TWO ALGORITHMS FOR LOCATING ANCESTORS OF A LARGE SET OF VERTICES IN A TREE

Oleksandr Panchenko, Arian Treffer, Hasso Plattner and Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering, P.O. Box 900460, 14440 Potsdam, Germany

Keywords: Query processing, Tree processing, XML database, Data storage, Algorithms.

Abstract: A lot of tree-shaped data exists: XML documents, abstract syntax trees, hierarchies, etc. To accelerate query processing on trees stored in a relational database a pre-post-ordering can be used. It works well for locating ancestors of a single or few vertices because pre-post-ordering avoids recursive table access. However, it is slow if it comes to locating ancestors of hundreds or thousands of vertices because ancestors of each of the input vertices are located sequentially. In this paper, two novel algorithms (*sort-tilt-scan* and *single-pass-scan*) for solving this problem are proposed and compared with a naïve approach. While the *sort-tilt-scan* improves the performance by a constant factor, the *single-pass-scan* achieves a better complexity class. The performance gain is achieved because of a single table scan which can locate all result vertices by a single run. Using generated data, this paper demonstrates that the *single-pass-scan* is orders of magnitude faster than the naïve approach.

1 INTRODUCTION

A lot of tree-shaped data exists: XML documents, abstract syntax trees, bills of material, hierarchies, etc. In many cases leaves of a tree contain the actual information and all other vertices (their ancestors) describe the meaning of the leaves and their dependencies. Usually, users request information stored in the leaves, but a (database) system requires all corresponding ancestors to reconstruct the context of the query or to visualize the result set properly. Therefore, the result set often should contain the matching vertices *and* their ancestors.

To store a tree in a relational database, each vertex is stored as a tuple with a unique ID. To represent the tree structure, an attribute for the parent ID is required. With this crude data schema, it is possible to fetch immediate children or parent of a vertex. To locate *all* ancestors or descendants of a vertex efficiently, pre-post-order numbering can be used (Grust et al., 2004). For creating pre-post-order labels, the entire tree has to be traversed once. For each vertex, on entering and leaving the pre- and post-order are assigned as consecutive numbers. Figure 1 illustrates the labeling process with an example.

For a given vertex, its ancestors are all vertices that were entered before and left after visiting this vertex. Thus, they all have a smaller pre-order number and a larger post-order number. Now it is possible to fetch *all* ancestors or descendants of a vertex with one table scan. For the example given in Figure 1, the following SQL query could be used to find the ancestors of the vertex H:

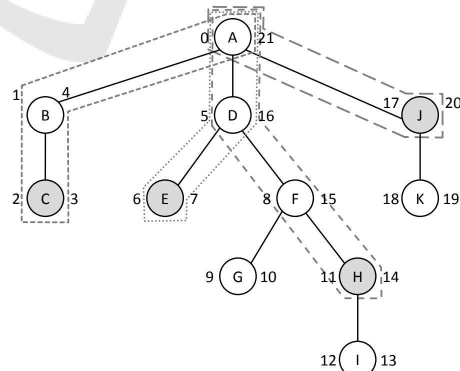

Figure 1: A set of matches. The result paths are highlighted.

```
SELECT * FROM vertices
WHERE preOrder < 11 AND postOrder > 14
```

If the comparison operators are changed to opposite, the same query can be used to compute the descendants of the vertex.

However, it is slow when it comes to locating ancestors of hundreds or thousands of vertices simultaneously because ancestors of each of the input ver-

Panchenko O., Treffer A., Plattner H. and Zeier A..
TWO ALGORITHMS FOR LOCATING ANCESTORS OF A LARGE SET OF VERTICES IN A TREE.
DOI: 10.5220/0003599202800285
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 280-285
ISBN: 978-989-8425-76-8
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

tices are located sequentially (see Section 2.1 for details). In this paper, two algorithms for solving this problem are proposed and compared with a naïve approach. While the *sort-tilt-scan* improves the performance by a constant factor, the *single-pass-scan* achieves a better complexity class. The performance gain is achieved because of a single table scan which can qualify all result vertices by a single run. Using generated data, this paper shows that the *single-pass-scan* is orders of magnitude faster than the naïve approach.

Pre-post-order numbering is not the only labeling scheme that was proposed to accelerate XPath queries. For instance, a scheme for improving sequences of child steps has been proposed (Chen et al., 2004). Since the problem addressed in this paper occurs after the evaluation of the query, the evaluation of XPath queries itself is out of scope and is not discussed further. Implementations of XPath engines are available as open source projects (i.e., Apache Xalan) and are subject to current research (Gou and Chirkova, 2007; Peng and Chawathe, 2005). Conceptually, the paper proposes applying an algorithm of a streaming nature to the data that is stored in a database. We show that such a combination is efficient for our scenario as we have a rather simple query but a large input data set.

Streaming processing of XML data (Li and Agrawal, 2005) is capable of handling large amounts of data. Although a number of algorithms exist for locating patterns on a tree, as far as we know no algorithm exists that is specially designed for the large number of input vertices as we have them.

Our algorithms assume that the data is rarely changed. This assumption allows us to use pre-post-ordering and to physically sort the data as described below. This condition significantly restricts the number of scenarios in which the algorithms can be applied. Nevertheless, there are still a number of cases that do not change data often. For example, an abstract syntax tree, if changed, the entire tree could be replaced because it should be parsed anyway (Panchenko et al., 2010). In this case the vertices of the new (or updated) tree will be placed at the end of the table. The new pre-orders and post-orders will be assigned to preserve the physical order. The old data will then be removed from the table.

The paper is organized as follows. Section 2 discusses a naïve approach to the problem and introduces the two novel algorithms. In Section 3 the algorithms are evaluated. The last section summarizes the results and discusses future work.

2 ALGORITHMS

Once the XPath[1] query was successfully executed, the paths to the result vertices have to be computed. Figure 1 shows an example that will be used throughout this paper to explain the proposed algorithms.

2.1 Naïve Approach

A frugal way of solving the problem is appending /ancestor-or-self::* to each query. However, with a certain database size, this approach is not feasible since the computation would take too long. For the example of Figure 1, the ancestors of the result [C, E, H, J] could be computed with this SQL query:

```
SELECT * FROM vertices
WHERE (preOrder  <  2 AND postOrder >  3)
                                /* Vertex C */
   OR (preOrder  <  6 AND postOrder >  7)
                                /* Vertex E */
   OR (preOrder  < 11 AND postOrder > 14)
                                /* Vertex H */
   OR (preOrder  < 17 AND postOrder > 20)
                                /* Vertex J */
```

It is clear that for each additional result vertex another condition has to be added to the WHERE clause. Since low selective queries in large systems can produce thousands of hits, this approach quickly exceeds the maximum query size of most database systems. But even a manual implementation would not solve the fundamental problem, namely that it is necessary to iterate over the entire vertex set and that for each vertex it has to be checked whether it is an ancestor of one of the result vertices. This leads to a runtime complexity of $O(|vertices| * |input|)$. Thus, assuming that for a given query the size of the result is proportional to the size of the database, the naïve approach has a quadratic complexity. The naïve algorithm is depict in the following listing.

Algorithm 2.1: NAÏVE APPROACH *(input,vertices)*.

```
comment: evaluate each input vertice
for each c ∈ input
  do { comment: iterate over each vertex in the table
       for each v ∈ vertices
         do { comment: check if the vertex qualifies as ancestor
              comment: of one of the input vertices
              if v.pre < c.pre
               and v.post > c.post
                then { comment: add to the result set
                       output (v)
```

[1]We mentioned XPath here, but other query languages can be used as well.

2.2 Sort-tilt-scan Algorithm

Since in general the entire tree is significantly bigger than the result set, it is better to iterate over the entire tree in the outer loop and use the inner loop to probe the result set. Even if the entire database is kept in memory, switching the loops slows down the execution. However, this loss of performance may be justified if the overall efficiency of the algorithm can be improved. With a sorted index over the pre-order attribute it is possible that once a vertex itself has been reached to stop searching for ancestors of this vertex. Furthermore, for vertices in the rear part of the tree, the same can be done even faster with an index that is sorted by post-order. Figure 2 shows how this approach would process the example tree. For each vertex it has to be decided first in which table it has a higher position by using the function usePreOrderSorting(c)). This can be done in constant time if the pre-post-order numbering scheme is adjusted. When using separate counters for pre- and post-order, the positions in the sorted indices can be derived directly from the order numbers. For consistency with the other figures, this modification is not shown in the example. After the index was selected, the algorithm searches for ancestors until the result vertex is reached. Because of the sorting, no ancestors will be found after that point. This algorithm has the same complexity as the naïve approach, but improves execution time with a constant factor. We also tried to further improve the performance by calculating the min pre-order and max post-order. Starting the table scans with min pre-order and max post-order should reduce the number of tuples to be scanned. However, in practice this optimization negligibly improves performance because of the distribution of the pre- and post-orders. The *sort-tilt-scan* algorithm is depict in the following listing.

Algorithm 2.2: SORT-TILT-SCAN *(in, preOrdered, postOrdered).*

comment: evaluate each input vertex

for each $c \in in$

do
- **comment:** decide which sorting to use
- **if** $usePreOrderSorting(c)$
  - **then**
    - **for each** $v \in preOrdered$, $(v.pre < c.pre)$
    - **do**
      - **if** $v.post > c.post$
      - **then**
        - **comment:** v=ancestor(c)
        - **output** (v)
  - **else**
    - **for each** $v \in postOrdered$, $(v.post > c.post)$
    - **do**
      - **if** $v.pre < c.pre$
      - **then**
        - **comment:** v=ancestor(c)
        - **output** (v)

Node	pre	post
A	0	21
B	1	4
C	2	3
D	5	16
E	6	7
F	8	15
G	9	10
H	11	14
I	12	13
J	17	20
K	18	19

Node	pre	post
A	0	21
J	17	20
K	18	19
D	5	16
F	8	15
H	11	14
I	12	13
G	9	10
E	6	7
B	1	4
C	2	3

Figure 2: The vertex set, once sorted ascending by pre-order, once sorted descending by post-order. Result vertices and their ancestors are circled. The arrows indicate rows scanned by *sort-tilt-scan*.

2.3 Single-pass-scan Algorithm

As it can be seen in Figure 2, some vertices are checked multiple times. This is true insofar as these vertices are ascendants of more than one result vertex. However, for the final result it is not important to find these vertices more than once. This trait can be exploited if the table and the result vertices are sorted by pre-order. Once all ancestors of a result vertex V_1 have been found, some ancestors of the next result vertex V_2 are found as well. As it can be seen in Figure 3, all ancestors of V_2 that are not ancestors of V_1 must have a higher pre-order than N_1. Thus, for computing the ancestors of a result vertex V_i, in a table sorted by pre-order only the vertices between V_{i-1} and V_i have to be checked. This algorithm assumes that the input vertices are sorted by pre-order. Figure 4 shows how the algorithm scans the table.

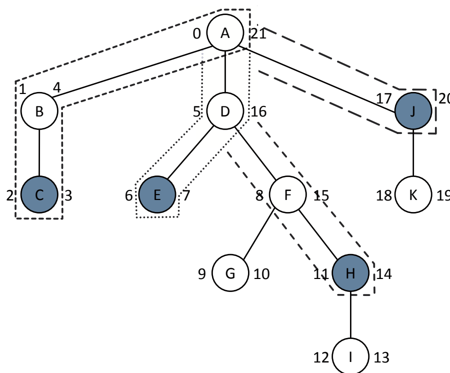

Figure 3: Ancestors of result vertices without overlapping.

The removed overlapping of the result paths is shown in Figure 3. As it can be seen in Figure 4, the algorithm iterates over the vertex set only once. In the worst case scenario, when at least one result vertex is at the very end of the table, the execution

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Node	pre	post
A	0	21
B	1	4
C	2	3
D	5	16
E	6	7
F	8	15
G	9	10
H	11	14
I	12	13
J	17	20
K	18	19

Figure 4: The vertex set is sorted by pre-order. Result vertices and their ancestors are highlighted. The arrows show which rows are scanned by *single-pass-scan*.

time of the algorithm only depends on the size of the vertex set, and is independent of the number of result vertices. Thus, especially for queries with low selectivity, a good execution time is expected. After the *single-pass-scan* is done, each of the output vertices has to be mapped to corresponding ancestors. This can be done using the naïve approach or *sort-tilt-scan*. This additional step negligibly affect the performance because the number of output vertices is significantly smaller than the size of the entire table. The algorithm is depict in the following listing.

Algorithm 2.3: SINGLE-PASS-SCAN *(input, preOrdered)*.

```
comment: evaluate each input vertex
for each c ∈ input
do {
    comment: continue with last v!
    continue for each v ∈ preOrdered
    do {
        if v.preOrder > c.preOrder
          then {
            comment: no ancestors of c after this
            break
          }
        else if v.postOrder > c.postOrder
          then {
            comment: v is an ancestor of c
            output (v)
          }
    }
}
```

3 EVALUATION

When writing algorithms for large datasets, it is important that they operate as closely as possible on the original data. Furthermore, for the purpose of comparing the algorithms, the execution should not be affected by database specific optimizations. Therefore, a simple prototype was developed in Java for the evaluation. The prototype uses a column store that is implemented with int-arrays. On startup, the generated data is loaded from a file, sorted by pre-order. Additionally, a sorted index is created on the post-order attribute.

Each algorithm accesses the data with an iterator that hides the column architecture. This provides data access in constant time and ensures the benefits of a column store, such as sequential reading. As a result of the computation, the indices of the selected vertices are stored in an array list, which provides amortized constant time for inserts. The generated data consisted of 2.000 equally shaped trees that were combined to one large super tree. Each tree had a depth of seven with four children per vertex. In total, the data set contained 10.922.000 vertices. Since the location of the ancestors is independent of the evaluation of the query[2], the set of vertices was generated of which the ancestors will be computed. Although a number of standard XML benchmark exists, we decided to generate data in such a way that we can show the properties of the algorithm in the best way as standard benchmarks do not have such a query. We performed our tests on a Linux machine equipped with Intel®Xeon®CPU E5450 takted at 3GHz.

The execution time of the algorithms mainly depends on the size of the database and the selectivity of the original query. Figure 5 presents dependency of query execution time from the query selectivity. The selected vertices are distributed randomly in the super tree. Only leaf vertices were selected. Each data point shows the average of five runs with a different random seed. As expected, the execution time of the naïve approach and the *sort-tilt-scan* grows linear with the selectivity, while the execution time of the *single-pass-scan* remains almost constant. A slight increase can be observed because of the additional overhead of adding more vertices to the ascendants list.

We also compared our implementation with MySQL and MonetDB. Although it was not a primary goal of our research, we added these measurements to illustrate that convenient database systems behave similarly to our naïve implementation.

In this test, the vertices were distributed randomly throughout the entire tree. However, when searching for certain vertices, it is likely that most matches concentrate on a certain area of the table. To represent this trait, the ordered vertex set was split into 200 equal subsets. Then, for each of 11 tests, all vertices were selected from only one of these subsets. Figure 6 shows the result of these tests. The selectivity of the input set of vertices was chosen to be 0.0007%. The naïve approach exhibits almost constant execution time because the entire table has to be scanned

[2]Some algorithms for streaming evaluation of XPath are capable of locating ancestors during the evaluation of the query. However, in this paper we focus on those algorithms which are not.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

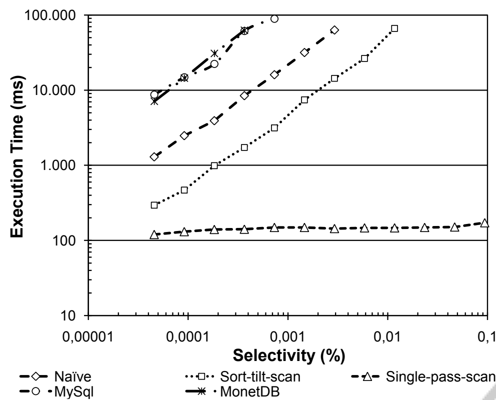

Figure 5: Query execution time of the three algorithms, MySQL and MonetDB depending on the selectivity of the original query.

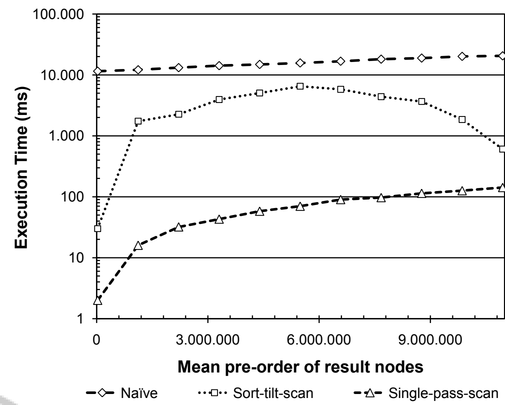

Figure 6: Query execution time, depending on the position of the selected vertices.

anyway. Since the last two algorithms start with iterating over the vertex set, beginning with the smallest pre-order, they are faster when all hits can be found early in the table. With increasing pre-order of the result vertices, the algorithms become linearly slower. As it was expected the *sort-tilt-scan* becomes faster again in the second half of the table, when the vertices ascend in the table that is sorted by post-order.

In the next test run, the execution time was measured for different database sizes. To simulate that all vertex sets were queried with the same query, the selectivity for creating the result set was 0.0007% for each run, which corresponds to 80 hits in the full database. The results of this test are shown in Figure 7. On one hand, one can see that the execution time of the naïve approach and of the *sort-tilt-scan* is growing quadratic, caused by the linear increase of database and result set sizes. On the other hand, the *single-pass-scan* shows a linear behavior as it was predicted in the analysis in Section 2.3. When comparing the overall performance of larger result sets, the *single-pass-scan* is orders of magnitude faster, while the *sort-tilt-scan* is faster than the naïve approach by a constant factor.

Since the *single-pass-scan* scans tuples of the table sequentially, it can exploit the advantages of modern processors if the table is stored using column-oriented layout. Furthermore, to reduce the amount of scanned data a simple dictionary compression can be used (Cockshott et al., 1998). Figure 8 illustrates the impact of the physical data layout on performance. In our prototype we changed the layout of the arrays as proposed by Li et al. (Li et al., 2004).

The performance advantages come at costs of additional space requirements because a sorted index is needed. Furthermore, the performance of data manipulation operations (insert, update, delete) is affected

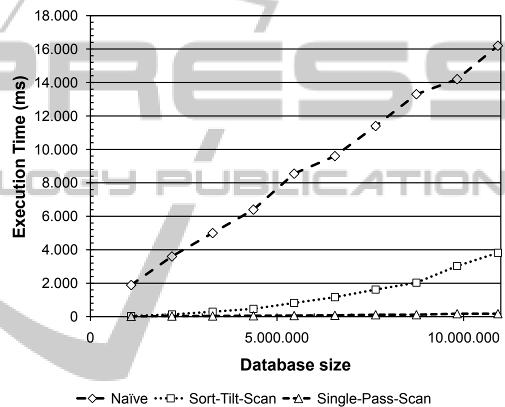

Figure 7: Query execution time for different database sizes with constant selectivity.

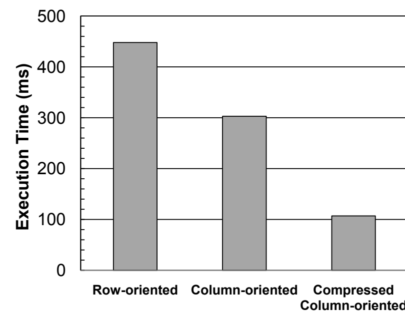

Figure 8: Query execution time of *single-pass-scan* for different data layouts. The selectivity is 0.1%.

because of the need of maintaining the index and of keeping the records sorted by pre-order.

4 CONCLUSIONS AND FUTURE WORK

This paper showed how the computation of the paths

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

from the root of the tree to the result vertices of an XPath query can be improved. Therefore two algorithms were proposed. The algorithms were designed for a relational column-oriented in-memory database and rely on pre-post-order numbering. The *sort-tilt-scan* improves the naïve approach by using two sorted tables and minimizing the number of rows that have to be scanned for a vertex to find its ancestors. The *single-pass-scan* exploits the fact that the result is a set and does not need the information how often a certain vertex was found as an ancestor. In this way it is possible to solve the problem with a single table scan. In the evaluation part it was found that the *single-pass-scan* greatly improves the performance of the task, especially for queries with a high selectivity. It was significantly faster than other algorithms, regardless of the database size, the queries selectivity or the distribution of the results.

Since the *single-pass-scan* is based on a table scan, it provides possibilities for parallelization. Further tests should show how the algorithms behave in a multithreaded environment. Another approach for fetching the paths to the result vertices would be using a streaming-based XPath engine as XSQ (Peng and Chawathe, 2005). It could be modified to keep a stack of ancestors and write them into the result once the query produced a hit. However, this might slow down the actual execution of the query, as the advantage of targeted, index-based access to the data could not be used. Finally, it should be mentioned that with a slight modification the algorithms could be used to fetch the descendants of a set of vertices as well. This way, a new potential for optimizing the implementation of the XPath axes arises. Thus, the algorithms proposed in this paper can not only improve the post-processing of the result, but the evaluation of the query as well.

REFERENCES

Chen, Y., Davidson, S. B., and Zheng, Y. (2004). BLAS: an efficient XPath processing system. In *Proceedings of the SIGMOD international conference on Management of data*, pages 47–58, New York, NY, USA. ACM.

Cockshott, W. P., McGregor, D., Kotsis, N., and Wilson, J. (1998). Data Compression in Database Systems. In *Proceedings of the International Workshop on Database and Expert Systems Applications*, page 981. IEEE Computer Society.

Gou, G. and Chirkova, R. (2007). Efficient Algorithms for Evaluating XPath over Streams. In *Proceedings of the SIGMOD international conference on Management of data*, pages 269–280, New York, NY, USA. ACM.

Grust, T., Keulen, M., and Teubner, J. (2004). Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 1.

Li, F., Agrawal, P., Eberhardt, G., Manavoglu, E., Ugurel, S., and Kandemir, M. (2004). Improving Memory Performance of Embedded Java Applications by Dynamic Layout Modifications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 159. IEEE Computer Society.

Li, X. and Agrawal, G. (2005). Efficient Evaluation of XQuery over Streaming Data. In *Proceedings of the 31st international conference on Very large data bases*, pages 265–276. VLDB Endowment.

Panchenko, O., Treffer, A., and Zeier, A. (2010). Towards Query Formulation and Visualization of Structural Search Results. In *Proceedings of the ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, pages 33–36, New York, NY, USA. ACM.

Peng, F. and Chawathe, S. S. (2005). XSQ: A Streaming XPath Engine. *ACM Transactions on Database Systems*, 30(2):577–623.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS