteration content of the table for new records is used as the entire for the recursive part. The result of the recursive part execution is added to the table where the final result is stored and new received lines are written in the table for the new lines. If there is no data in the table for the new lines, the process ends. Otherwise, the result is added to the tables for the recursive links.

As stated before, there is one temporary table for each reference to the recursive CTE. All these tables are equal to each other, but storing them all is determined by characteristics of MariaDB server. Later an optimization which solves this problem will be made.

Also it must be said, that reference for the recursive CTE computed once will never be computed again.

B. Non-linear recursion

Non-linear recursion as it was said before is forbidden by Standard. However, MariaDB supports this feature.

Non-linear recursion can be used when some Standard restrictions on recursive CTEs need to be ignored. So, non-linear recursion can be used when:

- there is more than one reference on recursive CTEs in a FROM clause of *recursive_part* of CTE;
- there are some references on recursive CTEs in the right part of LEFT JOIN or in the left part of RIGHT JOIN;
- there are some references on recursive CTEs in a subquery that is defined in a WHERE clause of some recursive CTE;

The main difference in computation of non-linear recursion from linear one is that on each iteration as the entries for a recursive part not only last received records are used, but all received records. So, in MariaDB implementation of CTE, a table for new lines contains the same data as the table for the final result but with new lines added. The recursion stops when the comparison of these two tables shows that they are the same.

Whereas in some cases a query looks cleaner and executing converges faster, usage of non-linear recursion needs careful user's control. For instance, using UNION ALL should be checked by the user himself to prevent the same results on each step of recursion. It will be better to use UNION in this case.

If the user is ready to use non-linear recursion in MariaDB, he can set *@@standard_compliant_cte=0* and work with it.

C. Mutual recursion

Mutual recursion is said to be one of the most interesting forms of recursion. It is such a form where two or more CTEs refer to each other.

In MariaDB all mutual recursive CTEs are searched on the preparatory stage. For every *recursive_part* of CTE a search is made for all *recursive_parts* mutually recursive with the CTE where it was defined. It must be said that recursive CTE is mutually recursive with itself. All found mutually recursive CTEs are linked into a ring chain. Further, it is checked if there are enough *seed_parts* for a mutually recursive group. There should be at least one anchor for the mutually recursive group.

Mutual recursion is allowed by Standard, but restricted in the way that it can be transformed into common recursion. An example of the non-restricted case of mutual recursion can be this: when there are two recursive CTEs, where on each iteration one CTE waits until second one ends computation with the content of first CTE as entire, and only after that goes to the next step. MariaDB supports Standard version. It also must be said that MariaDB is the first RDBMS that implemented mutual recursion and the only one who did it at the time of writing.

D. How recursion stops

In MariaDB using linear recursion during a tree or a directed acyclic graph walking execution is guaranteed to stop. However, in some other cases the user has to make some conditions to prevent a looped process.

When a transitive closure is computed, in the definition of recursive CTE only UNION can be used to join recursive and seed parts. For instance, when the user needs to find all cities that he can reach from some place by bus, there can be more than one bus route with the same point of arrival. If there are such routes and the user applies UNION ALL, some destinations will be added repeatedly and bus routes will be searched again. This will lead to infinite process.

In the case when the paths over the graph with the loops need to be computed, for example, all paths consist of cities that can be reached by bus from some place, there might be special condition written into WHERE in the recursive part of CTE definition to stop influentially increasing paths computing. As well as in the previous case there can be some bus routes with the same destination. Adding a city that already exists in the path will lead to overlooped process, that's why a condition that checks if city exists in the path needs to be written.

Also in MariaDB there is a safety measure to control infinite process - special variable *@@max_recursive_iteration*s that controls count of iterations during recursive CTE computing. The user can change it himself if needed.

IV. Optimizations

A basic algorithm of CTE executing stores results of CTE in a temporary table. When the query where CTE was defined calls for CTE results, information is taken from this temporary table. Although this algorithm always works, in most cases it is not optimal. Some optimization techniques on non-recursive CTEs are discussed and a comparison between different RDBMSs approaches is made below.

A. CTE merging

During this optimization CTE is merged into parent's join such that parts of CTE definition replace corresponding parts of the parent query. There are some restrictions on CTE so that it can be merged: GROUP BY, DISTINCT, etc can't be used in CTE definition.

This optimization technique is the same as ALGORITHM=MERGE for views in MySQL.

On the Fig. 1 the example of how this technique works is shown. Upper listing shows the initial query and lower shows how optimizer will transform it.

```
WITH engineers AS (
 SELECT *
 FROM employees
 WHERE dept = 'Development')
SELECT *
FROM engineers E, support_cases SC
WHERE E.name = SC.assignee AND
      SC.created = '2016-09-30' AND
      E.location = 'Amsterdam'
```

```
SELECT *
FROM employees E, support_cases SC
WHERE E.dept = 'Development' AND
      E.name = SC.assignee AND
      SC.created = '2016-09-30' AND
      E.location = ’Amsterdam’
```

Fig. 1. Example of CTE merging.

B. Condition pushdown

A condition pushdown is used when merging is not possible, for example when CTE has GROUP BY. Conditions in the WHERE clause of a query that depend only on the columns of CTEs are pushed into the query defining this CTEs. In the general case conditions can be pushed only in the HAVING clause of the CTEs, but at some conditions it makes sense to push them into the WHERE clause. As a result, a temporary table is made smaller.

Besides CTEs, this optimization works for derived tables and non-mergiable views.

On the Fig. 2 the example of how this technique works is shown. Upper listing shows the initial query and lower shows how optimizer will transform it.

```
WITH sales_per_year AS (
 SELECT year(order.date) AS years,
        sum(order.amount) AS sales
 FROM order
 GROUP BY year)
SELECT *
FROM sales_per_year
WHERE year IN ('2015','2016')
```

```
WITH sales_per_year AS (
 SELECT year(order.date) AS years,
        sum(order.amount) AS sales
 FROM order
 WHERE year IN ('2015','2016')
 GROUP BY year)
SELECT *
FROM sales_per_year
```

Fig. 2. Example of condition pushdown.

C. CTE reuse

The main idea of this method is to fill CTE once and then use multiple times. It works with condition pushdown only in difficult cases, for instance, when CTE is used in different parts of the query with different restrictions. In this case disjunction of these conditions can be pushed into CTE.

D. Comparison of optimizations in MariaDB, PostgreSQL, MS SQL Server, MySQL 8.0.0-labs

MariaDB as MS SQL Server supports merging and condition pushdown. PostgreSQL supports reuse only. MySQL 8.0.0-labs supports both merging and reuse and it works in such way: it tries merging otherwise makes reuse.

TABLE I. EXISTENCE OF OPTIMIZATION TECHNIQUES IN DIFFERENT RDBMSS

DBMS	Optimization technique exists		
	CTE merge	***Condition pushdown***	***CTE reuse***
MariaDB 10.2	yes	yes	no
MS SQL Server	yes	yes	no
PostgreSQL	no	no	yes
MySQL 8.0.0-Labs	yes	no	yes

V. THE RESULTS OF EXPERIMENTS ON MARIADB AND POSTGRESQL

Tests have been conducted on the computer with processor Intel(R) Core(TM) i7-4710HQ CPU, 2.50GHz, 6144 KB cache size, 7 GB RAM on Opensuse 13.2 operating system. We tested following database systems: PostgreSQL 9.3 and MariaDB 10.2.

The experiments were made in a database containing the information about domestic flights in the USA during 2008. Database schema consists of the following relations:

- tab_2008(month, dayofmonth, dep_time, arrtime, flightnum, airtime, origin, dest, dist);
- airports(names);

We wanted to find multi-destination itineraries. So, we decided to find the shortest way between the airports of interest by plane. The table *airports* shows which airports should be visited. None of the airports can be visited twice. Besides, the plane should leave for the next destination a day or more after the previous plane.

The following query *Q1* for MariaDB is shown on Fig.3. Script for PostgreSQL has a difference in functions *cast(origin as char(32))* and *locate(tab_2008.dest, s_planes.path)*. Analogue of *locate* function in PostgreSQL is *position* function and its return type is *text*, that’s why the result of *cast* function in PostgreSQL in this query will be not *char(32)*, but *text*.

This query starts from 'IAD' airport in *seed_part* and looks through the table *tab_2008* to find flights with 'IAD' as origin and one of the airports from table *airports* as destination. As needed destination is found it is checked if it has already been visited on the route to prevent repeats. From the received data we take only those paths that involve all airports from table *airports* and have the smallest overall distance.

We've made a query *Q1* on table *tab_2008* consists of different number of records: 587130, 1134055 and 6858079. The results of the experiments are shown in TABLE II.

What can be seen from the table is that the results of tests in MariaDB are more than in three times better than in PostgreSQL.

```
WITH RECURSIVE s_planes (path, dest, dayofmonth,
dist, it) AS (
 SELECT cast(origin as char(30)), origin,
        dayofmonth, 0, 1
 FROM tab_2008
 WHERE dayofmonth = 3 AND origin = 'IAD' AND
       flightnum = 3231
 UNION
 SELECT
  concat(s_planes.path,',',tab_2008.dest),
  tab_2008.dest, tab_2008.dayofmonth,
  s_planes.dist+tab_2008.dist, it+1
 FROM tab_2008, airports, s_planes
 WHERE
  tab_2008.origin = s_planes.dest AND
  locate(tab_2008.dest, s_planes.path)=0 AND
  tab_2008.dest = airports.name AND
  tab_2008.dayofmonth > s_planes.dayofmonth)
SELECT *
FROM s_planes
WHERE it = 8 AND
      dist = (SELECT min(dist)
              FROM s_planes
              WHERE it = 8);
```

Fig. 3. Query Q1 for MariaDB

TABLE II. THE RESULTS OF THE QUERY Q1 (OVERALL RESULT)

DBMS	Records count		
	587130	*1134055*	*6858079*
MariaDB	16.72 sec	31.97 sec	3 min 9 sec
PostgreSQL	60.29 sec	1 min 91 sec	11 min 50 sec

TABLE III. THE RESULTS OF THE EXPERIMENTS DURING AIRPORTS COUNT MINIMIZATION (OVERALL RESULT)

DBMS	Airports count				
	4	*5*	*6*	*7*	*8*
MariaDB	2.28 sec	3.49 sec	5.93 sec	14.45 sec	31.97 sec
PostgreSQL	1.13 sec	2.97 sec	6.74 sec	38.66 sec	1 min 91 sec

Also this query *Q1* was made on the same tables with different number of records but with index on *dest* column. Optimizer didn't access index so index existence didn't affect on results of the query and further experiments were made on tables without indexes.

Although, we decided to make some other experiments and minimize the count of searched airports. So, less steps of recursion will be made. We made these experiments only on the table with 1134055 records. The results are shown in TABLE III.

When the count of searched airports is 8, MariaDB has much better results than PostgreSQL. But during the minimization of airports count PostgreSQL results become closer and closer to MariaDB ones. When the count of airports is less than 5 PostgreSQL results become better than MariaDB and this trend continues.

VI. CONCLUSION

In this paper we presented a number of techniques for execution of CTEs and provided an implementation of these techniques for MariaDB. We described in details recursive CTE computation and mutual recursive CTE computation. Also we discussed in which cases non-linear recursion can be used. We compared existence of optimization techniques for non-recursive CTE in different databases.

What is more, we made experiments using flights table on PostgreSQL and MariaDB. They showed that PostgreSQL has better results only when few steps of recursion are needed. For the long recursive process on a huge amount of data MariaDB is a better choice.

Authors want to gratitude MariaDB developers Igor Babaev and Sergey Petrunya for their participating in work on MariaDB CTE implementation and help in writing this article.

REFERENCES

[1] SQL/Foundation ISO/IEC 9075-2:1999.

[2] P. Przymus, A. Boniewicz, M. Burza´nska, and K. Stencel. Recursive query facilities in relational databases: a survey. In DTA and BSBT, pages 89–99. Springer, 2010.

[3] Oracle Database Online Documentation, 10g Release 2 (10.2)

[4] D. Stuparu and M. Petrescu. Common Table Expression: Different Database Systems Approach. Journal of Communication and Computer, 6(3):9–15, 2009.

[5] TPC Benchmark™DS (TPC-DS): The New Decision Support Benchmark Standard.

[6] 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 Communications in Computer and Information Science. Springer Berlin Heidelberg (2012) 345–354.

[7] Szumowska, A., Burzańska, M., Wiśniewski, P., Stencel, K.: Efficient Implementation of Recursive Queries in Major Object Relational Mapping Systems. In FGIT 2011 78-89.

[8] Burzanska, M., Stencel, K., Suchomska, P., Szumowska, A., Wisniewski, P.: Recursive queries using object relational mapping. In Kim, T.H., Lee, Y.H., Kang, B.H., Slezak, D., eds.: FGIT. Volume 6485 of Lecture Notes in Computer Science., Springer (2010) 42–50.

[9] Wiśniewski, P., Szumowska, A., Burzańska, M., Boniewicz, A.: Hibernate the recursive queries - defining the recursive queries using Hibernate ORM. In Eder, J., Bielikov´a, M., Tjoa, A.M., eds.: ADBIS(2). Volume 789 of CEUR Workshop Proceedings., CEUR-WS.org (2011) 190–199.

[10] Ghazal, A., Crolotte, A., Seid, D.Y.: Recursive sql query optimization with k-iteration lookahead. In Bressan, S., K¨ung, J., Wagner, R., eds.: DEXA. Volume 4080 of Lecture Notes in Computer Science., Springer (2006) 348–357.

[11] Ordonez, C.: Optimization of linear recursive queries in sql. IEEE Trans. Knowl. Data Eng. 22 (2010) 264–277.

[12] Burzanska, M., Stencel, K., Wisniewski, P.: Pushing predicates into recursive sql common table expressions. In Grundspenkis, J., Morzy, T., Vossen, G., eds.: ADBIS. Volume 5739 of Lecture Notes in Computer Science., Springer (2009) 194–205