Recursive Query Facilities in Relational Databases: A Survey

Piotr Przymus[1], Aleksandra Boniewicz[1], Marta Burzańska[1], and Krzysztof Stencel[1,2]

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland
[2] Institute of Informatics, University of Warsaw, Warsaw, Poland

Abstract. The relational model is the basis for most modern databases, while SQL is the most commonly used query language. However, there are data structures and computational problems that cannot be expressed using SQL-92 queries. Among them are those concerned with the bill-of-material and corporate hierarchies. A newer standard, called the SQL-99, introduced recursive queries which can be used to solve such tasks. Yet, only recently recursive queries have been implemented in most of the leading relational databases. In this paper we have reviewed and compared implementations of the recursive queries defined by SQL:1999 through SQL:2008 and offered by leading vendors of DBMSs. Our comparison concerns features, syntax and performance.

1 Introduction

Recursion is fundamental for common programming languages. People working with databases quickly realized that there are problems, such as bill-of-material or corporate hierarchy problem that cannot be efficiently solved using early SQL standards. Because there was a need for working with hierarchical data, several DBMS vendors have introduced proprietary ways of providing such functionality for their users. One of them was Oracle, which provided hierarchical queries using *Start With ... Connect By* construction in their DBMS version 5.0. The first DBMS to provide recursive Common Table Expressions was IBM's DB2 v.7 in 1997. The research on SQL's recursive CTEs have been greatly influenced by Datalog ([1], [2]). But it was ANSI SQL-99 standard that officially introduced recursive queries and views into SQL language. Throughout the next 10 years there has been little happening in the field of recursive SQL queries, especially in comparison to other relational database research. But with each passing year the problem of recursion became more popular with database vendors. Nowadays most major DBMSs support recursive CTE. In the 2003 Sybase released SQL Anywhere 9 - their first DBMS equipped with recursive CTEs. Two years later Microsoft included them within MS SQL Server 2005. Among other popular databases Firebird supports relational CTEs since 2008 (Firebird 2.1), PostgreSQL and Oracle since 2009 and their databases PostgreSQL 8.4 and Oracle 11gR2. The modern research on recursive queries focuses on optimization. The

Y. Zhang et al. (Eds.): DTA/BSBT 2010, CCIS 118, pp. 89–99, 2010.

papers [3,4,5] present propositions of new optimization techniques specially designed for recursive SQL-99 queries. However even though the topic of recursive queries in SQL is not something new, there is still no complete survey on the subject. The paper [6] provides a basic comparison of features and performance of recursive CTEs implemented by MS SQL 2005 and PostgreSQL 8.4 databases. Unfortunately this paper lacks a more general approach.

The latest public versions of the selected DBMSs have been tested: IBM DB2 Express-C 9.5, RDBMS X, Sybase SQL Anywhere 11, PostgreSQL 8.4, Firebird 2.1.3 and Microsoft SQL Server 2008. This paper is organized as follows: §2 discusses the general construction of recursive CTE in SQL; §3 compares syntax and features of the recursive queries; §4 presents the results of performance testing on those databases under two operating systems: MS Windows 7 and Ubuntu 9.10; § 5 concludes. Detailed data from the experiments is presented in the Appendix.

2 Recursive Common Table Expressions

In SQL a recursive query specifies a temporary result set known as a recursive Common Table Expression (CTE). It is usually defined within the scope of a SELECT query, but may also be used with INSERT, UPDATE, MERGE or DELETE statements. The recursive clause itself may be defined explicitly as a CTE or it may be used in a CREATE VIEW statement as a part of its definition. Each recursive CTE definition consist of a CTE header, a seed query and a recursive query. The general syntax of a recursive SQL query is presented below:

WITH RECURSIVE cte_0 $(A_{01}, \cdots, A_{0n})$ **AS**
(*seed_query* **UNION ALL** *recursive_query* additional_clauses),
[RECURSIVE] $cte_1(A_{11}, \cdots, A_{1n})$ ... *outer query with* cte_i $(i \geq 0)$

In the example given above the CTE header is formed out of the *WITH RECURSIVE* keywords, the CTE's name and a list of columns. Both seed query and a recursive query may consist of multiple SELECT statements, all of them should be connected using *UNION ALL* operator. The seed query forms the initial set of tuples which will be used by the recursive query. The SQL standard disallows non-linear recursion thus the recursive query is restricted to exactly one reference to the CTE. The CTE definition may contain additional clauses that provide additional functionality. For example in order to specify the search order one may use the SEARCH clause with DEPTH FIRST or BREADTH FIRST keywords. The CYCLE clause may be used to detect cycles that could result in loops.

There are subtle differences in the implementations of the recursive queries provided by various vendors. The next section discusses those issues in detail.

3 Features Comparison

This section's aim is to compare syntax differences, available features and limitations between databases. The Table 1 contains compact information about

Table 1. Features Comparison

Features		MS SQL	SQL Anywhere	PostgreSQL	DB2	RDBMS X	Firebird
Multiple queries in	Initial step	Y	Y	Y	Y	N*	Y
	Recursive step	Y	N	N	Y	N	Y
Other (then UNION ALL) set operators	Between initial queries	Y*	Y*	Y*	Y	-*	Y*
	Between recursive and initial queries	N	N	Y*	N	N	N
	Between recursive queries	N	-	-	N	-	N
Referencing CTE	Recursive CTE to other recursive CTE	Y	Y*	Y	Y	Y	Y
	Recursive CTE to non-recursive CTE	Y	Y	Y	Y	Y	Y
	Non-recursive CTE to recursive CTE	Y	Y	Y	Y	Y	Y
	Mutual recursive CTE	N	N	N	N	N	N
Using Recursive CTE in	SELECT	Y*	Y*	Y	Y	Y	Y
	INSERT	Y	Y*	N	Y	Y	N
	UPDATE	Y	N	N	Y*	Y	N
	DELETE	Y	N	N	N	Y	N
	CREATE VIEW	Y	Y*	N	N	Y	N
	OTHER	Y*	N	N	N	Y*	N
Various	Protection from infinite loops	Y*	Y*	N	N	Y	Y*
	Search clause	N	N	N	N	Y	N
	Cycle clause	N	N	N	N	Y	N
Recursive query part limitations	Group by	N	N	N	N	N	N
	Having	N	N	N	N	N	N
	Order by	N	N	N	N	Y	Y
	Distinct	N	N	Y	N	N	N
	Non aggregate functions	Y*	Y	Y	N	Y*	Y
	Agregate functions	N	N	N	N	N	N
	Recursive in subquery	N	N	N	N	N	N
	Left outer join	N	Y*	Y	N	Y*	N
	Right outer join	N	Y*	Y	N	Y*	N
	Full outer join	N	N*	N	N	N	N
	Limiting results	N	Y	N	Y	Y	Y
Initial subquery part support for	Group by, Order by, Distinct, Having, Aggregate and Non aggregate Functions	Y	Y	Y	Y	Y	Y
Syntax	obligatory RECURSIVE keyword	N	Y	Y	N	N	Y
	obligatory column list after WITH	?	Y	N	Y	Y	N

features and information about support in specified databases ("Y" - yes, "N" - no or "-" not applicable, "*" - additional information is in Table 2). The comparison has been made based on the product documentation, supported error codes and results of list of control queries. **Impact of recursive CTE features on RDBMS functionality**: Multiple queries in the seed part of the CTE are supported in almost every tested database, only RDBMS X does not support this feature. Multiple queries in the recursive part are more troublesome. Only half of the database vendors decided to implement this feature. The lack of this feature

Table 2. Features Comparison-Details

Database	Feature	Description
MS SQL Server 2008	Protection from infinite loops	Query hint MAXRECURSION from 0 to 32767
	Other (then UNION ALL) set operators in CTE between initial queries	UNION, EXCEPT, or INTERSECT
	Recursive subquery limitations - Non aggregate functions	Functions not allowed: functions with parameters, functions with side effects
	Using Recursive CTE in OTHER	MERGE
SQL Anywhere	Protection from infinite loops	Database option max_recursive_iterations(default 100). Additionally two connection type options appear MAX_TEMP_SPACE prevents the server from utilizing an unlimited amount of space for the TEMP file. TEMP_SPACE_LIMIT_CHECK limits temporary space usage on a connection-by-connection basis.
	Other (then UNION ALL) set operators in CTE between initial queries	UNION, EXCEPT, or INTERSECT
	LEFT,RIGHT OUTER JOIN	the reference to the recursive table cannot appear on the null-supplying side of an outer join
	Referencing CTE recursive CTE to other recursive CTE	Works, but documentation states differently
Postgre SQL	Other (then UNION ALL) set operators in CTE between initial queries	UNION, EXCEPT, or INTERSECT
	Other (then UNION ALL) set operators in CTE between recursive and initial queries	UNION
	LEFT,RIGHT OUTER JOIN	the reference to the recursive table cannot appear on the null-supplying side of an outer join
DB2	Using Recursive CTE in UPDATE	Non official way (but suggested by a DB2 developer) WITH v **AS** (..)**SELECT COUNT**(∗)**FROM** OLD **TABLE**(**UPDATE** ..)
RDBMS X	Recursive query part analytic functions are permitted	limitations on analytic functions are permitted
	Multiple queries in initial step	Feature described in documentation but not implemented
	Other (then UNION ALL) set operators in CTE between initial queries	Feature described in documentation but not implemented. Documentation mentions UNION, INTERSECT and MINUS.
	LEFT,RIGHT OUTER JOIN	the reference to the recursive table cannot appear on the null-supplying side of an outer join
	Using Recursive CTE in OTHER	CREATE TABLE
Firebird	Other (then UNION ALL) set operators in CTE between initial queries	UNION
	Protection from infinite loops	Hardcoded recursion depth 1024

probably could be overcome by a user with other language constructions, but this would lead to overcomplicated solutions. Therefore, native support would be appreciated since no obvious difficulties are involved.

All databases supporting multiple initial queries are allowing other than UNION ALL set operators in the initial queries part. On the other hand, there is a lack of other set operators in the recursive part. Additionally only PostgreSQL supports other than UNION ALL operators (in this case UNION) between initial and recursive part. This construction may be used as some form of protection against infinite loops, but it will work only in some cases (for example when cycle is a result of duplicate rows).

Usage of CTE in various SQL statements slightly differs for each SQL dialect. Lack of support for some statements (especially UPDATE or DELETE) may be burdensome and may lead to strange, non-standard solutions (see row 4 of the Table 2). Differences are probably engendered by general implementation of CTE and not related strictly to recursive extension.

Protection from infinite loops is a major feature. Data with cycles may cause unconstrained reachability queries. There is a serious threat for stability of databases without this protection. From the authors' experience such cases may end with depletion of system resources. Only four databases assure such protection.

Cases of support for DISTINCT (PostgreSQL) and ORDER BY (RDBMS X and Firebird) clauses are worth mentioning and further investigation. As OUTER JOIN may lead to various problems, there are different policies for using them. However, after restricting OUTER JOIN construction there is no technical reason to forbid them.

4 Performance Tests

Tests have been conducted on two identical computers equipped with processor Intel(R) Core(TM)2 Quad CPU Q9400, 2.66GHz, 3072 KB cache size, 4 GB RAM and 320 GB hard-drive. Used operating systems: Windows 7 Enterprise 64bit, and Ubuntu 9.10 Server Edition. We tested following databases: MS SQL Server 2008 Express, PostgreSQL 8.4, DB2 Express-C 9.7, Firebird 2.1.3, Sybase SQL Anywhere 11, RDBMS X. RDBMS X on Windows does not support SQL-99 compliant recursive queries and MS SQL Server is only available on MS Windows, so they were tested only on Linux and Windows respectively.

Database schema consists of the following relations:

- CITIES(cid, city) contains 200 records,
- TRAINS(departure, arrival, railline, tid, price) contains 800 records,
- FLIGHTS(departure, arrival, carrier, fid, price) contains 800 records.

Each query was executed 6 times. The average duration of the query was measured. The tests were conducted for the data containing cycles and without cycles, on both with and without indexes. Because some databases automatically place indexes on key columns, by "no indexes" we mean that no indexes have been placed manually - a default situation after simple CREATE TABLE command. Indexes were placed on: the cid column from relation CITIES, departure and arrival columns from TRAINS and FLIGHTS.

We have identified two kinds of problem being solved by recursion: hierarchy (like in corporate hierarchy or bill-of-material) and arbitrary graph (like in transport connections). Therefore, we have prepared two sets of data: the one without cycles and the one with cycles. In our opinion those cover most of the sorts of problems the recursive queries have been designed to solve.

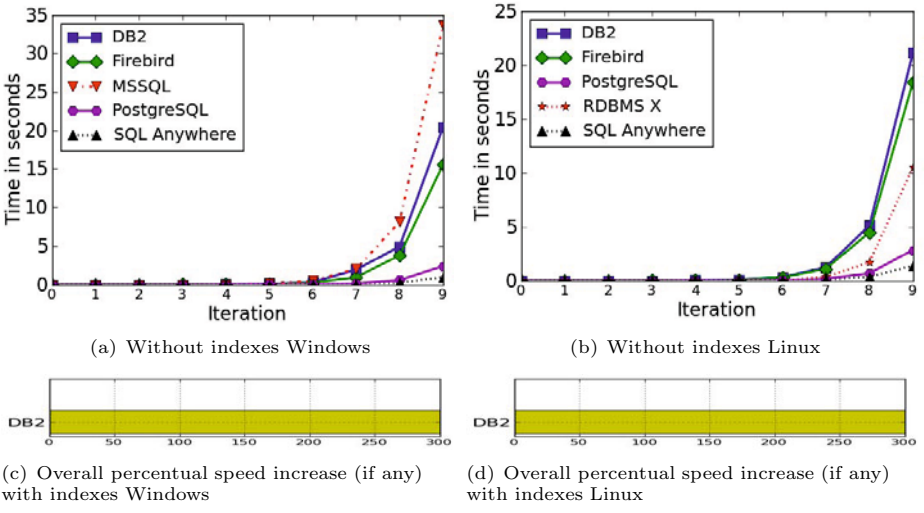
The cyclic data are tested with the parameter $I = 1 \cdots 9$ which limits the recursion depth. For $I = 9$ it was enough to fully exhaust the system resources and in many cases to crash of the database system.

4.1 Test Suite 1

Query Q1 displays all the cities reachable by plane starting from Toronto, the number of connections is limited by the parameter I. This query has been tested on: Windows - DB2, Firebird, MS SQL, PostgreSQL, SQL Anywhere and Ubuntu - DB2, Firebird, PostgreSQL, RDBMS X and SQL Anywhere.

```
WITH destinations (origin, departure, arrival, connections) AS
  (SELECT a.departure, a.departure, a.arrival, 1
   FROM flights a, cities c
   WHERE a.departure = c.cid AND c.city = 'Toronto'
  UNION ALL
   SELECT r.origin, b.departure, b.arrival, r.connections + 1
   FROM destinations r, flights b
   WHERE r.arrival = b.departure AND r.flight_count < I )
SELECT count(*) FROM destinations
```

Listing 1.1. Query Q1



(a) Without indexes Windows

(b) Without indexes Linux

(c) Overall percentual speed increase (if any) with indexes Windows

(d) Overall percentual speed increase (if any) with indexes Linux

Fig. 1. Result of the query Q1 for data with cycles

For data without cycles query Q1 completed instantly for $I = 0 \cdots 9$ with and without indexes.

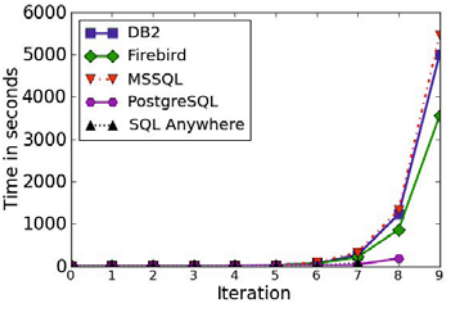
SQL Anywhere had the best performance on both OSs. Only for the DB2, placing indexes made a huge speed improvement, for data without indexes full table scans were used, while after adding indexes, light index scans and immediate tuple fetching occurred.

4.2 Test Suite 2

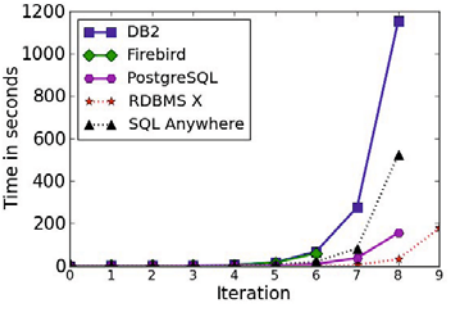
Query Q2 has been tested on the same set of databases as Q1. Q2 displays all the cities reachable by plane starting from Toronto or Warsaw, the number of connections is limited by the parameter I.

```
WITH destinations (departure, arrival, connections, cost) AS
  (SELECT a.departure, a.arrival, 0, price
   FROM flights a, cities c
   WHERE a.departure = c.cid
     AND c.city = 'Toronto' OR c.city = 'Warsaw'
  UNION ALL
   SELECT r.departure, b.arrival, r.connections + 1, r.cost + b.price
   FROM destinations r, flights b
   WHERE r.arrival = b.departure AND r.connections < 1)
SELECT count(*) FROM destinations
```

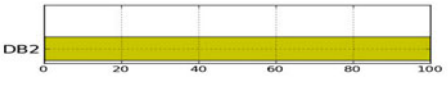
Listing 1.2. Query Q2



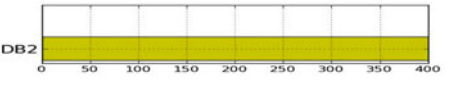
(a) Without indexes Windows



(b) Without indexes Linux

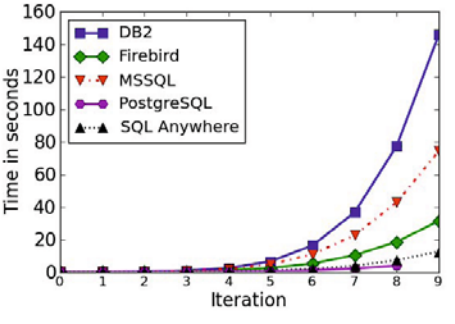


(c) Overall percentual speed increase (if any) with indexes Windows

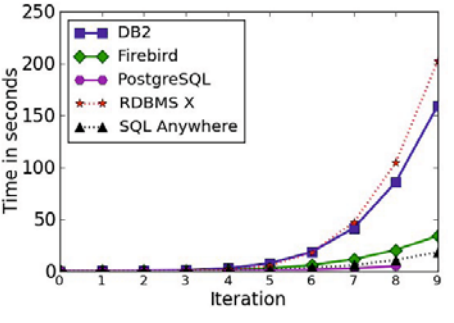


(d) Overall percentual speed increase (if any) with indexes Linux

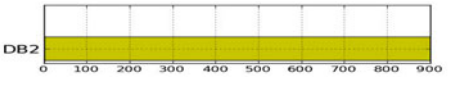
Fig. 2. Result of the query Q2 for data with cycles



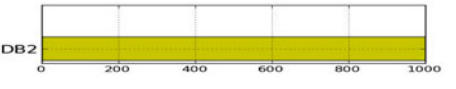
(a) Without indexes Windows



(b) Without indexes Linux



(c) Overall percentual speed increase (if any) with indexes Windows



(d) Overall percentual speed increase (if any) with indexes Linux

Fig. 3. Result of the query Q2 for data without cycles

For data with cycles the test results indicate that for the parameter $I = 0 \cdots 8$ PostgreSQL has the best execution times on Windows system, while RDBMS X has the best execution times on Ubuntu system.

On Ubuntu for $I = 9$ and data with or without indexes only RDBMS X completed successfully. While on Windows system only two databases returned errors for data without indexes(SQL Anywhere, PostgreSQL), and after adding indexes only one(PostgreSQL). All occurring errors where results of memory allocation error, despite of 4 GB RAM and 10 GB free hard drive space.

For $I = 0 \cdots 8$ and data without cycles PostgreSQL was leading database, unfortunately for $I = 9$ it was the only database that returned memory allocation error.

Only DB2 got huge performance bost from indexes, the same methods of optimization as in Q1 was used.

4.3 Test Suite 3

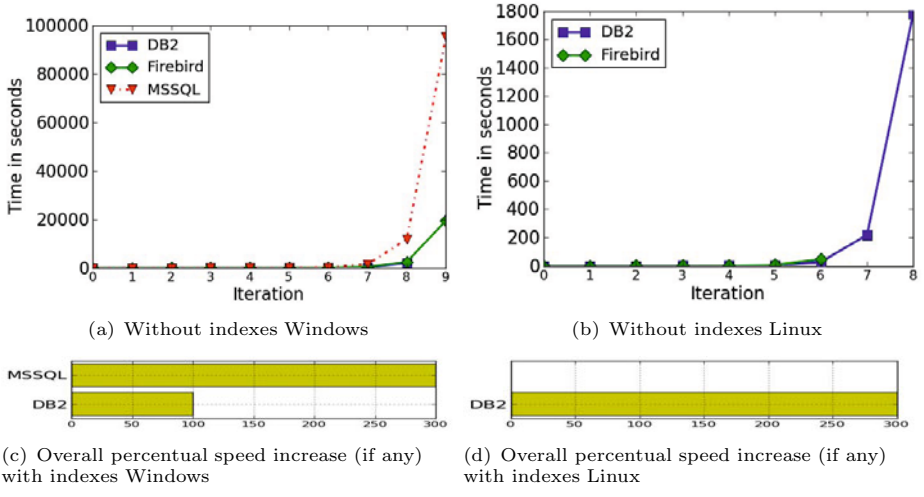
Query Q3 has been tested on databases Firebird, MS SQL and DB2 on Windows and Firebird and DB2 on Ubuntu (other databases do not support this query's syntax). This query displays all the cities reachable by planes and trains from Toronto, the number of connections is limited by the parameter I.

```
WITH destinations(departure, arrival, connections, flights, trains, cost
    ) AS
  (SELECT f.departure, f.arrival, 0, 1, 0, price
   FROM flights f, cities c
   WHERE f.departure = c.cid AND c.city = 'Toronto'
  UNION ALL
   SELECT t.departure, t.arrival, 0, 0, 1, price
   FROM trains t, cities c
   WHERE t.departure = c.cid AND c.city = 'Toronto'
  UNION ALL
   SELECT r.departure, b.arrival, r.connections+1 , r.flights+1, r.trains
       , r.cost + b.price
   FROM  destinations r, flights b
   WHERE r.arrival = b.departure AND r.connections<I
  UNION ALL
   SELECT r.departure, v.arrival, r.connections+1 , r.flights, r.trains +
        1, r.cost + v.price
   FROM destinations r, trains v
   WHERE r.arrival = v.departure AND r.connections<I )
SELECT count(*) FROM destinations
```

Listing 1.3. Query Q3

For $I = 6 \cdots 8$ DB2 gained best performance, but for $I = 9$ ended with memory allocation error on both OSs. On Ubuntu system for $I = 7$ Firebird returned memory allocation error. On Windows system for $I = 9$ with no indexes Firebird was stooped after 30 hours, after adding indexes, computation ended in reasonable time.

This time MS SQL clearly benefited from indexes. Analysis of the execution plan, indicates that for data without manually created indexes the clustered index scans were used, while with manually created indexes, index seek using user indexes.



(a) Without indexes Windows

(b) Without indexes Linux

(c) Overall percentual speed increase (if any) with indexes Windows

(d) Overall percentual speed increase (if any) with indexes Linux

Fig. 4. Result of the query Q3 for data with cycles

For data without cycles for all of the test cases the query Q3 completed instantly (0s with respect to measurement errors).

4.4 Test Suite 4

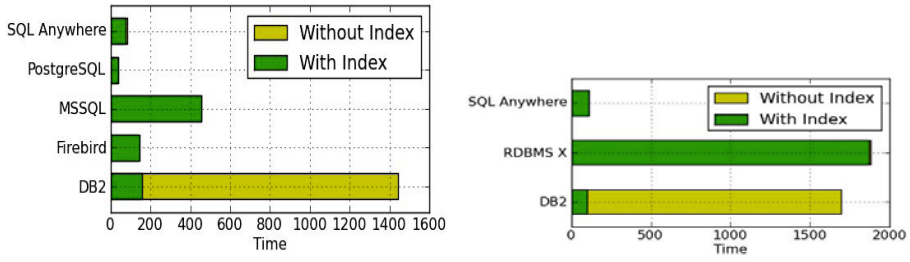
Query Q4 is similar to Q2 and has been tested on the same set of databases. This query displays all the cities reachable by plane starting from Toronto or Warsaw without placing limits on the number of connections. This query has been tested only for data without cycles.

```
WITH destinations (departure, arrival, connections, cost) AS
  (SELECT a.departure, a.arrival, 0, price
   FROM flights a, cities c
   WHERE a.departure = c.cid AND c.city = 'Toronto' OR c.city = 'Warsaw'
  UNION ALL
   SELECT r.departure, b.arrival, r.connections+1, r.cost + b.price
   FROM destinations r, flights b WHERE r.arrival = b.departure)
SELECT count(*) FROM destinations
```

Listing 1.4. Query Q4

On Windows best performance was gained by PostgreSQL, on Ubuntu SQL Anywhere showed best performance without indexes, and DB2 was the leading competitor for data with indexes. On Ubuntu OS Firebird ended exceeding limit level of depth, and PostgreSQL returned memory allocation error. Both errors occurred for data with and without indexes.

Only DB2 benefited from manually created indexes (with a huge performance boost), additional indexes resulted in replacement of expensive full table scans with operations of index scans and row fetching.



(a) Windows (b) Ubuntu

Fig. 5. Result of the query Q4 for data without cycles

4.5 Test Suite 5

For the pure purpose of testing the data independent query labeled Q5 has been executed on all of the tested databases. Its purpose was to calculate the sum of the numbers starting from 0 to 10 multiplied by $I = 0 \cdots 10$. Each time it was completed instantly (app. 0s).

```
WITH RECURSIVE t(n) AS ( VALUES (1) UNION ALL SELECT n+1 FROM t WHERE n
    < 100*I) SELECT sum(n) FROM t;
```

Listing 1.5. Query Q5

5 Conclusions

In this paper we have investigated the implementation of recursive SQL Common Table Expressions in most of the popular DBMSs. We have cataloged and compared common features of the syntax of such queries choosing SQL-99 standard as a base for comparison. Furthermore we performed performance tests on MS Windows and Linux systems, including cases of data with and without cycles and data independent queries. Also, we have checked whether placement of indexes would make a significant difference. Last, but not least, the schema, data and queries presented in this paper may be a starting point for development of a benchmark for recursive queries.

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS 1986, pp. 1–15. ACM, New York (1986)
2. Letuchy, D.: Recursive queries in sql. In: Proceedings of SYRCoDIS (2005)
3. Ordonez, C.: Carlos Ordonez. Optimization of linear recursive queries in sql. IEEE Trans. on Knowl. and Data Eng. 22(2), 264–277 (2010)
4. Burzańska, M., Stencel, K., Wiśniewski, P.: Pushing Predicates into Recursive SQL Common Table Expressions. In: Grundspenkis, J., Morzy, T., Vossen, G. (eds.) ADBIS 2009. LNCS, vol. 5739, pp. 194–205. Springer, Heidelberg (2009)

5. Ghazal, A., Crolotte, A., Seid, D.Y.: Recursive sql query optimization with k-iteration lookahead. In: Bressan, S., Küng, J., Wagner, R. (eds.) DEXA 2006. LNCS, vol. 4080, pp. 348–357. Springer, Heidelberg (2006)
6. Stuparu, D., Petrescu, M.: Common Table Expression: Different Database Systems Approach. Journal of Communication and Computer 6(3), 9–15 (2009)
7. Melton, J., Simon, A.R.: SQL: 1999: understanding relational language components. Morgan Kaufmann Pub., San Francisco (2002)