Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1439–1446

On Redundant Data for Faster Recursive Querying Via ORM Systems

Aleksandra Boniewicz
Faculty of Mathematics
and Computer Sciences
Nicolaus Copernicus University
Toruń, Poland
Email:grusia@mat.umk.pl

Piotr Wiśniewski
Faculty of Mathematics
and Computer Sciences
Nicolaus Copernicus University
Toruń, Poland
Email:pikonrad@mat.umk.pl

Krzysztof Stencel
Institute of Informatics
University of Warsaw
Warsaw, Poland
Email:stencel@mimuw.edu.pl

***Abstract*—Persistent data of most business applications contain recursive data structures, i.e. hierarchies and networks. Processing such data stored in relational databases is not straightforward, since the relational algebra and calculus do not provide adequate facilities. Therefore, it is not surprising that initial SQL standards do not contain recursion as well. Although it was introduced by SQL:1999, even now it is implemented in few selected database management systems. In particular, one of the most popular DBMSs (MySQL) does support recursive queries yet. Numerous classes of queries can be accelerated using redundant data structures. Recursive queries form such a class. In this paper we consider four materialization solutions that speed up recursive queries. Three of them belong to the state-of-the-art, while the fourth one is the contribution of this paper. The latter method assures that the required redundant storage is linearithmic. The other methods do not guarantee such a limitation. We also present thorough experimental evaluation of all these solutions using data of various sizes up to million records. Since all these methods require writing complex code if applied directly, we have prototyped an integration of them into Hibernate object-relational mapping system. This way all the peculiarities are hidden from application developers. Architects can simply choose the appropriate materialization method and record their decisions in configuration files. All necessary routines and storage objects are then generated automatically by the ORM layer.**

I. Introduction

DATA models of numerous business enterprises encompass recursive data structures in the form of hierarchies and networks. They store data on e.g. railway networks, bill of material and product categorization. Their actual storage format can be chosen from a plethora of proposals [1]. There are various ways to query such data. Obviously, a dedicated 3GL client code can be written. Then, the data processing is done on the client side. In this case a significant amount of complex source code must be created, debugged and maintained. This usually causes a noteworthy increase of the budget and a shift in the delivery schedule. Therefore, a server side solution is called for. It was proposed as extensions to SQL, e.g. Oracle's CONNECT BY clause or recursive Common Table Expressions eventually adopted in SQL:1999. Such extensions have been implemented in numerous database systems [2]. Simultaneously the academia worked on optimization methods for such queries [3], [4], [5]. However, there are still database managements systems that do not support recursion in queries, e.g. MySQL. Since they are widely adopted and used, applications programmers often face the question how to query their recursive data. As noted above, they can choose to hardcode suitable logic in the application. In spite of deceptive simplicity of this solutions, it causes merely troubles: lower efficiency, increased cost and complexity, as well as reduced maintainability.

On the other hand, object-relational mapping systems (ORM) [6], [7] are a possible way to solve the above problem. They bridge the gap between data models of relational storage and object-oriented code [8], [9]. Besides this basic functionality, they also establish a thick abstraction layer that can be augmented with abundant features. In our research, we have prepared proof-of-concept extensions to Hibernate that realize recursive queries [10], [11], [12], partial aggregation [13] and functional indices [14]. In particular, we experimented with adding recursion on top of database systems that do not implement it directly [15]. In order to accelerate processing recursive queries in such a setting we proposed adding redundant data.

In this paper we describe another format of redundant data called *logarithmic paths*. Its advantage lays in its linearithmic size, while most state-of-the-art methods possibly lead to squared space complexity. We also describe our proof-of-concept implementation of this new method and three known techniques to build redundant data that facilitate recursive querying. They are *nested sets*, *materialized paths* and *full paths*. We show results of extensive performance experiments to verify the quality of these solutions. They have shown that there is no dominating method. All of them have advantages and disadvantages. We summarize them and present recommendations when each of them seems to be the most suitable.

The contributions of this paper are as follows:

- a novel (linearithmic in space) method to build redundant data that accelerate recursive querying,
- a proof-of-concept implementation of this method in Hibernate assisted with the implementation of three state-of-the-art methods,

This work was supported by the Polish National Science Centre grants 2011/01/B/ST6/03867

978-1-4673-4471-5/$25.00

```
> SELECT * FROM emp;

 eid |   fname   |   sname   | bid
-----+-----------+-----------+-----
   1 | John      | Travolta  |
   2 | Bruce     | Willis    |   1
   3 | Marilyn   | Monroe    |
   4 | Angelina  | Jolie     |   3
   5 | Brad      | Pitt      |   4
   6 | Hugh      | Grant     |   4
   7 | Colin     | Firth     |   3
   8 | Keira     | Knightley |   6
   9 | Sean      | Connery   |   1
  10 | Pierce    | Brosnan   |   3
  ...
```

Fig. 1. Example persistent data on the hierarchy of employees in a company.

- a thorough experimental evaluation of the performance of these four methods,
- an analysis of their quality and circumstances under which each of them is recommended.

The paper is organized as follows. In Section II we address the related work. Section III describes the integration of the proposed method with Hibernate object-relational mapping system. In Section IV we present the new materialization method that uses only linearithmic space. Section V reports the results of an experimental evaluation of four methods to build redundant data for recursive queries. Section VI contains recommendations when each of the considered methods is most suitable. Section VII concludes.

II. Related work

Recursive relationships between entities can be implemented with an additional database table or by a single foreign key in case of hierarchies (many-to-one association). If nodes and edges are stored in the same table, querying such data can be more efficient. There are numerous optimisation methods for recursive queries [3], [4], [5]. The survey [2] summarizes implementations of recursive queries in commercial and open-source database management systems.

As noted above, a single table with self-referencing foreign key is the most straightforward way to store hierarchical data. In all sections of this paper we use a hierarchy of employees in a company as the running example. The number of levels of the hierarchy is not limited. Therefore, there exists no number n such that all leaves of the hierarchy are no further than n hops from the root. Figure 1 contains data on an example hierarchy recorded in the table `emp`. Figure 2 shows the schema of this table. The standard SQL:1999 query that retrieves all records from the subtree spanned by a particular record is presented on Figure 3.

A. Unrolling

There are database management systems that do not execute recursive queries with MySQL as the most famous example.

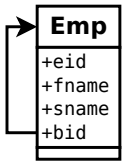

Fig. 2. The schema of the table `Emp`.

```
WITH RECURSIVE rcte (
  SELECT eid, fname, sname, bid,
         0 as level
    FROM Emp  WHERE sname = 'Travolta'
 UNION
  SELECT e.eid, e.fname, e.sname, bid,
         level +1  as level
    FROM Emp e JOIN rcte r
      ON (e.bid = r.eid)
    WHERE r.level<4
)
SELECT e.eid, e.fname, e.sname, bid
  FROM rcte
```

Fig. 3. This query retrieves all subordinates of `Travolta`.

The wide adoption of the LAMP paradigm of web development make us convinced that numerous web enterprises have to write extra code that queries recursive data. Most of such projects have similar data, e.g. nested categories of stock items, posts in discussion forums or revenue sharing chains in multi-level marketing. In order to address such applications, we have created a number of methods to run queries to such data even against database management systems that lack this feature [15]. Since the resulting solutions are non-trivial, we have prepared appropriate extensions to Hibernate. Our intent has been to hide the details from applications programmers and offer them a uniform API regardless of the chosen backend storage. Both methods considered in [15] have been integrated with API as presented on Figure 6. The first method, called *horizontal unrolling*, joins the subject table `maxlevel` times. Figure 4 shows the horizontal unrolling up to the 3$^{\text{rd}}$ level.

```
SELECT *
  FROM Emp l0
    LEFT JOIN Emp l1
      ON (l0.eid = l1.bid)
    LEFT JOIN Emp l2
      ON (l1.eid = l2.bid)
    LEFT JOIN Emp l3
      ON (l2.eid = l3.bid)
  WHERE l0.sname = 'Travolta'
```

Fig. 4. The horizontal unrolling of the query from Figure 3 up to the third level.

The other method, called *vertical unrolling*, uses temporary tables. It sends a number of queries and constructs the answer

from partial results. Both unrolling methods are notably more efficient than the potential naïve method that loops over nodes of the graphs and poses a separate query for each encountered node. Our experiments indicate that the horizontal variant is faster.

An application programmer/designer/architect chooses the required method of unrolling using the annotation `@unrolling`. Its parameter `method` can have two values: `"horizontal"` or `"vertical"`. The second (vertical) variant is the default since this method yields the same form of result as standard recursive queries. The result of horizontally unrolled query is slightly different.

B. Redundant data

If an application frequently queries hierarchical data, the abovementioned unrolling methods will not be efficient. However, in such cases designers can impose using redundant materialized data. As mentioned in Section I a number of such methods has been proposed [16]. Here we consider three of them. The first method called *nested sets* and the second method called *materialized paths* change the definition of base tables. They require adding a new column. The third method called *full paths* leaves the base table intact and puts materialized data into an additional table. In the following subsections we analyze their details.

1) Nested Sets : If the *nested sets* are used, two columns will be added to the base table. These are the columns `left` and `right`. The values of these columns satisfy the following constraints:

- $e.\texttt{left} < e.\texttt{right}$ for every tuple e,
- If a tuple e is in the subtree spanned by a tuple b, then it is true that $e.\texttt{left} > b.\texttt{left}$ and $e.\texttt{right} < b.\texttt{right}$.

Figure 5 shows the values of these two columns for the sample data from Figure 1. Arrows present how the values grow.

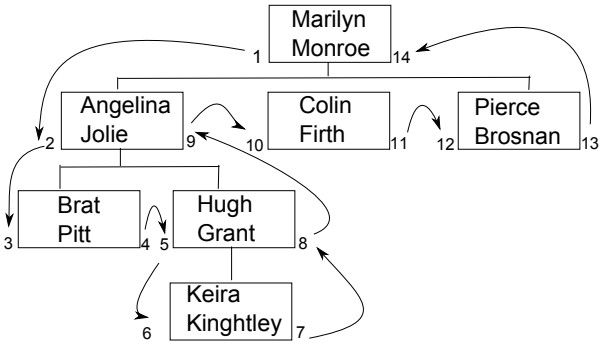

Fig. 5. Values of the redundant columns `left` and `right` in the method *nested set* computed for sample data.

The most significant advantage of this method is the possibility to query for descendants by means of a plan vanilla join. In order to find subordinates of `Travolta` we just execute the following query:

```
SELECT e.eid, e.name
  FROM emp e, emp b
  WHERE b.name = 'Travolta'
    AND e.left BETWEEN b.left AND b.right
    AND e.right BETWEEN b.left AND b.right
```

2) Materialized Paths: The method of *materialized paths* stores the whole path from the node to its root in the additional column `paths` as shown below:

```
 eid | fname |   sname    | bid | paths
-----+-------+------------+-----+------
   6 | Hugh  | Grant      |   3 | [4,3]
   7 | Colin | Firth      |   3 | [3]
   8 | Keira | Knightley  |   6 | [6,4,3]
```

This methods has proven to be noteworthy more universal that *nested sets* as discussed in Section V. Unfortunately, such materialization of paths breaks the first normal form. The following query enumerates all subordinates of `Travolta` when the method *materialized paths* is applied.

```
SELECT *
  FROM emp
 WHERE path_string LIKE (
     SELECT concat(path_string,'%')
       FROM emp
      WHERE sname = 'Travolta'
```

3) Full Paths : The method of *full paths* has been studied in [17]. That paper contains also a comparison of effectiveness of the full paths approach against unrolling methods. The full paths are similar to the *materialized paths*. However, the method of full paths stores redundant data in an extra table. We assume the name of this table to be `fullpaths`. It contains a distinct row for every step in any path towards the root. For `Keira Knightley` and `Colin Firth` the table *fullpaths* will contains the following rows.

```
 eid | bid | pl
-----+-----+----
   7 |   3 | 1
   8 |   6 | 1
   8 |   4 | 2
   8 |   3 | 3
```

The column `pl` (path length) contains the number of steps in the path.

This method is particularly universal. However, if the structure is deep, the size of the table *fullpath* can even be square with respect to the size of the base tables.

If the method of full paths is used, the query for `Travolta` subordinates will have the following form.

```
SELECT e.eid, e.name
  FROM emp e JOIN fullpaths fp USING (eid)
    JOIN emp b ON (fp.bid = b.eid)
  WHERE b.name = 'Travolta'
```

III. ORM CAN HIDE RECURSIVE PECULIARITIES

The layer of object-relational mapping [6], [7] facilitates cleaning the architecture and reducing the complexity of a system. However, an additional overhead between the application logic and the storage layers can hindrance the performance. In our research we assume that it is not the case. In our opinion an additional mapping layer *can* help optimizing the system. We can put there disparate algorithms and redundant data structures that aid improving the communication with the backend storage. The ORM layer also hides most of the peculiarities of such features from application programmers.

A. Hibernate interface for recursive queries

In our research we have presented the integration of recursive queries with object-relational mapping systems [10], [11], [12], [18]. In particular, our API for Hibernate allows defining recursive queries by XML annotations to Java entity classes [12]. Figure 6 shows a sample annotated entity class. For this class, ORM produces the table `Emp` presented in Figure 2.

```
package sample.recursive.mapping;
import org.ncu.hibernate.annotations.*;
@RecursiveQuery (maxLevel = 4)
@Tables (name = "Emp")
@RecursiveCondition (on = "Emp.bid",
                     to = "Emp.eid")
@Filter (seed = "Emp.sname = $Param(sname)")
public class Suboridnates {
    @Column(name = "Emp.eid")
    public String id;
    ...
}
```

Fig. 6. This annotation of an entity class causes generation of recursive facilities.

Consider the following scenario. The class from Figure 6 is registered in Hibernate. If the backend database connected to Hibernate implements recursive queries, the programmer can call the API function `getRecursive(String)`. This function sends the recursive query to the storage layer. If PostgreSQL is the backend, it will process the query from Figure 3.

B. Setting the support for recursive queries

If a programmer wants to use some of the above-mentioned methods to query recursive structures, e.g. unrolling or materialization, he/she just adds the annotation `@unrolling(method = "`*method name*`")`. The name of available methods are: `horizontal unrolling`, `vertical unrolling`, `full paths`, `nested sets`, `materialized paths` and `logarithmic paths`. The method of logarithmic paths is described in Section IV.

If no unrolling is specified and the database does not support recursive queries, the vertical unrolling will be used by default. More details of horizontal and vertical unrolling can be found in [15].

If a method based on materialization is chosen, the ORM layer will build the required redundant data at the first recursive access to the data. Depending on the method chosen, the main table will be altered (for nested sets or materialized paths) or an additional table will be created (for full paths and logarithmic paths). Then, redundant data get populated. Eventually, ORM automatically creates all necessary triggers. Thus, a programmer does nothing but chooses the method and specifies it in the `@unrolling` annotation.

IV. LOGARITHMIC PATHS

Section II-B presents three methods to build redundant materializations that facilitate querying recursive structures efficiently. In this Section we describe another such materialization method, called *logarithmic paths* or shortly *log paths*. This method is a kind of a compromise between full paths and vertical unrolling [15].

The idea of logarithmic paths is to store only those paths whose length is a power o 2. We assume that the data on such paths is stored in the redundant table `logpaths`. For `Keira Knightley` and `Colin Firth` the table *logpaths* will contains the following rows.

```
 eid | bid | pl
-----+-----+----
   7 |   3 | 1
   8 |   6 | 1
   8 |   4 | 2
```

If the data contains n tuples stored in trees of depth m, then the table `logpaths` will contain $O(n \log m)$ tuples. Therefore, we keep only $O(\log m)$ tuples for each arbitrary tuple of the base table, while *full paths* store $O(m)$ redundant tuples for each base tuple. For a user query the method of *log paths* issues $O(\log m)$ database queries for $O(1)$ columns, while the *horizontal unrolling* sends $O(1)$ queries for $O(m)$ columns.

A. Building the table `logpaths`

In order to populate the table `logpaths`, the paths of length 1 are copied from base table:

```
INSERT INTO logpaths
  SELECT eid, bid, 1 AS pl
    FROM emp
```

Next for $n = 1, 2, 4, 8, \ldots$ (powers of 2) the following query is executed as long as it adds new tuples. The population process will certainly finish since at each step strictly longer paths are created. Finite hierarchical data can contain only finite paths. In fact the number of those executions is $O(\log m)$ where m is the maximum depth of the hierarchy.

```
INSERT INTO powpath
  SELECT e.eid, b.bid, 2n AS pl
    FROM powpath e
      JOIN powpath b ON (e.bid = b.eid)
    WHERE e.pl = n and b.pl = n
```

B. Querying

At the query time, we have to reconstruct the information on all paths (as in *full paths*). We use the following auxiliary query:

```
SELECT lp1.eid, lpk.bid,
  lp1.pl + lp2.pl + ... + lpk.pl AS pl
  FROM logpaths lp1
    JOIN logpaths lp2
      ON (lp1.bid = lp2.eid)
    ...
    JOIN logpaths lpk
      ON (lp(k-1).bid = lpk.eid)
  WHERE lp1.pl < lp2.pl
    AND ...
    AND lp(k-1).pl < lpk.pl
```

This query reconstructs data on paths whose length (`p1`) has exactly k ones in its binary representation. Therefore, we have to run this query for $k = 1, 2, ..., \log m$ and merge their results using set union. Obviously, it will generate all paths and no path will be repeated. The resulting union query will be a part of the eventual user query. If it contains selections, the optimizer should push them down the query tree. Thus, with logarithmic paths only a fraction of the potential content of the table `fullpaths` will be actually computed.

C. Using ORM

This method has also been integrated into the Hibernate framework. If the annotation `@unrolling(method = "logarithmic paths")` is present, the table `logpath` will be automatically created and populated at the first recursive query. All necessary triggers are also created automatically.

V. Performace

The methods discussed in this paper has been tested on a computer with AMD Phenom II 3,4GHz, 8GB RAM and 2 Caviar Black 7400rpm 500 GB HDDs. The test has been run against Hibernate with a standard installation of MySQL as the backend database. We used six data sets of various sizes. Three of them contain 100 000 records organized in trees of depths 10, 15 and 20. The other three contain 1000000 records organized in trees with the same depths, i.e. 10, 15 and 20. We have tested seven usage scenarios.

The tables that report the results are organized as follows. Each table is divided into two parts. The first part presents results for data sets of 100 000 records, while the second part corresponds to data sets composed of 1 000 000 records. The first column of each table shows the depths of the trees. The second column presents times of evaluation for the presented methods. The names of two methods, namely *nested sets* and *materialized paths* are abbreviated to *ns* and *mp* respectively.

A. Building redundant materialization

In the first test we examine the time required to build redundant data structures that accelerate perspective recursive queries. We assume that the base table contains appropriate data and the derived table is empty. The results are presented in Table I.

TABLE I
Time necessary to build materialized data

100 000 rec				
	full path	logpath	ns	mp
10	00:00:47,94	00:00:15,90	00:00:29,41	00:00:12,59
15	00:01:26,20	00:00:17,67	00:00:29,26	00:00:13,23
20	00:02:54,21	00:00:21,82	00:00:29,55	00:00:13,54
1 000 000 rec				
	full path	logpath	ns	mp
10	00:22:13,21	00:04:53,27	00:34:59,03	00:05:16,97
15	01:00:24,07	00:05:30,94	00:39:47,73	00:05:10,29
20	02:03:50,40	00:06:24,73	00:39:25,69	00:05:31,39

B. Finding subordinates of root

We assume that we have an object representing a root in the hierarchy. We want to enumerate all nodes in the subtree below this root. The results are presented in Table II.

TABLE II
Time necessary to find subordinates of a root

100 000 rec				
	full path	logpath	ns	mp
10	00:00:01,57	00:00:07,99	00:00:00,51	00:00:00,56
15	00:00:01,91	00:00:08,90	00:00:00,51	00:00:00,60
20	00:00:03,64	00:00:05,27	00:00:00,51	00:00:00,64
1 000 000 rec				
	full path	logpath	ns	mp
10	00:00:30,16	00:04:32,97	00:00:10,27	00:00:09,70
15	00:00:40,09	00:07:08,23	00:00:10,96	00:00:10,09
20	00:00:49,72	00:07:44,32	00:00:10,94	00:00:10,08

C. Finding a subordinate of an arbitrary node

We assume that we have an arbitrary object in the hierarchy. We want to find an example node in the subtree below this node. The results are presented in Table III.

TABLE III
Time necessary to find subordinates of an arbitrary node

100 000 rec				
	full path	logpath	ns	mp
10	00:00:00,01	00:00:00,02	00:00:00,02	00:00:00,02
15	00:00:00,02	00:00:00,02	00:00:00,01	00:00:00,02
20	00:00:00,07	00:00:00,01	00:00:00,01	00:00:00,02
1 000 000 rec				
	full path	logpath	ns	mp
10	00:00:00,27	00:00:00,06	00:00:00,03	00:00:00,48
15	00:00:00,40	00:00:00,12	00:00:00,03	00:00:00,44
20	00:00:00,36	00:00:00,12	00:00:00,03	00:00:00,42

D. Finding the root for an arbitrary node

We assume that we have an arbitrary object in the hierarchy. We want to find the root of the tree that contains the given object. The results are presented in Table IV.

TABLE IV
TIME NECESSARY TO FIND THE ROOT FOR AN ARBITRARY NODE

100 000 rec				
	full path	logpath	ns	mp
10	00:00:00,00	00:00:00,01	00:00:00,01	00:00:00,01
15	00:00:00,00	00:00:00,05	00:00:00,01	00:00:00,01
20	00:00:00,01	00:00:00,05	00:00:00,01	00:00:00,01
1 000 000 rec				
	full path	logpath	ns	mp
10	00:00:00,01	00:00:00,01	00:00:00,01	00:00:00,01
15	00:00:00,01	00:00:00,07	00:00:00,01	00:00:00,01
20	00:00:00,01	00:00:00,08	00:00:00,01	00:00:00,01

TABLE VII
TIME NECESSARY TO MOVE A SUBTREE

100 000 rec				
	full path	logpath	ns	mp
10	00:00:08,53	0:00:26,60	00:00:19,28	00:00:00,61
15	00:00:16,01	00:00:51,14	00:00:08,70	00:00:00,64
20	00:00:52,75	00:01:28,36	00:00:24,15	00:00:01,19
1 000 000 rec				
	full path	logpath	ns	mp
10	00:04:30,89	00:13:37,25	00:00:22,51	00:00:00,47
15	00:07:49,17	00:35:06,68	00:01:20,40	00:00:00,08
20	00:13:38,99	00:57:39,43	00:00:28,07	00:00:00,92

E. Inserting new nodes

The next three tests are devoted to the assessment of the overhead imposed by all four methods when updates of the structure occur. Table V presents efficiency measures for the operation of inserting new rows into an existing base table. In this test we only added leaves to the hierarchy.

TABLE V
TIME NECESSARY TO INSERT NEW NODES TO THE HIERARCHY

100 000 rec				
	full path	logpath	ns	mp
10	00:00:00,14	00:00:00,08	00:00:05,22	00:00:00,06
15	00:00:00,20	00:00:00,09	00:00:04,35	00:00:00,06
20	00:00:00,20	00:00:00,08	00:00:16,54	00:00:00,05
1 000 000 rec				
	full path	logpath	ns	mp
10	00:00:00,15	00:00:00,11	00:07:47,67	00:00:00,05
15	00:00:00,17	00:00:00,12	00:08:27,64	00:00:00,05
20	00:00:00,16	00:00:00,12	00:08:16,85	00:00:00,05

F. Deleting nodes

In this test we examine the time needed to complete the delete operation. As in the previous test, we deleted leaves only. Deleting rows from the structure. The results are presented in Table VI.

TABLE VI
TIME NECESSARY TO DELETE A NODE FROM THE HIERARCHY

100 000 rec				
	full path	logpath	ns	mp
10	00:00:00,05	00:00:00,08	00:00:05,57	00:00:00,06
15	00:00:00,05	00:00:00,08	00:00:06,67	00:00:00,05
20	00:00:00,04	00:00:00,09	00:00:16,92	00:00:00,04
1 000 000 rec				
	full path	logpath	ns	mp
10	00:00:00,04	00:00:00,07	00:09:53,84	00:00:00,05
15	00:00:00,04	00:00:00,08	00:09:37,94	00:00:00,04
20	00:00:00,04	00:00:00,09	00:08:59,66	00:00:00,04

G. Updating nodes

In this test we measure the time needed to move a subtree to some other place. The results are presented in Table VII.

VI. ANALYSIS

In this Section we analyze the results of performance tests presented in Section V. We also formulate recommendations for which usage scenarios each of the analyzed methods is the most suitable. We presented four methods to build redundant data for efficient recursive queries. We divide them into two groups of two methods each. The first group contains methods that store data in additional tables, i.e. *full paths* and *logarithmic paths*. Since they do not require altering the original database schema, it will be easier to introduce them into an existing installation. The size of redundant data for *full paths* is $O(nm)$ where n the size of the data and m is the maximum depth of the hierarchy. In case of *logarithmic paths* this size is only $O(n \log m)$. Thus the method is feasible also in case of deep trees, i.e. when $m = O(n)$.

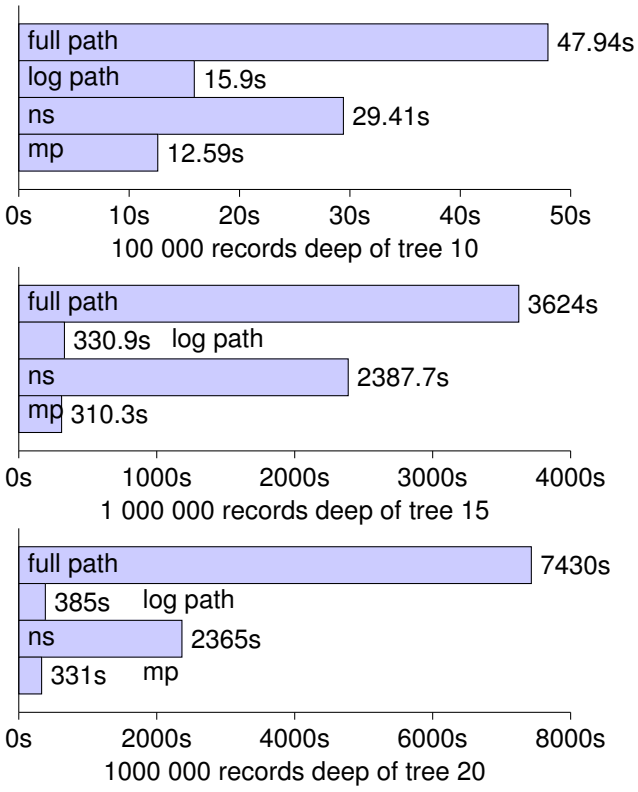

Fig. 7. The comparison of times necessary to build materialized data.

A significant advantage of *logaritmic paths* is the time necessary to construct the redundant data structure. For each analyzed size of data this method has proven to be fast and

comparable only to *materilized paths*. Figure 7 shows this comparison. Unfortunately, the efficiency of queries with this methods is notably lower. It is caused by multiple equijoins that are required to assemble all paths. For depth 20, *logarithmic paths* execute four subqueries combined by the set union. The most complex component of this union is a 4-way self join of the table `logpath`. The tests have shown that if the depth of the hierarchy grows, so does the execution time of queries. Figure 8 presents the comparison of times necessary to run the query that retrieves subtrees.

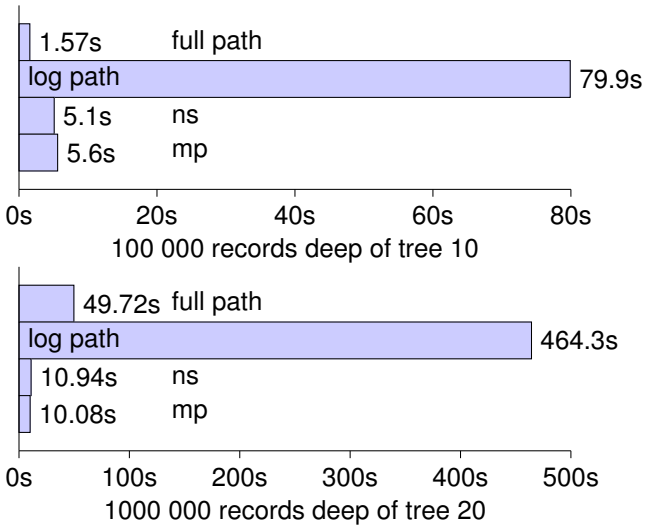

Fig. 8. The comparison of times necessary to retrieve whole subtrees.

On the other hand, *full paths* are useful when the data is collected incrementally as presented on Figure 9. Both creation and synchronization of redundant data is reasonably fast. However, in presence of deletes and updates that thoroughly *change* the structure of the tree, *full paths* are too inert and such operations are exceptionally costly.

The second group of methods modifies the base table by adding extra columns. Such an intervention into the schema may be too severe in existing deployments and probably will never be accepted in such circumstances. This group of methods contains *nested sets* and *materialized paths*. The first of them is optimized for retrieving whole subtrees. It requires careful allocation of identifiers. Furthermore, it is definitely the most expensive for maintenance in case of updates. The operations that modify the structure are two orders of magnitude slower than in case of other methods. On the other hand, *nested sets* are extremely fast for queries that search subtrees.

However, also for such scenarios *materialized paths* are slightly faster. This method is usually more efficient than all other methods. Unfortunately, it is also the only one that uses *semistructured columns*, i.e. non first normal form. If we accept this drawback, we will get the fastest querying method. Its disadvantages show at updates changing the logical structure of the tree.

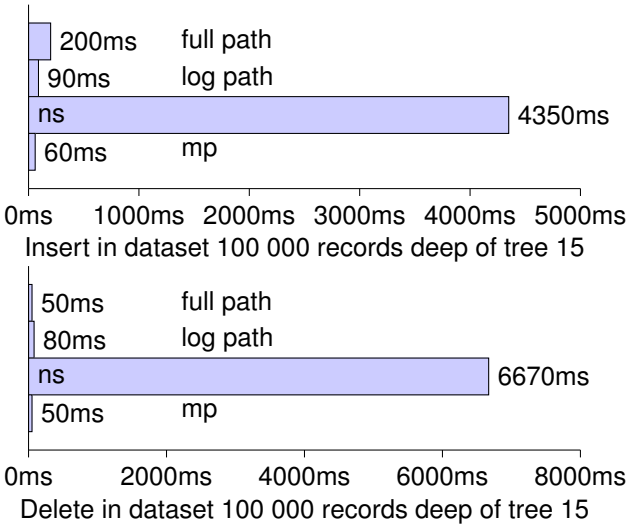

Fig. 9. The comparison of times necessary to insert and delete leaves.

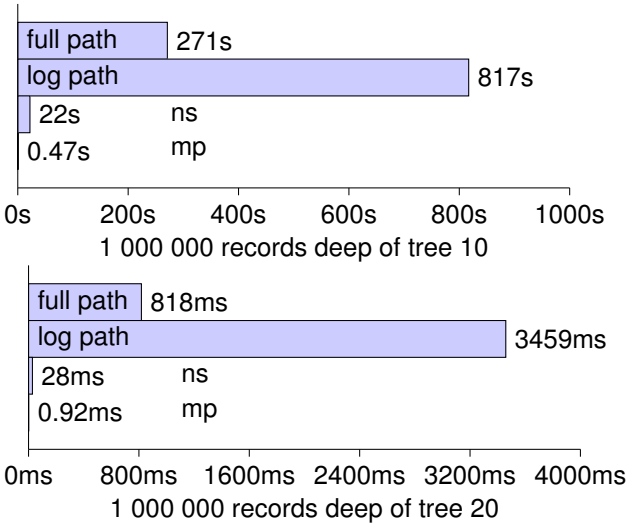

Fig. 10. The comparison of times necessary to move subtrees.

VII. Conclusions

In this paper we discussed four materialized data structures that accelerate recursive queries to hierarchical data. One of them called logarithmic paths is an original contribution of this paper. Logarithmic paths is the place where other methods meet halfway. Its efficiency is worse by a logarithmic factor than other methods that consume square space. However, logarithmic paths consume only linearithmic space.

We reported the results of experimental evaluation of all the four methods. None of them proved to be always worse or better than another tested method. For each of them, there are scenarios where it is the recommended materialization. We summarized our advises when to use each of the methods.

All these methods have been prototypically implemented as part of Hibernate, i.e. the most popular Java object-relational mapping system. This allows (1) hiding all the peculiarities of

these solutions from application programmers and (2) offering architects and tuners an easy choice of the materialization that is the most suitable for the application at hand.

REFERENCES

[1] D. Brandon, "Recursive database structures," *J. Comput. Sci. Coll.*, vol. 21, no. 2, pp. 295–304, Dec. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1089053.1089098

[2] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel, "Recursive query facilities in relational databases: A survey," in *FGIT-DTA/BSBT*, ser. Communications in Computer and Information Science, Y. Zhang, A. Cuzzocrea, J. Ma, K.-I. Chung, T. Arslan, and X. Song, Eds., vol. 118. Springer, 2010, pp. 89–99.

[3] A. Ghazal, A. Crolotte, and D. Y. Seid, "Recursive sql query optimization with k-iteration lookahead," in *DEXA*, ser. Lecture Notes in Computer Science, S. Bressan, J. Küng, and R. Wagner, Eds., vol. 4080. Springer, 2006, pp. 348–357.

[4] C. Ordonez, "Optimization of linear recursive queries in sql," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 2, pp. 264–277, 2010.

[5] M. Burzańska, K. Stencel, and P. Wiśniewski, "Pushing predicates into recursive sql common table expressions," in *ADBIS*, ser. Lecture Notes in Computer Science, J. Grundspenkis, T. Morzy, and G. Vossen, Eds., vol. 5739. Springer, 2009, pp. 194–205.

[6] S. Melnik, A. Adya, and P. A. Bernstein, "Compiling mappings to bridge applications and databases," *ACM Trans. Database Syst.*, vol. 33, no. 4, 2008.

[7] W. Keller, "Mapping objects to tables. a pattern language," in *EuroPLoP*, 1997, pp. 1–26.

[8] E. J. O'Neil, "Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM)," in *SIGMOD Conference*, J. T.-L. Wang, Ed. ACM, 2008, pp. 1351–1356.

[9] C. Bauer and G. King, *Java Persistence with Hibernate*. Greenwich, CT, USA: Manning Publications Co., 2006.

[10] M. Burzańska, K. Stencel, P. Suchomska, A. Szumowska, and P. Wiśniewski, "Recursive queries using object relational mapping," in *FGIT*, ser. Lecture Notes in Computer Science, T.-H. Kim, Y.-H. Lee, B. H. Kang, and D. Ślęzak, Eds., vol. 6485. Springer, 2010, pp. 42–50.

[11] P. Wiśniewski, A. Szumowska, M. Burzańska, and A. Boniewicz, "Hibernate the recursive queries - defining the recursive queries using Hibernate ORM," in *ADBIS (2)*, ser. CEUR Workshop Proceedings, J. Eder, M. Bieliková, and A. M. Tjoa, Eds., vol. 789. CEUR-WS.org, 2011, pp. 190–199.

[12] A. Szumowska, M. Burzańska, P. Wiśniewski, and K. Stencel, "Efficient implementation of recursive queries in major object relational mapping systems," in *FGIT*, ser. Lecture Notes in Computer Science, T.-H. Kim, H. Adeli, D. Slezak, F. E. Sandnes, X. Song, K.-I. Chung, and K. P. Arnett, Eds., vol. 7105. Springer, 2011, pp. 78–89.

[13] M. Gawarkiewicz and P. Wiśniewski, "Partial aggregation using Hibernate," in *FGIT*, ser. Lecture Notes in Computer Science, T.-H. Kim, H. Adeli, D. Slezak, F. E. Sandnes, X. Song, K.-I. Chung, and K. P. Arnett, Eds., vol. 7105. Springer, 2011, pp. 90–99.

[14] A. Boniewicz, M. Gawarkiewicz, and P. Wiśniewski, "Automatic selection of functional indexes for object relational mappings system," *accepted to International Journal of Software Engineering and Its Applications*, vol. 7, 2013.

[15] A. Boniewicz, K. Stencel, and P. Wiśniewski, "Unrolling SQL:1999 recursive queries," in *Computer Applications for Database, Education, and Ubiquitous Computing*, ser. Communications in Computer and Information Science, T.-h. Kim, J. Ma, W.-c. Fang, Y. Zhang, and A. Cuzzocrea, Eds. Springer Berlin Heidelberg, 2012, vol. 352, pp. 345–354.

[16] J. Celko, *Joe Celko's Trees and hierarchies in SQL for smarties, second edition*. Elsevier/Morgan Kaufmann, 2012.

[17] A. Boniewicz, P. Wiśniewski, and K. Stencel, "On materializing paths for faster recursive querying," in *ADBIS*, 2013, p. to appear.

[18] A. Szumowska, M. Burzańska, P. Wiśniewski, and K. Stencel, "Extending HQL with plain recursive facilities," in *ADBIS (2)*, ser. Advances in Intelligent Systems and Computing, T. Morzy, T. Härder, and R. Wrembel, Eds., vol. 186. Springer, 2012, pp. 265–272.

[19] T.-H. Kim, H. Adeli, D. Slezak, F. E. Sandnes, X. Song, K.-I. Chung, and K. P. Arnett, Eds., *Future Generation Information Technology - Third International Conference, FGIT 2011 in Conjunction with GDC 2011, Jeju Island, Korea, December 8-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7105. Springer, 2011.