FAKULTÄT FÜR !NFORMATIK

Optimizations of structural join algorithms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

ausgeführt von

Markus Dorner
Matrikelnummer 0225927

am:
Institut für Informationssysteme
Arbeitsbereich für Datenbanken und Artificial Intelligence

Betreuung:
Betreuer/Betreuerin: Univ.-Prof. Dr. Reinhard Pichler

Wien, 21.07.2008

_____ (Unterschrift Verfasser/in)

_____ (Unterschrift Betreuer/in)

Abstract

Joins are a core operation in database systems. Nowadays every larger database system comes with XML support which allows XPath or XQuery statements to retrieve the stored data. Structural joins have to perform as good as possible to support efficient data retrieval.

Many different algorithms have been proposed which work with different strategies to process a structural join. Every algorithm has its strengths and weaknesses. The core of every algorithm is the numbering schema which allows determining a structural relationship between two nodes efficiently.

Earlier proposed algorithms like the TreeMerge and the StackTree only handle queries with two nodes. A longer query has to be split and its intermediate results merged together afterwards.

Later algorithms like the TwigStack or the PathStack process a query at once and use a special stack-encoding to store intermediate results efficiently. The usage of stacks has proven as very efficient. The StackTree was the first algorithm which used one stack to cache ancestor nodes.

State of the art is the Twig^{2}Stack which can also handle optional query nodes. It uses a much more complex stack-encoding combined with the numbering schema.

Even specific index structures have been proposed to speed up the search for ancestors or descendants of a node. Some structures like the B+ tree have been adapted from the relational databases to work with the numbering schema.

Other structures like the XR-tree were especially designed to find ancestors and descendants of a node very efficiently. New concepts like stab lists have been introduced in this structure.

This work will give an overview how the algorithms have improved and how the new strategies affect the join performance.

Kurzfassung

Join Operationen spielen bei Datenbanksystemen eine entscheidende Rolle. Sie gehören zu den wichtigsten Operationen und beeinflussen stark die Leistung des gesamten Systems. Nahezu jeder namhafte Datenbankhersteller liefert seine Produkte mit XML Unterstützung aus und ermöglicht so Datenbankabfragen mit XPath oder XQuery. Somit sind strukturelle Joins sehr wichtig, um eine effiziente Abfrage zu ermöglichen.

Im Laufe der Zeit wurden zahlreiche Algorithmen veröffentlicht, welche die verschiedensten Strategien haben, um möglichst effizient zu arbeiten. Die wichtigste Gemeinsamkeit ist die Nummerierung der Knoten. Dieser Index macht es möglich schnell und effizient die Beziehung zwischen zwei Knoten zu ermitteln.

Frühere Algorithmen wie der TreeMerge oder der StackTree können jeweils immer nur zwei Abfrageknoten verarbeiten. Dafür müssen längere Abfragen aufgespalten werden und die Teilergebnisse später zusammengefügt werden. Es ist klar, dass durch das Zusammenfügen die Performanz des Algorithmus leidet. Für Abfragen, die nur aus zwei Knoten bestehen, liefert der StackTree jedoch sehr gute Ergebnisse, da der Algorithmus selbst nur wenig Overhead erzeugt.

Neuere Algorithmen wie der PathStack oder der TwigStack verarbeiten eine Query als ganzes und sparen somit das Zusammenführen der Zwischenergebnisse ein. Natürlich haben diese Algorithmen mehr Overhead beim verwalten ihrer Datenstrukturen. Die zuvor erwähnten Algorithmen verwenden zum Beispiel eine spezielle Datenstruktur aus Stacks die Zwischenergebnisse optimal speichert um eine optimale Leistung zu erzielen.

Der Aktuellste ist der Twig^{2}Stack Algorithmus, welcher sogar optionale Abfrageknoten verarbeiten kann. Dieser hat eine noch komplexere Datenstruktur und ist somit eine Weiterentwicklung des PathStack.

Es wurden ebenfalls weitere Indexstrukturen verwendet, welche die Suche nach einem Vorgänger und Nachfolger einer Knoten beschleunigen. Dabei wurden bewährte Strukturen wie der B+-Baum angepasst aber auch neue Strukturen wie der XR-Baum entwickelt. Dabei wurden zum Beispiel neue Konzepte wie die Stab-lists eingeführt.

Diese Arbeit gibt einen Überblick wie die Algorithmen mit der Zeit verbessert wurden und wie sich neue Ideen und Strategien auf die Join Operation auswirken.

Index

1 Introduction

The XML format is well defined and wide spread. Due to its tree structure it is easy to build up hierarchical structured repositories and makes XML perfect to store and transmit data. Another advantage is the processing flexibility. The XML document can be read at once, which makes it possible to browse through the DOM structure. It can also be read from a stream and in this case each tree node will be processed when it is read. These advantages made XML standard for storing data and it is used for example in various different communication protocols.

The amount of data increases rapidly which causes that it is very important to access saved data efficiently. Many programmers shred [22] a received XML document to a relational database. With the relational database they assure that they can use all its advantages like joins and indexes to receive needed data quickly. Obviously the shredding process of the data has a big disadvantage. Every transformation causes overhead which uses system resources and decreases processing performance. Applications often need data in the original format to retransmit it. Then the data will be read out of the relational database and transformed to XML again. In cases where the data is needed in the original XML format the wrapping operation is very inefficient. Especially when a lot of requests have to be processed it can decrease system performance. Then it is better to store the data in its native XML format to avoid wrapping of the data. Techniques from relational databases have been adapted that they could be used for XML repositories. For example various index types which make it possible to find data much faster.

Structural joins are an XML database core operation. The information has to be retrieved efficiently and in a specified structure. An XML query also specifies the tree structure of the searched data and not only the node labels/values. Many different join algorithms have been proposed. Some algorithms are a direct optimization from recently proposed algorithms and other algorithms introduced new concepts. For example the Tree-Merge algorithms have a direct counterpart in the relational database systems and have been adapted to work with XML repositories.

In section 2 the numbering schema is presented which is very important to determine structural relationships and is used in every algorithm.

Section 3 shows the structural join algorithms which are using indexes like the B+ tree or the R-tree. The Stack-Tree compared with the Tree-Merge algorithms avoids multiple accesses to nodes by introducing a stack. This technique brings a performance gain which is also used later when more complex index structures are used.

Section 4 introduces holistic path join algorithms which use a compact stack encoding to optimize the join operation.

Section 5 presents holistic path join algorithms which avoid large intermediate results by using streams and the stack-encoding.

Section 6 presents to major index structures for join algorithms. The B+ tree is the basic structure which is used by the XR-tree and the XB-tree. For this reason its basic operations are shortly presented. In this section also the containment forest enhancement for the B+ tree is presented. It has been shown that the XB-tree outperforms the other structures in case of highly recursive data.

Section 7 shows algorithms which make use of index structures to skip elements which are not part of the solution. Especially for large input lists it speeds up the join operation. All index structures are used which were presented in section 6.

Section 8 presents details about the results from an implementation of the TreeMerge, StackTree and the PathStack.

The goal of this work is to show how new ideas increase the join performance. It is interesting how one idea leads to another one and how index structures speed up the join operation.

This work gives a detailed overview about join algorithms and shows in the practical section how the algorithms perform. It has been shown that there is not always one best algorithm for every query. For example the StackTree is very fast if the query has only two nodes, but it gets slower if the query is longer. A combination of these algorithms might be very interesting and can be future work on this sector.

2 Basic Definitions

2.1 XML introduction

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages. [12] XML is classified as an extensible language, because users can create their own elements. Its primary usage is to share structured data over different information systems. It is a simplified subset of the Standard Generalized Markup Language (SGML).

```
<book>
   <title> XML </title>
   <allauthors>
      <author> jane </author>
      <author> john </author>
   </allauthors>
   <year> 2000 </year>
   <chapter>
      <head> Origins </head>
      <section>
         <head> ...</ head>
         <section> ...</ section>
      </ section>
      <section> ...</ section>
   </ chapter>
   <chapter> ...</ chapter>
</ book>
```

Figure 1 [1]

Figure 1 [1] shows an example XML document which holds all information about a book.

There are two levels of correctness of a document:

- **Well-formed:** The syntax is correct. For example every start-tag is closed by an end-tag.
- **Valid:** A valid XML documents conforms to some semantic rules. These rules are for example defined in a DTD.

One big advantage of XML is the flexibility of processing XML documents. Traditionally the DOM or the SAX API is used.

The DOM API allows navigation of the entire document as if it were a tree of nodes representing the documents contents.

The SAX API works event driven and contents are reported as “callbacks” to various methods of a handler object.

2.2 DTD

Document Type Definition (DTD) is used to represent documents of a special type. [13] A DTD is consists of element types, attributes of elements, entities and notations. It declares the structure of the XML document.

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?, gender?,
socialsecuritynumber?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT gender (#PCDATA)>
<!ELEMENT socialsecuritynumber (#PCDATA)>
```

Figure 2 [13]

Figure 2 [13] shows an example which expresses [13]:

1. `people_list` is a valid element name, and an instance of such an element contains any number of `person` elements. The `*` denotes there can be 0 or more `person` elements within the `people_list` element.
2. `person` is a valid element name, and an instance of such an element contains one element named `name`, followed by one named `birthdate` (optional), then `gender` (also optional) and `socialsecuritynumber` (also optional). The `?` indicates that an element is optional. The reference to the `name` element name has no `?`, so a `person` element *must* contain a `name` element.
3. `name` is a valid element name, and an instance of such an element contains "parsed character data" (#PCDATA).
4. `birthdate` is a valid element name, and an instance of such an element contains character data.
5. `gender` is a valid element name, and an instance of such an element contains character data.
6. `socialsecuritynumber` is a valid element name, and an instance of such an element contains character data.

2.3 XPath

The XML Path Language (XPath) is a query language which makes it possible to select parts of the XML document. The current version is XPath 2.0.

An XPath expression consists of a sequence of location steps and each location step has three components [14]:

- an *axis*
- a *node test*
- and a *predicate*.

An Axis Specifier like 'child' specifies the direction to navigate from a context node. The node test and the predicate are used to filter the nodes specified by the axis specifier.

```
<A>
  <B>
   <C/>
  </B>
</A>
```

Figure 3 [14]

Figure 3 [14] shows a short XML file to show the abbreviated and the expanded syntax of XPath.

The simplest XPath query is `A/B/C` to select node C.

The expanded syntax for the same query is: `/child::A/child::B/child::C`

In this work the abbreviated syntax is used to save space.

2.4 Numbering Scheme

A very important operation is to test if a node is an ancestor/descendant of another node. It is certainly the same problem with parent child relationships. The intuitive solution is to search each node in the repository and search the other node in the sub tree. Certainly it is very inefficient so an index is needed which helps to determine this relationships efficiently. So a numbering scheme was introduced which can easily be generated and updated when the XML repository changes.

A position of an element can be represented by a 3-tuple: (DocumentId, StartPosition : EndPosition, LevelNumber)

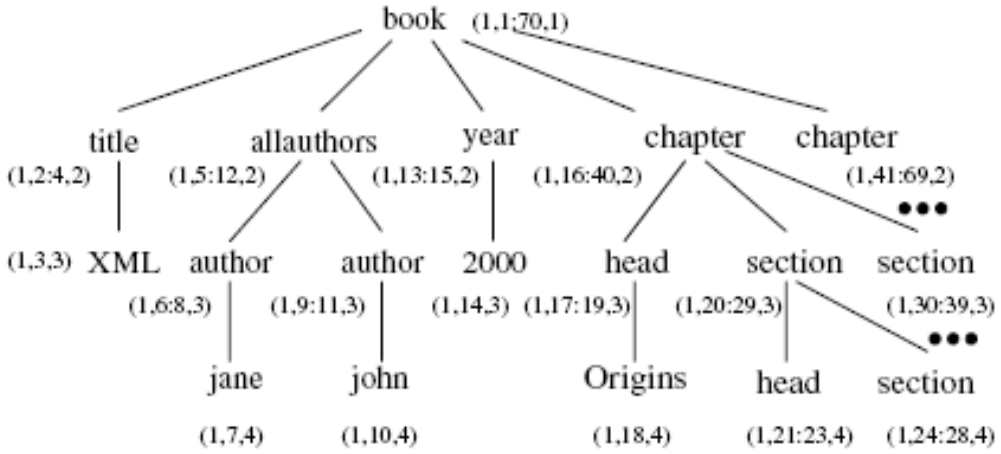


Figure 4 [1]

The DocumentId is the identifier for the XML document. In the example above it is set to 1. [1] If the structural join algorithm handles multiple documents this identifier is very important to determine a possible relationship. If the identifier of both nodes does not match they are not related in any way.

The StartPosition can be generated by counting all elements above the current node. For the EndPosition all elements under the current node will be counted. To handle future updates, where new elements will be inserted, it is possible to extend the (StartPosition : EndPosition) interval. When an insert occurs all Start and EndPositions have to be updated what can cause in the worst case that every tuple will be updated.

The LevelNumber is the nesting depth of the node in the document. It is needed to determine a parent-child relationship.

Structural relationships between nodes which are indexed in this fashion can be easily and efficiently determined. The node n_1 is encoded as (D_1, S_1 : E_1, L_1) and node n_2 is encoded as (D_2, S_2 : E_2, L_2).

Ancestor-descendant relationship between node n_1 and node n_2:

1) $D_1 = D_2$
2) $S_1 < S_2$ and $E_2 < E_1$

Parent-child relationship between node n_1 and node n_2:

1) $D_1=D_2$
2) $S_1 < S_2$ and $E_2 < E_1$
3) $L_1 + 1 = L_2$

The LevelNumber is not needed to check an ancestor-descendant relationship. The most important advantage of this representation is, that the ancestor-descendant relationship can be tested without any knowledge of the intermediate nodes in the path.

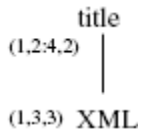


Figure 5 [1]

The relationship between 'title' and 'XML' is an ancestor-descendant and also a parent-child. The DocumentIds of the nodes are both 1. Then the StartPosition of 'title' (2) is smaller than the StartPosition of 'XML' (3) and the EndPosition of 'title' (4) is larger than the EndPosition of 'XML' (3). With these three checks an ancestor-descendant relationship is verified, but for a parent-child relationship the LevelNumbers have to be tested too. The LevelNumber from 'title' (2) increased with one is equal to the LevelNumber of 'XML' (3) which shows the parent-child relationship.

3 Structural Joins

An XML document organizes data in a tree structured format. The primitive tree relationships are parent-child and ancestor-descendant. A query specifies patterns of selection predicates on multiple items that have specific tree relationships.

For example take a look at the following XPath expression: book[title = 'XML']//author[. = 'jane']

It selects all author elements which have the content 'jane' and are descendants of a book with the title 'XML'. This shows that an XQuery expression can be represented as a node-labeled tree pattern. Node-labels are elements and string values. Such a query tree pattern can be decomposed into a set of parent-child or ancestor-descendant relationships. For this example the parent-child relationships are (book, title), (title, XML) and (author, jane) and the ancestor-descendant relationship is (book, author). Below the relationships are shown graphically. [1]

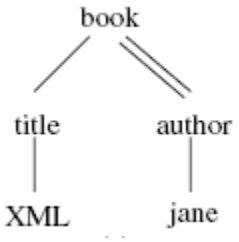


Figure 6 [1]

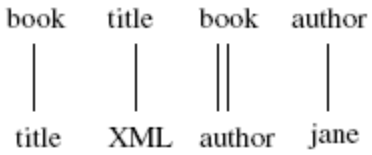


Figure 7 [1]

To find all occurrences each of the binary structural relationships will be matched against the XML database and the matches will be stitched together. Even a complex query tree can be decomposed into a set of basic binary relationships.

3.1 Example

The presented algorithms assume that they have as input the list of potential ancestors and the list of potential descendants. How these lists are generated is not covered by the join algorithms directly. It is mentioned to create an index which makes it possible to generate these lists quickly. Both lists are sorted with the StartPosition of the nodes.

For better understanding of the algorithms an example is used to show step by step how the algorithm is working. Each node is indexed with the numbering scheme to detect a structural relationship between two nodes.

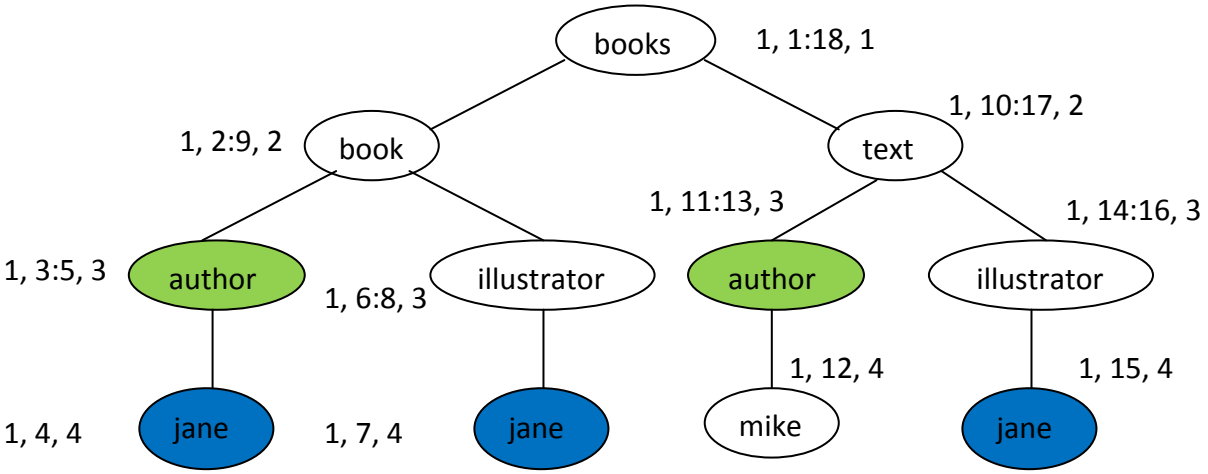


Figure 8

Jane nodes colored in blue are descendants and author nodes colored in green are ancestors. For this example the DocumentId of all nodes is 1.

For the example query author/jane the AList consists of nodes labeled with author and the DList consists only of nodes labeled with jane.

AList: (1, 3:5, 3), (1, 11:13, 3)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

To show how the algorithm matches an ancestor descendant relationship the example query book//jane is used.

AList: (1, 2:9, 2)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

3.2 Tree-Merge Algorithm

Both Tree-Merge algorithms assume that all nodes in the input lists (AList, DList) have the same DocumentId and are sorted by the StartPosition attribute. A modification to handle multiple DocumentIds is straight forward. The input lists would have to be sorted by the DocumentId first and afterwards sorted by the StartPosition. Then the check of the structural relationships must cover the DocumentId too. Obviously the nodes must have the same DocumentId to have a structural relationship like parent-child or ancestor-descendant.

Tree-Merge-Anc uses the ancestor list as outer join list and the Tree-Merge-Desc the descendant list. It is the decision of a query planner which algorithm performs best on a given query. As in relational databases it is important to collect statistics about the stored data to make predictions about query runtimes and make optimizations possible. Both algorithms have the disadvantage that in the worst case they have to skip a lot of nodes which are not part of the solution.

3.2.1 Tree-Merge-Anc

The algorithm is an extension of the tradition relational merge join algorithms.

```
Algorithm Tree-Merge-Anc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId
*/
/* AList is the list of potential ancestors, in sorted order of
StartPos */
/* DList is the list of potential descendants in sorted order of
StartPos */

01 begin-desc = DList->firstNode; OutputList = NULL;
02 for (a = AList->firstNode; a ! = NULL; a = a->nextNode) {
03   for (d = begin-desc; (d ! = NULL && d.StartPos <
          a.StartPos); d = d->nextNode) {
04            /* skipping over unmatchable d's */
     }

05   begin-desc = d;
06   for (d = begin-desc; (d ! = NULL && d.EndPos < a.EndPos); d
07        = d->nextNode) {
08             if ((a.StartPos < d.StartPos) && (d.EndPos <
               a.EndPos) [&& (d.LevelNum = a.LevelNum + 1)]) {
               /* the optional condition is for parent-child
                    relationships */
09                  append (a,d) to OutputList;
          }
     }
}
```

Figure 9 [1]

For simplicity the algorithm is represented in pseudo code. [1] The outer join operand is the ancestor which is similar to the MPMGJN algorithm which was proposed by Zhang et al. [2]. The optional clause [&& (d.LevelNum = a.LevelNum + 1)] is used to match parent-child relationships. The inner loop which skips unmatchable descendants ignores all descendants which have a lower StartPosition than the current ancestor.

Step-by-step execution with the example query author/jane:

AList: (1, 3:5, 3), (1, 11:13, 3)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) Node (1, 3:5, 3) from AList is stored into variable a
2) Node (1, 4, 4) from DList is stored into variable d
3) The first inner loop which skips unmatchable nodes terminates with the first check d.StartPos < a.StartPos (4<3 is not true)
4) The second loop checks d.EndPos < a.EndPos which is true (4 < 5)

5) The node-pair ((1, 3:5, 3), (1, 4, 4)) will be appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) & (d.LevelNum = a.LevelNum + 1) is true. (3<4 and 4<5 and 4=3+1)
6) Node (1, 7, 4) is assigned to d and the loop terminates d.EndPos < a.EndPos (7<5)
7) Node (1, 11:13, 3) is assigned to a in the first loop.
8) Node (1, 4, 4) and Node (1, 7, 4) are skipped. (4<11) , (7<11)
9) Node (1, 15, 4) is assigned to d
10) The skipping loop is terminated (15<11)
11) The second loop checks d.EndPos < a.EndPos which is false (15 < 13)
12) The AList is now empty and the algorithm terminates with the solution ((1, 3:5, 3),(1, 4, 4))

Step-by-step execution with the example query book//jane:

AList: (1, 2:9, 2)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) Node (1, 2:9, 2) from AList is stored into variable a
2) Node (1, 4, 4) from DList is stored into variable d
3) The first inner loop which skips unmatchable nodes terminates with the first check d.StartPos < a.StartPos (4<2)
4) The node-pair ((1, 2:9, 2), (1, 4, 4)) is appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) is true. (2<4 and 4<9)
5) Node (1, 7, 4) from DList is stored into variable d
6) The node-pair ((1, 2:9, 2), (1, 7, 4)) will be appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) is true. (2<7 and 7<9)
7) Node (1, 15, 4)is assigned to d and the loop terminates d.EndPos < a.EndPos (15<9)
8) The main loop terminates, because no ancestors are left
9) The algorithm outputs the solution ((1, 2:9, 2), (1, 4, 4)), ((1, 2:9, 2), (1, 7, 4)) and terminates.

Note: The optional clause [&& (d.LevelNum = a.LevelNum + 1)] is not used in ancestor-descendant relationship matching.

3.2.2 Analysis of Tree-Merge-Anc for ancestor/descendant relationship

Theorem 3.2.2.1 [1]:

> *The space and time complexities of Algorithm*
> *Tree-Merge-Anc are O(|AList| + |DList| + |OutputList|),*
> *for the ancestor-descendant structural relationship.*

When no two ancestor nodes are themselves related with an ancestor-descendant relationship the size of OutputList is O(|AList| + |DList|). The algorithm makes a single pass over AList and at most two passes over DList[2].

In the next case multiple nodes in AList have an ancestor-descendant relationship. Then multiple passes over the same set of descendants may be made. The size of the OutputList is quadratic in the size of the input lists O(|AList| * |DList|). In [1] is shown that the algorithm still has optimal time complexity, but is not I/O optimal.

3.2.3 Analysis of Tree-Merge-Anc for parent/child relationship

The time and space complexity is the same as if one were matching a parent-child relationship between the same input lists. Only the size of OutputList can be much smaller. In the case that all ancestor nodes form a long chain of length n and each node has two descendants, the size of OutputList is O(|Alist| + |DList|). The time complexity is O((|Alist| + |DList|)[2]). The I/O complexity is also quadratic. [1] The structure of the sample tree is shown below.

The ancestors form a list and each node has two descendants:

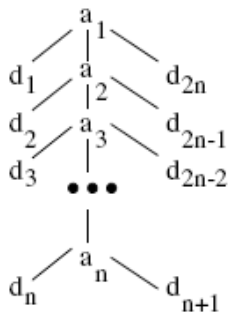


Figure 10 [1]

The graphic shows the matching from the structural relationships:

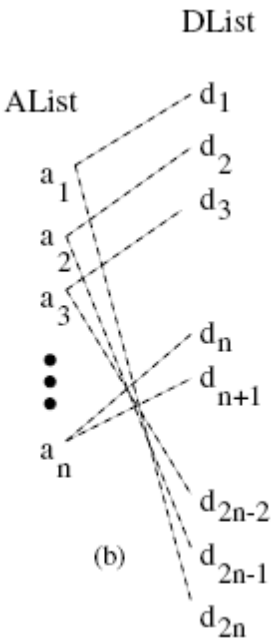


Figure 11 [1]

The dotted line shows the matching of the ancestor with its two descendants. When matching a_1 and d_{2n} all descendants from d_1 to d_{2n-1} have to be skipped. It is obvious that this is the worst case scenario for the Tree-Merge-Anc algorithm, because of the high amount of skipped descendants in every run.

3.2.4 Tree-Merge-Desc

```
Algorithm Tree-Merge-Desc (AList, DList)
/* Assume that all nodes in AList and DList have the same
DocId */
/* AList is the list of potential ancestors, in sorted order
of StartPos */
/* DList is the list of potential descendants in sorted
order of StartPos */

01 begin-desc = AList->firstNode; OutputList = NULL;
02 for (d = DList->firstNode; a ! = NULL; d = d->nextNode) {
03   for (a = begin-desc; (a ! = NULL && a.StartPos <
               d.StartPos); a = a->nextNode) {
04            /* skipping over unmatchable d's */
     }

05   begin-desc = a;
06   for (a = begin-desc; (a ! = NULL && a.EndPos <
           d.EndPos); a = a->nextNode) {
07             if ((a.StartPos < d.StartPos) && (d.EndPos <
                    a.EndPos) [&& (d.LevelNum = a.LevelNum +
                    1)]) {
               /* the optional condition is for parent-
                    child relationships */
08                  append (a,d) to OutputList;
          }
     }
}
```

Figure 12 [1]

For simplicity the algorithm is represented in pseudo code, but originally it has an error. The condition of the second inner loop is originally a.StartPos < a.StartPos which can never evaluate to true. The correct condition is a.StartPos < d.StartPos. The Tree-Merge-Desc algorithm deals with the case when the outer loop is the descendant list.

Step-by-step execution with the example query author/jane:

AList: (1, 3:5, 3), (1, 11:13, 3)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) Node (1, 4, 4) from DList is stored into variable d
2) Node (1, 3:5, 3) from AList is stored into variable a

3) Ancestor a will not be skipped, because check a.EndPos < d.StartPos is false. (5 < 4)
4) The node-pair ((1, 3:5, 3), (1, 4, 4)) will be appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) & (d.LevelNum = a.LevelNum + 1) is true. (3<4 & 4<5 & 4 = 3+1)
5) Node (1, 11:13, 3) from AList is stored into variable a
6) Loop terminates, because a.StartPos < d.StartPos is false (11<4)
7) Node (1, 7, 4) from DList is stored into variable d
8) Ancestor (1, 3:5, 3) will be skipped, because check a.EndPos < d.StartPos is true. (5 < 7)
9) Node (1, 11:13, 3) from AList is stored into variable a
10) Ancestor (1, 11:13, 3) will not be skipped, because check a.EndPos < d.StartPos is false. (13 < 7)
11) Node (1, 7, 4) is not part of the solution, because the loop check a.StartPos < d.StartPos fails
12) Node (1, 15, 4)from DList is stored into variable d
13) Ancestors (1, 3:5, 3), (1, 11:13, 3) will be skipped, because check a.EndPos < d.StartPos is true. (5< 15), (13 < 15)
14) No ancestors are left and the solution ((1, 3:5, 3), (1, 4, 4)) will be outputted and the algorithm terminates.

Step-by-step execution with the example query book//jane:

AList: (1, 2:9, 2)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) Node (1, 4, 4) from DList is stored into variable d
2) Node (1, 2:9, 2) from AList is stored into variable a
3) Ancestor a will not be skipped, because check a.EndPos < d.StartPos is false. (9 < 4)
4) The node-pair ((1, 2:9, 2), (1, 4, 4)) will be appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) & is true. (2<4 & 4<9)
5) The loop terminates, because no more ancestor are available
6) Node (1, 7, 4) from DList is stored into variable d
7) Ancestor a will not be skipped, because check a.EndPos < d.StartPos is false. (9 < 7)
8) The node-pair ((1, 2:9, 2), (1, 7, 4)) will be appended to the output list, because the condition ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos) is true. (2<7 and 7<9)
9) The loop terminates, because no more ancestor are available
10) Node (1, 15, 4) from DList is stored into variable d
11) Ancestor a will be skipped, because check a.EndPos < d.StartPos is false. (9 < 15)
12) The algorithm terminates and outputs the solution ((1, 2:9, 2), (1, 4, 4)), ((1, 2:9, 2), (1, 7, 4))

3.2.5 Analysis of Tree-Merge-Desc

There is no analog to theorem 3.2.2.1. The time complexity can be O(|AList| + |DList| + $|OutputList|^2$) in the worst case. [1] This happens for example if the first ancestor a_0 is the ancestor of all other nodes in AList.

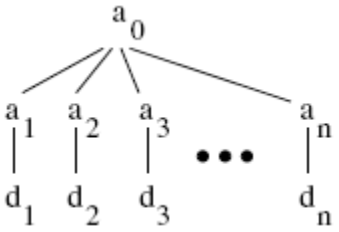


Figure 13 [1]

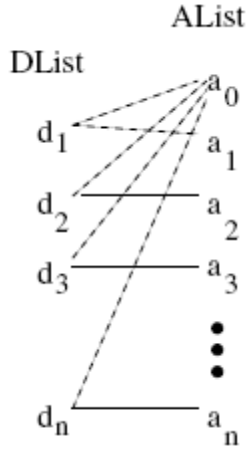


Figure 14 [1]

Here each node in DList has only two ancestors in AList. So the size of the OutputList is O(|AList| + |DList|). [1] The AList is repeatedly scanned which results in a time complexity of O(|AList| * |DList|). [1]

3.3 Stack-Tree algorithm

The stack-tree family of structural joins has no counterpart in traditional join processing. The idea is to introduce a stack which allows storing ancestors which are not fully processed and can be part of the final solution. The skipping of ancestors or descendants is not needed and gains a lot of performance. The worst case of the Tree-Merge algorithms is handled efficiently. The numbering schema which was introduced before is also used. Again AList and DList are given which contain all possible ancestors and descendants.

3.3.1 Stack-Tree-Desc

The output list i[(a_i, d_j)] is sorted by (DocumentId, d_j.StartPos, a_i.StartPos). This is extremely efficient in practice.

A stack will be used to store a sequence of ancestors. Each ancestor on the stack is a descendant of the nodes below it. If a new node from AList is a descendant of the current top of the stack it is simply pushed onto the stack. If a new node from DList is a descendant of the top of the stack it is a descendant of all elements on the stack. It is guaranteed that it is no descendant of any other node in AList. So this descendant is part of the solution with every ancestor node on the stack. If a new node from DList is not a descendant of the top of the stack the top element can be popped, because it can never be an ancestor of any future

descendant node. The stack elements will be popped till the stack is empty or the ancestor-descendant relationship is given.

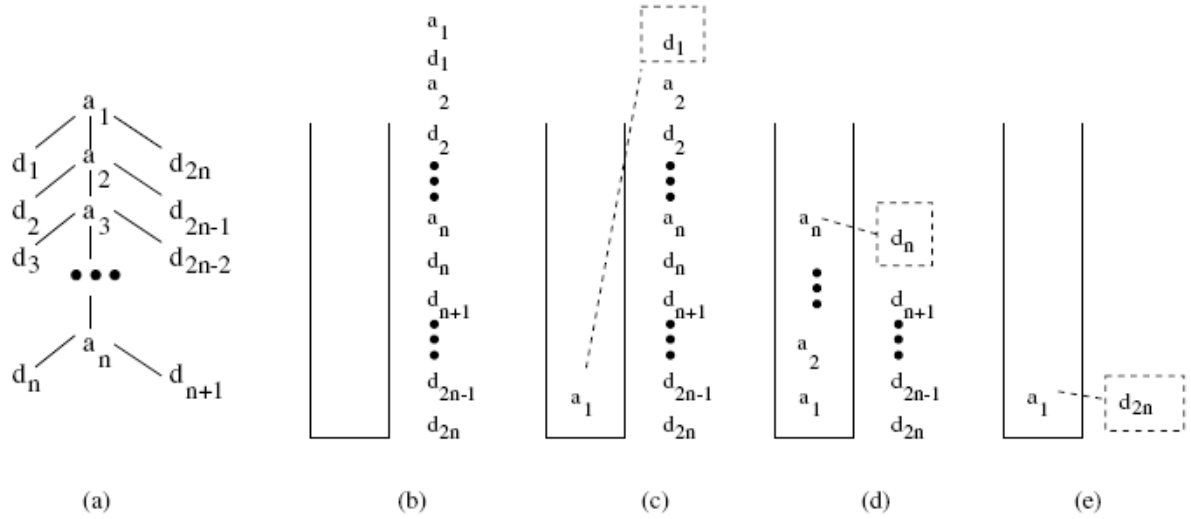


Figure 15 [1]

Figure 15 [1] shows the worst case for the tree-merge algorithms. As shown before a tree-merge algorithm has to skip a lot of ancestor or descendant elements depending of which is the inner/outer join operand. The stack which is empty in the beginning and both input lists are conceptually merged. (see b) a_1 has been pushed onto the stack and is now compared with d_1. This pair is part of the solution. Figure d shows the stack and the elements after many execution steps. At last the element a_1 on the stack will be compared with d_{2n} and will be added to the output list. (see c-e)
This shows that the stack-tree algorithms perform much better, because the skipping of elements is not needed and every descendant element will be checked only once. Skipping of elements means that elements will not be processed more than once. Only elements on the stack may be scanned more than once.

```
Algorithm Stack-Tree-Desc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId
*/
/* AList is the list of potential ancestors, in sorted order of
StartPos */
/* DList is the list of potential descendants in sorted order of
StartPos */

a = AList->firstNode;
d = DList->firstNode;
OutputList = NULL;

01 while (the input lists are not empty or the stack is not
          empty){
02   if ((a.StartPos > stack->top.EndPos) && (d.StartPos >
          stack->top.EndPos)) {
          /* time to pop the top element in the stack */
03        tuple = stack->pop();
04   } else if (a.StartPos < d.StartPos) {
05        stack->push(a)
06        a = a->nextNode
     }else{
07        for (a1 = stack->bottom; a1 ! = NULL; a1 = a1->up) {
08             [if(d.LevelNum = stack->top.LevelNum + 1)]
09                  append (a1,d) to OutputList
          }
10        d = d->nextNode
     }
}
```

Figure 16 [1]

The original Stack-Tree-Desc algorithm [1] contained some errors. The stack is initially empty which results in an error when resolving stack->top.EndPos in the first run of the while loop. Then imagine the case when the last ancestor from AList is pushed onto the stack. The variable a is now null and it is not possible for the algorithm to get into the branch where the top stack element is popped. From this time on the algorithm will cycle in an endless loop, because the stack can never get empty. Conditions were corrected and the condition (a.StartPos > stack->top.EndPos) was eliminated which is not needed.

```
Algorithm Stack-Tree-Desc (AList, DList)
/* Assume that all nodes in AList and DList have the same DocId
*/
/* AList is the list of potential ancestors, in sorted order of
StartPos */
/* DList is the list of potential descendants in sorted order
of StartPos */
/* Stack is empty */

a = AList->firstNode;
d = DList->firstNode;
OutputList = NULL;

01 while ((DList != NULL || d != null) && (AList != NULL
          || a != null || !stack.empty)) {
02   if (!stack.empty && d != null && (d.StartPos > stack-
          >top.EndPos)) {
          /* time to pop the top element in the stack */
03        tuple = stack->pop();
04   } else if (a != null  && d != null  && a.StartPos <
                  d.StartPos) {
05        stack->push(a)
06        a = a->nextNode
     }else{
07        for (a1 = stack->bottom; a1 ! = NULL; a1 = a1->up) {
08             [if(d.LevelNum = stack->top.LevelNum + 1)]
09                  append (a1,d) to OutputList
          }
10        d = d->nextNode
     }
}
```

Figure 17

The loop now terminates if all descendants have been processed or no ancestors are left. This avoids an endless loop if ancestor elements remain on the stack. Now an ancestor element will be popped from the stack when its StartPosition is smaller than d.StartPosition. It is obvious that this ancestor cannot be part of the solution and can be dropped.

The optional clause [if(d.LevelNum = stack->top.LevelNum + 1)] is needed to check a parent-child relationship between an ancestor and a descendant.

Step-by-step execution with the example query author/jane:

AList: (1, 3:5, 3), (1, 11:13, 3)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) (1, 3:5, 3) is stored into a
2) (1, 4, 4) is stored into d

3) a.StartPos < d.StartPos so node (1, 3:5, 3) will be pushed onto the stack and (1, 11:13, 3) will be stored into a
4) Next loop cycle the else branch will be entered. The only element on the stack is (1, 4, 4). The parent-child relationship is given (d.LevelNum = stack->top.LevelNum + 1) so the pair ((1, 3:5, 3), (1, 4, 4)) will be appended to the solution. Ancestor (1, 7, 4) will be stored into d.
5) Then the stack element (1, 3:5, 3) will be popped, because d.StartPos > stack->top.EndPos is true. (7 > 5)
6) Descendant (1, 7, 4) will be skipped in the else branch. The stack is empty and node (1, 15, 4) will be stored into d.
7) a.StartPos < d.StartPos so node (1, 11:13, 3) will be pushed onto the stack. Now the ancestor list is empty and a stays null.
8) d.StartPos > stack->top.EndPos and node (1, 11:13, 3) will be popped.
9) Now there is no ancestor left and the algorithm finishes.

In case of an ancestor-descendant relationship the execution is similar. When iterating over the stack elements, no check of the node levels is needed.

Step-by-step execution with the example query author//jane:

AList: (1, 3:5, 3), (1, 11:13, 3)
DList: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) (1, 3:5, 3) is stored into a
2) (1, 4, 4) is stored into d
3) a.StartPos < d.StartPos so node (1, 3:5, 3) will be pushed onto the stack and (1, 11:13, 3) will be stored into a
4) Next loop cycle the else branch will be entered. The only element on the stack is (1, 4, 4). The pair ((1, 3:5, 3), (1, 4, 4)) will be appended to the solution. Ancestor (1, 7, 4) will be stored into d.
5) Then the stack element (1, 3:5, 3) will be popped, because d.StartPos > stack->top.EndPos is true. (7 > 5)
6) Descendant (1, 7, 4) will be skipped in the else branch. The stack is empty and node (1, 15, 4) will be stored into d.
7) a.StartPos < d.StartPos so node (1, 11:13, 3) will be pushed onto the stack. Now the ancestor list is empty and a stays null.
8) d.StartPos > stack->top.EndPos and node (1, 11:13, 3) will be popped.
9) Now there is no ancestor left and the algorithm finishes.

3.3.2 Analysis of Stack-Tree-Desc

Theorem 3.3.2.1 [1]:

> *The space and time complexities of Algorithm*
> *Stack-Tree-Desc are O(|AList| + |DList| + |OutputList|)*
> *for both ancestor-descendant and parent-child structural relationships.*
> *Further, Algorithm Stack-Tree-Desc is a non-blocking algorithm.*

Each page of the input lists is read once. The result is output as soon as it is computed.

Theorem 3.3.2.2 [1]:

> *The I/O complexity of Algorithm*
> *Stack-Tree-Desc* $O(\frac{|AList|}{B} + \frac{|DList|}{B} + \frac{|OutputList|}{B})$,
> *for ancestor-descendant and parent-child structural relationships,*
> *where B is the blocking factor.*

3.3.3 Stack-Tree-Anc

In this case the output list [(ai,dj)] is sorted by (DocumentId, ai.StartPos, dj.StartPos). It is not trivial to modify the Stack-Tree-Desc to produce this output. If a node from AList is an ancestor of any node of DList then every node a' from AList that is ancestor from a is also ancestor of d. The StartPosition from a' is smaller than the StartPosition of a the output of the pair (a,d) has to be delayed till the pair (a', d) has been added to the output list. It is also possible that a new element d' after d joins with a' as long as a' is on the stack. So the pair (a, d) cannot be appended to the output list until a' is popped from the stack. There can be built up large join results which cannot yet be output. [1]

In the Stack-Tree-Anc algorithm also a stack is used to store ancestor nodes. Each ancestor on the stack is a descendant of the nodes below it. Two lists are associated with each ancestor on the stack. The first list is called self-list. [1] It is a list of result elements from the join of this element with DList elements. The second one is called inherit-list. [1] This list contains join results involving AList elements that where descendants of the current top of the stack.

When a new node from AList is a descendant from the node on top of the stack it is simply pushed onto the stack. When a new node from DList is a descendant from the current top of stack it is added to the self-lists of the nodes on the stack. If no node (from AList or DList) is a descendant from the top of the stack it can be popped. It is guaranteed that no future node is a descendant of the current top of stack. [1] All nodes will be popped till a descendant is found. When the bottom element is popped its self-list is output and then follows its inherit-list. When any other element in the stack is popped no output is generated. Instead its inherit-list will be appended to its self-list, and the result appended to the inherit-list of the new top of stack.

```
Algorithm Stack-Tree-Anc (AList, DList)
/* Assume that all nodes in AList and DList have the same
DocId */
/* AList is the list of potential ancestors, in sorted order
of StartPos */
/* DList is the list of potential descendants in sorted order
of StartPos */

a = AList->firstNode;
d = DList->firstNode;
OutputList = NULL;

01 while (the input lists are not empty or the stack is not
                empty) {
02   if ((a.StartPos > stack->top.EndPos) && (d.StartPos >
                stack->top.EndPos)) {
          /* time to pop the top element in the stack */
03        tuple = stack->pop();
04        if (stack->size == 0) {
               /* we just popped the bottom element */
05             append tuple.inherit-list to OutputList
          } else {
06             append tuple.inherit-list to tuple.self-list
07             append the resulting tuple.self-list to stack-
                         >top.inherit-list
          }
08   } else if (a.StartPos < d.StartPos) {
09        stack->push(a)
10        a = a->nextNode
     }else {
11        for (a1 = stack->bottom; a1 ! = NULL; a1 = a1->up) {
12             if (a1 == stack->bottom)
13                  append (a1,d) to OutputList
               else
14                  append (a1,d) to the self-list of a1
          }
15        d = d->nextNode
     }
} [1]
```

Figure 18 [1]

3.3.4 Analysis of Stack-Tree-Anc

The main difference to the Stack-Tree-Desc is that join results are associated with nodes in the stack. The only thing which has to be analyzed is the appending of the lists, each time when a stack element gets popped. If the lists are implemented as linked lists these append operations can be carried out in unit time. Combined with the analyses of the Stack-Tree-Desc the algorithm is O(|input| + |output|) in the worst case. [1]

For the I/O complexity analysis it cannot be assumed that all result lists fit to memory. A buffer management is needed. The only performed operation on the lists is to append one list to another. The read out at the end is the only exception. Access to the tail of each list is needed during the computation, so rest of the list can be paged out. When list x is appended to list y it is not necessary to hold the head of list x in memory. The append operation only establishes a link to this head in the tail of y. The number of entries in the lists is equal to the number of entries in the output.

Theorem 3.3.4.1 [1]:

> *The space and time complexities of Algorithm*
> *Stack-Tree-Anc are O(|AList| + |DList| + |OutputList|),*
> *for both ancestor-descendant and parent-child structural relationships.*
> *The I/O complexity of Algorithm Stack-Tree-Anc is*
> $O(\frac{|AList|}{B} + \frac{|DList|}{B} + \frac{|OutputList|}{B})$*, for both ancestor-descendant*
> *and parent-child structural relationships, where B is the blocking factor.*

3.4 Summary

The first structural join algorithms split a query in binary relationships and merge the intermediate results. The TreeMerge algorithm is the simplest algorithm which uses two nested loops to match all nodes which are part of the result. Obviously many nodes get scanned more than once which causes a quadratic runtime.

The StackTree introduces a stack which helps to cache ancestor nodes to avoid the repeated scanning of nodes. This idea brings a huge performance gain and has shown to be a very lightweight implementation.

One big disadvantage of all algorithms presented in this section is that a longer query has to be split. The merge of the intermediate results decreases the performance. Later algorithms try to avoid splitting and process the query at once.

Algorithms which deal especially with sibling relationships have been proposed by [15]. This work proposes two structural join algorithms which efficiently process sibling relationships.

Another work proposes the PBiTree [16] which helps to determine the ancestor-descendant relationship between two nodes.

The paper [18] proposes a new operator called structural semi-join and the algorithms for efficient processing XML path queries. With this operator it is possible to process queries which return the descendant or ancestor nodes only.

4 Holistic Path Join Algorithms

Previous algorithms decompose the twig pattern into binary structural relationships. The matching is achieved by using structural join algorithms to match binary relationships and stitching together these matches. A limitation of this approach is that intermediate results can get very large which decreases the join performance. A holistic twig join creates no large intermediate results and gains more performance. The numbering scheme (DocumentId, StartPosition : EndPosition, LevelNumber) mentioned above is used to determine structural relationships between nodes. Also a chain of linked stacks is used to compactly represent partial results which are then used to obtain matches to the query twig.

4.1 Notation

The StartPosition and EndPosition will be denoted as LeftPosition and RightPosition, but the functionality does not change. It is just a more intuitive label. [3]

A twig node supports the following operations:

1) isLeaf: Node -> Bool
2) isRoot: Node -> Bool
3) parent: Node -> Node
4) children: Node -> {Node}
5) subtreeNodes: Node -> {Node}

Path queries have only one child per node. Otherwise children(q) returns a set of children nodes. Result of subtreeNodes(q) is the node q and all its descendants.

With each node q in a query twig pattern a stream Tq is associated. The stream contains the positional representation of the nodes which match twig pattern node q. This can be obtained by using an index or another efficient access method. The nodes in the stream are ordered by (DocumentId, LeftPosition) values. [3]

A stream supports the following operations:

1) eof
2) advance
3) next
4) nextL
5) nextR

Operation nextL returns the LeftPosition coordinates in the positional representation of the top element in the stack and operation nextR the RightPosition.

The nodes in stack Sq (from bottom to top) lie on a root-to-leaf path in the XML database. The set of stacks always keep a compact encoding of partial and total answers to the query twig pattern.

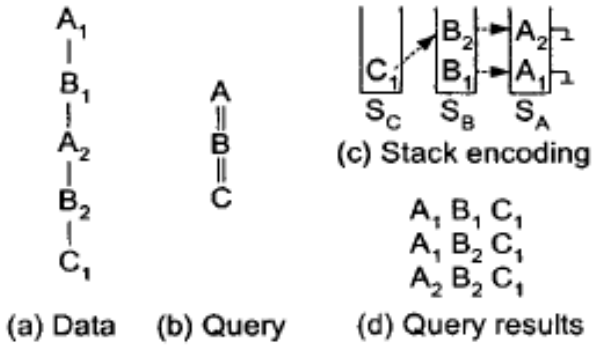


Figure 19 [3]

Figure 19 [3] a shows an example repository and graph b shows the query.[3] The query twig pattern searches for the nodes A, B and C in an ancestor-descendant relationship. The results are encoded in the stacks of the query nodes (c).

[A_2, B_2, C_1] is a solution, because C_1 points to B_2 and B_2 points to A_2. A_1 is below A_2 on the stack S_A which makes [A_1, B_2 , C_1] to an answer. [A_1, B_1, C_1] is also an answer, because B_1 is below B_2 on stack S_B and B_1 points to A_1.
[A_2, B_1 , C_1] is not an answer, because A_2 is above the node A_1 on stack S_A to which B_1 points.

4.2 PathStack

The PathStack[3] computes answers to a query path pattern for the case when the streams contain the same DocumentId. If multiple XML documents should be handled it can be adapted quite easily. In that case the DocumentId of two nodes has to be tested for equality before the streams and stacks are manipulated.

```
Algorithm PathStack(q)
01 while !end(q)
02   q_min = getMinSource(q)
03   for q_i in subtreeNodes(q) // clean stacks
04        while (!empty(S_qi) && topR(S_qi ) < nextL(T_qmin))
05             pop(Sqi)
06   moveStreamToStack(T_qmin,S_qmin, pointer to top(S_parent (q_min)))
07   if(isLeaf(q_min))
08        showSolutions(S_qmin , 1)
09        pop(S_qmin)

Function end(q)
     return for all q_i element subtreeNodes(q):isLeaf(q_i)=>eof(T_qi )

Function getMinSource(q)
     return q_i element subtreeNodes(q) such that nextL(T_qmin)
     is minimal

Procedure moveStreamToStack (T_q, S_q, p)
01   push(S_q, (next(T_q),p))
02   advance (Tq)
```

Figure 20 [3]

The key idea is to repeatedly construct compact stack encodings of partial and total answers to the query path pattern. This will be achieved by iterating through the stream nodes in sorted order. (The nodes are sorted by their LeftPosition) The query path pattern will be matched from the query root down to the query leaf. [3]

At Line 2 the Algorithm identifies the stream containing the next node to be processed. Lines 3-5 remove partial answers which cannot be extended to total answers. Line 6 augments the partial answers which are encoded in the stacks with the new stream node. When a node is pushed onto the stack S_{qmin} (q_{min} is the leaf node of the query path) the stacks contain an encoding of total answers and function showSolutions[3] is invoked.

```
Procedure showSolutions (S N, SP)
// Assume, for simplicity, that the stacks of the query
// nodes from the root to the current leaf node we
// are interested in can be accessed as S[1] . . . . . S[n].
// Also assume that we have a global array index[1..n]
// of pointers to the stack elements.
// index[i] represents the position in the i'th stack that
// we are interested in for the current solution, where
// the bottom of each stack has position 1.
// Mark we are interested in position SP of stack SN.

01 index[SN] = SP
02 if (SN== 1) // we are in the root
03 // output solutions from the stacks
04 output (S[n].index[n],..., S[1].index[1])
05 else // recursive call
06 for i = 1 to S[SN].index[SN].pointer_to_parent
07 showSolutions (SN – 1, i)
```

Figure 21 [3]

ShowSolutions outputs query path answers encoded in the stacks as n-tuples sorted in leaf-to-root order. That ensures that over the sequence of invocations the answers are also computed in leaf-to-root order. The showSolutions function above is for the case when only ancestor-descendant relationships are in the query path. When parent-child edges are also present the LevelNumber has to be checked. The PathStack algorithm does not need to be changed. Each time when it calls showSolutions it has to ensure that it does not output incorrect tuples. Otherwise it causes unnecessary work. To avoid this unnecessary work the recursive call (lines 6-7) has to be modified to check for parent-child edges. Only a single recursive call ((showSolutions(SN-1, S[SN].index[SN].pointer_to_the_parent_stack)) has to be invoked after verifying that the LevelNumber of the two nodes differs by one. Looping through all nodes in the stack S[SN-1] would still be correct, but causes useless work, which decreases the performance of the algorithm.

If the final answers should be sorted root-to-leaf it does not suffice that each invocation of showSolutions outputs answers in root-to-leaf order. To accomplish this order the algorithm would need to block answers and delay their output till there is no answer prior to them.

Step-by-step execution with the example query author//jane:

Stream T_{author}: (1, 3:5, 3), (1, 11:13, 3)
Stream T_{jane}: (1, 4, 4), (1, 7, 4), (1, 15, 4)

1) Author node (1, 3:5, 3) has the smallest StartPos and will be processed.
2) All stacks are empty at startup so no cleanup is needed.

3) (1, 3:5, 3) will be pushed onto S_{author}
4) Jane node (1, 4, 4) will be processed in the next loop cycle
5) (1, 4, 4) will be popped onto S_{jane} and linked to the top of S_{author}
6) Jane is a leaf node so show solution outputs ((1, 3:5, 3), (1, 4, 4)) and pops (1, 4, 4) from S_{jane}
7) Then jane node (1, 7, 4) will be processed.
8) Again is will be pushed onto the stack S_{jane} and linked to the top of S_{author}
9) Jane is a leaf node of the query so show solution is executed and jane node (1, 7, 4) will be popped from stack
10) Next loop cycle processes author node (1, 11:13, 3) and pops (1, 3:5, 3) from stack S_{author}.
11) Then node (1, 11:13, 3) will be pushed onto S_{author}
12) Then last jane node (1, 15, 4) will be popped onto S_{jane}
13) ShowSolutions will be executed and jane node (1, 15, 4) popped.
14) Then the algorithm terminates, because no leaf nodes are left.

4.3 Analysis of PathStack

PROPOSITION 4.3.1 [1]:

If we fix node Y, the sequence of cases between node Y and nodes X on increasing order of LeftPos *(L) is:* (1|2)*3*4*. *Cases 1 and Cases 2 are interleaved, then all nodes in Case 3 before any node in Case 4, and finally all nodes in Case 4*

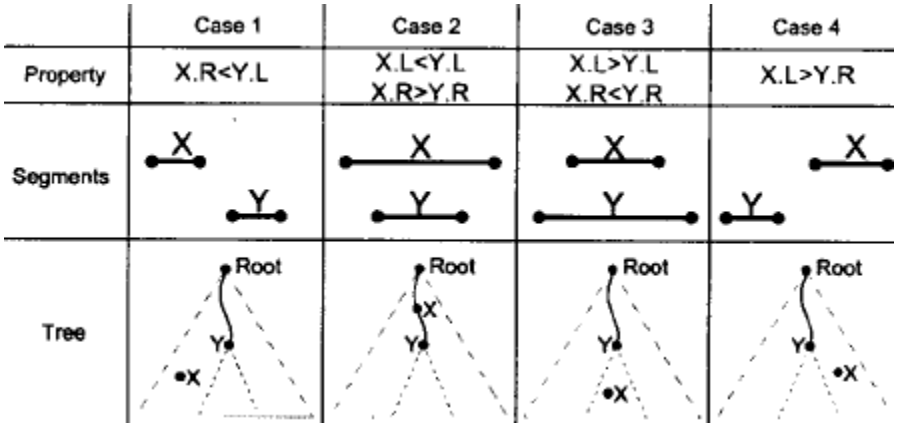


Figure 22 [3]

LEMMA 4.3.1 [3]:

Suppose that for an arbitrary node q in the path pattern query, we have that getMinSource(q) = q_N. *Also, suppose that* t_{qn} *is the next element in q's stream. Then, after* t_{qN} *is pushed on to stack* S_{qN} *, the chain of stacks from* S_{qN} *to* S_q *verifies that their labels are included in the*

chain of nodes in the XML data tree from t_{qN} to the root.

For each node t_{qmin} pushed onto stack S_{qmin} the above lemma shows that all answers in which t_{qmin} is a match for query node q_{min} will be output.

THEOREM 4.3.2 [3]:

Given a query path pattern q and an XML database D, Algorithm PathStack *correctly returns all answers for q on D.*

An XML path of length n is given. PathStack takes n input lists of nodes which are sorted by (DocumentId, LeftPosition) and computes a sorted output list of n tuples that match the query path. The I/O and CPU costs of PathStack are linear in the sum of sizes of the n input lists. The invocations of showSolutions are not included in this calculation. The costs of showSolutions is proportional to the size of the output list.[3]

The worst-case space complexity of PathStack is the minimum of the sum of sizes of the n input lists and the maximum length of a root-to-leaf path in the XML database.[3]

4.4 PathMPMJ

The generalization of the MPMGJN algorithm [3] (Multi-Predicate Merge Join) for path queries processes one stream at a time to compute all solutions. As example the path $q_1//q_2//q_3$ is used.

The algorithm uses the first element from the stream T_{q1} and generates all solutions that use that element from T_{q1}. Then T_{q1} will be advanced and T_{q2} and T_{q3} will be backtracked to the earliest position which might lead to a solution. This step will be repeated until T_{q1} is empty. [3]

To generate all solutions the algorithm starts recursively with the first marked element in T_{q2}. The step gets all solutions that use that element and the calling element in T_{q1} and advances stream T_{q2} until there are no more solutions with the current element in T_{q2}.

One mark per stream is too inefficient in practice, because all marks have to point to the earliest segment that can match the current element in T_{q1}. It is better to use a stack of marks. In this optimization each node will not have a single mark in the stream, but k marks. (k is the number of ancestors in the query) [3]

Each mark points to an earlier position in the stream. For query node q the i'th mark is the first point in T_q such that the element in T_q starts after the current element in the stream of q's i'th ancestor.

```
Algorithm PathMPMJ (q)
01 while (!eof(Tq) && (isRoot(q)||
               nextL(q) < nextR(parent(q))))
02   for (qi element subtreeNodes(q)) // advance descendants
03        while (nextL(qi) < nextL(parent(qi)))
04             advance (Tqi)
05        PushMark (Tqi)
06   if (isLeaf(q)) // solution in the streams' heads
          outputSolution()
07   else PathMPMJ (child(q))
08   advance (Tq)
09   for (qi element subtreeNodes(q)) // backtrack descendants
10        PopMark (Tqi)
```

Figure 23 [3]

THEOREM 4.4.1 [3]:

Given a query path pattern q and a XML database D, Algorithm PathMPMJ *correctly returns all answers for q on D.*

4.5 Summary

Holistic Path join algorithms try to avoid splitting and merging of intermediate results by processing a root-to-leaf path from a query at once. The most important algorithm is the PathStack which extends the idea from the StackTree and uses more than one stack to cache nodes. This stack encoding has been proven as very efficient and the PathStack is very fast in processing longer queries which are not nested.

If the query consists of more than one root-to-leaf path splitting becomes unavoidable which decreases the performance. Twig Join algorithms from section 5 were designed to deal with a query twig at once without splitting up the query.

5 Twig Join Algorithms

To decompose the twig into multiple root-to-leaf patterns and use PathStack to identify solutions and then merge-join these solutions is straight-forward. This computation has the same problem as techniques which base on binary structural joins. Many intermediate results may not be part of any final answer.

Example:

XML database

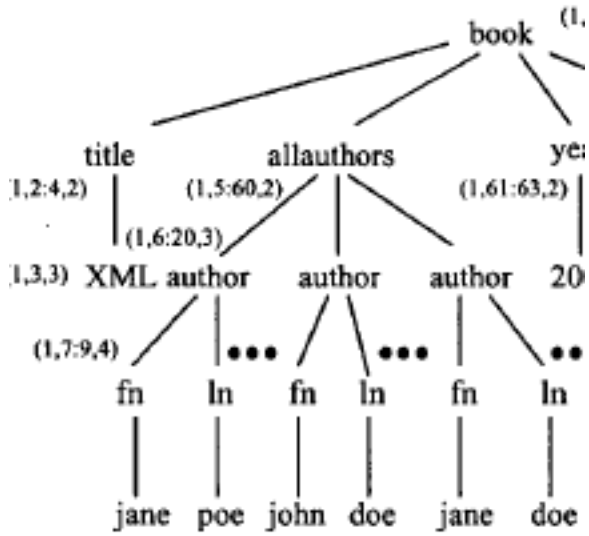


Figure 24 [3]

Query Twig

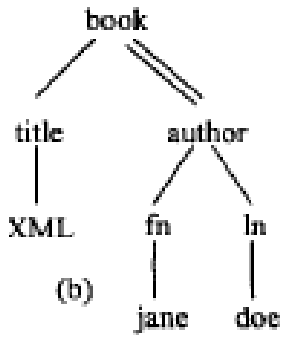


Figure 25 [3]

When matching the query twig against the given XML database the paths author-fn-jane and author-ln-doe have two solutions. The query twig pattern has only one solution.

5.1 Twig Stack

If the query paths have many solutions that are not part of the final output the usage of PathStack is suboptimal. The computation costs for a twig pattern would not be proportional to the sizes of the input and output only. Also the sizes of the intermediate results are added to the computation costs. The TwigStack avoids the costs of the intermediate results. [3]

Again the presented algorithm assumes that the DocumentId from all nodes is equal. In the case of more than one XML document the algorithm would have to check the DocumentIds first before manipulating the nodes in the streams and stacks.

```
Algorithm TwigStack(q)
     // Phase 1
01 while !end(q)
02   q_act = getNext(q)
03   if (!isRoot(q_act))
04        cleanStack(parent(q_act), nextL(q_act)
05   if (isRoot(q_act) || !empty(S_parent(q_act)))
06        cleanStack(q_act, next(q_act))
07        moveStreamToStack(T_qact, S_qact, pointer to
top(S_parent(q_act)))

08        if(isLeaf(q_act))
09             showSolutionsWithBlocking(S_qact,  1)
10             pop(S_qact)
11   else adcance(T_qact)

     // Phase 2
12 mergeAllPathSolutions()

Function getNext(q)
01 if (isLeaf(q)) return q
02 for q_i in children(q)
03   n_i = getNext(q_i)
04   if (n_i != q_i) return n_i
05 n_min = minarg_ni nextL(T_ni)
06 n_max = maxarg_ni nextL(T_ni)
07 while nextR(T_q) < nextL(T_nmax)
08   advance(T_q)
09 if nextL(T_q) < nextL(T_nmin) return q
10 else return n_min

Procedure cleanStack(S, actL)
01 while !empty(S) && (TopR(S) < actL)
02   pop(S)
```

Figure 26 [3]

The TwigStack algorithm operates in two phases.[3] In the first phase (lines 1-11) some solutions to individual query root-to-leaf paths are computed. In this step not all solutions are computed. In the second phase (line 12) the solutions are merge-joined to compute the answers to the query twig pattern.

There is a difference of PathStack and the first phase of TwigStack. First TwigStack ensures that before a node h_q from stream T_q is pushed on stack S_q, node h_q has a descendant h_{qi} in each of the streams T_{qi}. It also ensures that each of the nodes h_{qi} recursively satisfies the first property. [3]

When the query twig pattern has only ancestor-descendant edges, each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution to each of the other root-to-leaf paths. [3] That ensures that no intermediate result is larger than the final answer.

The second merge-join phase of TwigStack is linear in the sum of its input and output sizes, but only if the inputs are sorted in order of common prefixes of the different root-to-leaf paths. The solutions to individual query paths have to be output in root-to-leaf order which needs blocking. Function showSolutions cannot be used anymore, because it outputs the solutions in leaf-to-root order.

Consider the same query twig pattern and XML document from before. Before TwigStack pushes an author node onto the stack S_{author}, it ensures that this author node has a descendant fn node in the stream T_{fn} and a descendant ln node in the stream T_{ln}. This causes that only one of the three author nodes is pushed onto the stack. Finally the merge join phase computes the desired answer.

5.1.1 Analysis of TwigStack

DEFINITION 5.1.1.1 [3]:

> *Consider a twig query Q. For each node* q ∈ subtreeNodes(Q) *we define the* head *of q, denoted* h_q*, as the first element in* T_q *that participates in a solution for the sub-query rooted at q. We say that a node q has a* minimal descendant extension *if there is a solution for the subquery rooted at q composed entirely of the head elements of* subtreeNodes *(q).*

LEMMA 5.1.1.1 [3]:

> *Suppose that for an arbitrary node q in the twig query we have that* getNext(q) = q_N*. Then, the following properties hold:*
> *1.* q_N *has a minimal descendant extension.*
> *2. For each node q' ∈* subtreeNodes(q_N), *the first element in* T_q*' is* h_q*'.*

3. Either (a) $q = q_N$ or (b) parent(q_N) *does not have a minimal right extension because of q_N (and possibly other nodes). In other words, the solution rooted at* p = parent(q_N) *that uses h_p does not use h_q for node q but some other element whose L component is larger than that of h_q.*

With the lemma above, when getNext returns q_N, it is guaranteed that h_{qN} has a descendant extension in subtreeNodes(q_N). Any element in the ancestors of q_N that uses h_{qN} in a descendant was returned by getNext before h_{qN}.

THEOREM 5.1.1.1 [3]:

Given a query twig pattern q and an XML database D, Algorithm TwigStack *correctly returns all answers for q on D.*

Proof [3]:

In Algorithm TwigStack, we repeatedly find getNext(q) for query q (line 2). Assume that getNext(q) = q_N. Let A_{qN} be the set of nodes in the query that axe ancestors of q_N. We know that getNext already returned all elements from the streams of nodes in A_{qN} that are part of a solution that uses h_{qN}. If q != q_N, in line 4 we pop from parent(q_N)'s stack all elements that are guaranteed not to participate in any new solution. After that, in line 5 we test whether h_{qN} participates in a solution. We know that q_N has a descendant extension by Lemma 4.1 (see Lemma 5.1.1.1 above)*, property 1. If q != q_N and parent(q_N)'s stack is empty, node q_N does not have an ancestor extension. Therefore it is guaranteed not to participate in any solution, so we advance q_N in line 11 and continue with the next iteration. Otherwise, node q_N has both ancestor and descendant extensions and therefore it participates in at least one solution.*
We then clean q_N's stack (line 6) and push h_{qN} to it (line 7). Finally, if q_N is a leaf node, we output the stored solutions from the stacks (lines 8-10).

The optimality can only be proved when the query twig pattern has only ancestor-descendant edges. Only elements with a descendant and an ancestor extension are pushed onto the stack. This ensures that no element is pushed onto the stack which does not participate in any solution. The merge post processing step is optimal, which brings the result:

Given is a query twig pattern q with n nodes and only descendant edges and a XML database D. TwigStack has worst-case I/O and CPU time complexities linear in the sum of the n input and output lists. [3] O(|Input| + |Output|).

The worst-case space complexity of TwigStack is the minimum of (i) the sum of sizes to the n input lists and (ii) n times the maximum length of a root-to-leaf path in D. [3] O=(min(|Input|, n*maxRootToLeafPathLength))

In case of query twigs with only ancestor-descendant edges, the worst-case time complexity of TwigStack is independent of the sizes of solutions to any root-to-leaf path of the twig.[3]

In case of parent-child edges in the query twig TwigStack is no longer guaranteed to be I/O and CPU optimal. The algorithm can produce a solution for one root-to-leaf path that does not match with any solution in another root-to-leaf path.

As example consider the query twig pattern with the three nodes A, B and C. There are parent-child edges between (A, B) and (A, C). The XML data tree consists of node A_1, with children A_2, B_2, C_2. Node A_2 has children B_1, C_1. The three streams T_A, T_B und T_c have as first element A_1, B_1, C_1. It cannot be determined if any of them participates in another solution without advancing other streams, but a stream cannot be advanced before knowing if it participates in a solution. Optimality is no longer guaranteed. [3]

5.2 Twig²Stack

The Twig²Stack algorithm [9] was inspired by the PathStack algorithm. It processes generalized tree patterns and tries to avoid large intermediate results.

5.2.1 Generalized tree patterns

Multiple path expressions in the FOR, LET, WHERE and RETURN clauses may have different semantics. Existing work shows that it is better to consider these expressions as a whole in terms of a generalized tree pattern (GTP). [9]

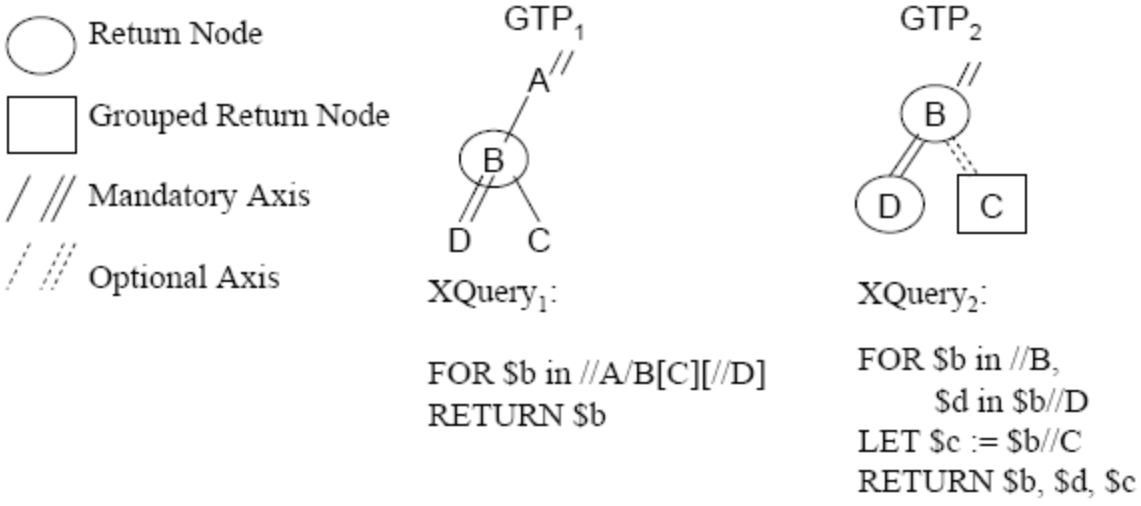


Figure 27 [10]

Figure 27 [10] shows two sample XQuery statements and their GTPs. In $XQuery_1$ node D is not a return node. Only its existence is of interest. In $XQuery_2$ node C is optional. Any matching C must be grouped together under their common ancestor element B. Returning the entire twig results is seldom necessary and may cause duplicate elimination or ordering problems. [10]

The concept of generalized twig patterns was introduced in [9]. A GTP may have solid and dotted edges which represent mandatory and optional structural relationships.

For a given GTP not all nodes are return nodes. For path expressions in the FOR clause only the last node is the return node. An example is the node B in GTP_1. For the path expression in LET or RETURN grouping of the matching elements under their common ancestor may be needed. An example is the node C in GTP_2.

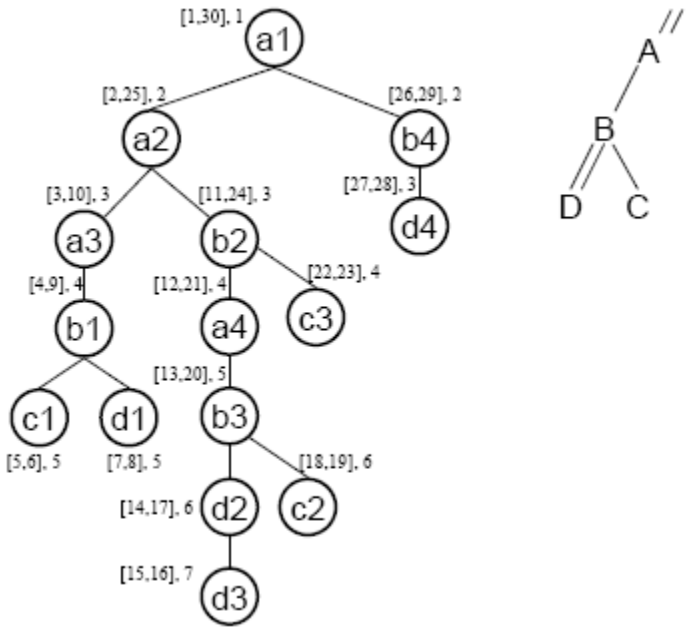


Figure 28 [10]

Now it will be described briefly how the results are being generated when there are non-return nodes in the GTP query (on example XML document from Figure 28 [10]):

1) For path query //B//D assume that B and D are both return nodes. The final matches are: (b1, d1), (b2, d2), (b2, d3), (b3, d2), (b3, d3), (b4, d4)
2) Assume D is the only return node. In this case the results should be (d1), (d2), (d3), (d4). To generate distinct paths duplicate elimination is unavoidable.
3) Consider query //A/B where B is the only return node. The results are (b1), (b2), (b3), (b4). The order is different from the order for the entire path matches (a1, b4),(a2, b2),(a3, b1), (a4, b3). Sorting becomes unavoidable in this case.

5.2.2 Hierarchical Stack Encoding

The stack encoding is used to record ancestor-descendant relationships between elements in the same query node.

For each query node N of a twig query Q a hierarchical stack HS[N] is associated. Each hierarchical stack consists of an ordered sequence of stack trees ST. A stack tree is an ordered tree where each tree node is a stack S.

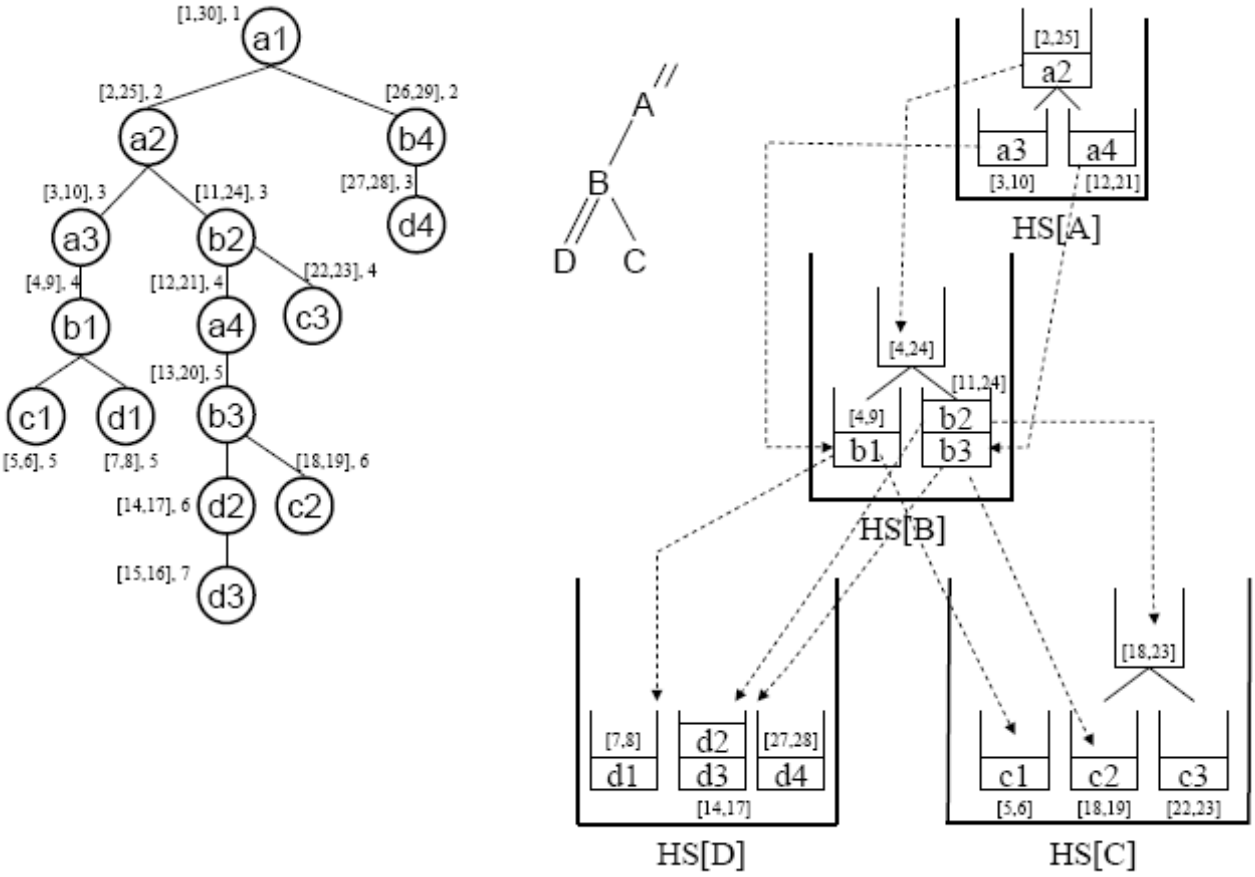


Figure 29 [10]

In Figure 29 [10] HS[A] contains one stack tree and HS[D] contains three stack trees. Each stack S contains zero or more elements. One document element is an ancestor for all elements below when in the same stack and it is also an ancestor for elements in descendant stacks. Two elements have no ancestor-descendant relationship if their corresponding stacks have no ancestor-descendant relationship. For example element a2 in HS[A] is ancestor for a3 and a4. A3 and a4 have no ancestor-descendant relationship.

To create the hierarchical structure among stacks when visiting the document in post-order a region encoding is associated to each stack. The LeftPos for a stack S is the smallest LeftPos among all elements in stack S and its descendant stacks. The RightPos for a stack S is the largest RightPos among all elements in stack S and its descendant stacks.

For example stack HS[B] has the encoding [4, 27]. A level number is not needed in the encoding.

The region encoding for a stack-tree ST is the same as the encoding of ST's root stack.

In a given hierarchical stack HS[N] its stack trees are ordered based on their RightPos. For a given stack S its child stacks are also ordered based on their RightPos.

5.2.3 Creating Hierarchical Stacks through merging

Leaves are visited before the root in post-order traversal. So the hierarchical structure is built in a bottom-up manner. For this a merge operation is needed to combine multiple stacks to a single one.

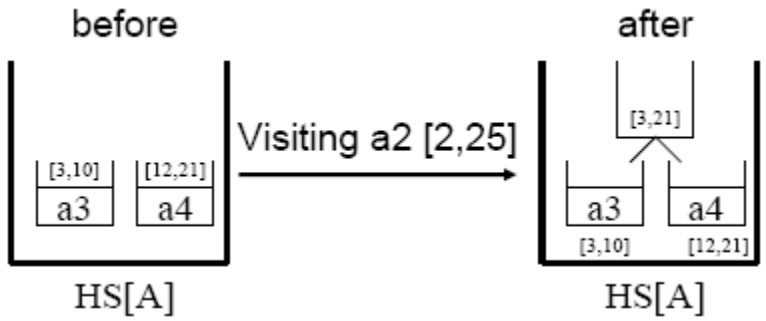


Figure 30 [10]

Figure 30 [10] shows how the stack trees in HS[A] are created.

```
Boolean merge (HierarchicalStack HS[M], docElement e,
Axis axis)

Boolean Satisfied = FALSE;
StackTreeSet STS = empty;

01 BEGIN
02   FOR each stack tree ST of HS[M]
          //Visit in descending order of ST.RightPos
03        IF ST.RightPos < e.LeftPos
04             break; //No need to keep visiting more
                        stack trees;
05        IF axis = PC AND ST.top.Level = e.Level+1
06             Satisfied = TRUE;
07             addPCEdge(e, M, ST.top);
08        ELSE IF axis = AD
09             Satisfied = TRUE;
10             addADEdge(e, M, ST.top);
11             STS = STS union ST;
12   createMergedStackTree(STS);
13   return Satisfied;
14  END
```

Figure 31

Function *createMergedStackTree* (line 12) creates a new stack and lets all stack trees in STS be its children. Line 5-10 is to process one query step. The newly or merged stack must always have the largest RightPos.

LEMMA 5.2.3.1

> *Assume that for a given document element* e*, the stack trees* ST_1*,* ST_2*, ..., and* ST_n *are merged and a new root stack* ST_{n+1} *is created. For any document element* e' *visited during the rest of the post-order document traversal, it will be either an ancestor of* all ST_1*,* ST_2*, ..., and* ST_n *or of* none *of them.*

5.2.4 Bottom-up query processing

The algorithm visits the nodes in post-order. A global stack is maintained to manage elements on the same path. When an element e is given, all elements will be popped that are not e's ancestors. Then e will be pushed onto the stack. The popped elements are in post-order. (see line 2, 3, 6)

The idea is as follows:

A given element e will be pushed into a hierarchical stack HS[E] if it satisfies the sub-twig query rooted at this query node E. Only E's child query nodes M need to be checked, because all elements in HS[M] have already satisfied the sub-twig query rooted at M. Finally the hierarchical stack structure is maintained using the merge algorithm when checking one query step or when pushing one document element into the hierarchical stack. Maintaining the hierarchical structure among stacks serves multiple purposes. First it encodes the partial/complete twig results to minimize intermediate results. Second it reduces the query processing costs and third it enables efficient enumeration. [10]

```
Procedure Twig²Stack(docElement e)

Stack docPath;
docElement currentElem;

01 BEGIN
02   WHILE docPath not empty AND docPath.top is not e's ancestor
03        currentElem = docPath.pop();
04        FOR each query node E with matching label of
               currentElem
05             MatchOneNode(currentElem, HS[E]);
06             docPath.push(e);
07 END

Procedure MatchOneNode (docElement e, HierarchicalStack HS[E])

Boolean Satisfied;

01 BEGIN
02   Satisfied = TRUE;
03   FOR each child query node M of E & Satisfied
04        Satisfied = merge(HS[M], e, axis(E_M));
05        IF Satisfied
06             merge(HS[E], e, "");
07             push (HS[E], e);
08 END
```

Figure 32 [10]

A document element e will be pushed into HS[E] if all query steps to E's child nodes M have been satisfied. Because of the post-order traversal all e's descendants must have been visited and have been pushed into HS[M] if satisfied. Checking of one query step E -> M for e can be done with merging of HS[M].

Assume n stack trees in HS[M] named ST_1 to ST_n.

First assume none of the stack trees are e's descendants. Then it can be concluded that e cannot satisfy E.

Then assume the stack trees ST_p … ST_n are e's descendants. ST_i.top is denoted as the top element of the root stack of stack tree ST_i. ST_i.top may be empty if the top stack is empty.

In case when the query step E -> M is an ancestor-descendant relationship then all elements in the stack trees satisfy this step. The results of this query step will be encoded by creating

edges e to ST_p.top, … , ST_n.top. The edge simply means that ST_i.top and the elements in the descendant stacks are e's descendants.

When the query step E -> M is a parent-child relationship only ST_p.top, … , ST_n.top might be child of e. The edges are created if the level number equals e.LevelNumber + 1.

After this query step checking phase the stack trees ST_p.top, … , ST_n.top will be merged and a new root stack is created.

Optional axis can easily be supported. An element will be pushed into the stack if all its mandatory axes are satisfied. Edges are created for optional and mandatory children.

THEOREM 5.2.4.1 [10]:

For any document element e, *it is pushed into* HS[E] *iff it satisfies the sub-twig query rooted at* E.

PROOF [10].

1) "->": The proof is straightforward based on the dynamic programming nature of the Twig^2*Stack algorithm.*
2) "<- ": If E *is a leaf query node, then any document element* e *with matching labels satisfies this query node and will be pushed into* HS[E]. *The theorem is trivially true. For a non-leaf query node* E, *we prove the theorem by contradiction.*
Assume one element e *satisfies* E *but is not in* HS[E]. *Then at least one query step* E -> M *failed when merging* HS[M]. *Since* e *satisfies* E, *there must exist one element* m *which satisfies* M *and the structural relationship between* e *and* m *satisfies the query step* E -> M. *There can be two reasons why the merging of* HS[M] *failed. They are either (i)* m *in* HS[M] *however the structural relationship between* e *and* m *is not captured through merging to satisfy the query step. Clearly,* m *must reside in one stack tree in* HS[M] *and* e *must be able to find this stack tree as its descendant - AD relationship is thus satisfied. If* m *is the child of* e, *then* m *must be at the top of one stack tree - PC relationship is thus satisfied. Hence, case (i) is not possible. (ii)* m *is not in* HS[M], *or in other words,* m *does not satisfy* M. *By applying the same reasoning, we can conclude that there must exist one* p *element that satisfies query node* P *(*P *is* M*'s child query node) but not in* HS[P]. *Eventually when reaching the leaf query node, as stated before, all the elements are satisfied and must be in the corresponding hierarchical stack. Hence, case (ii) is also not possible.*

5.2.5 Space Complexity and memory requirement

The number of path matches is in worst case exponential in terms of the size of the query $O(|D|^{|Q|})$.

THEOREM 5.2.5.1 [10]:

For a given twig query Q *and an XML document*

D, *both the space and time complexity of Twig²Stack algorithm in Figure 7 are* O(|D||B|)*, where* |B| *is the maximum of 1)* B_1 *=max(number of query nodes with the same label in* Q *) and 2)* B_2 *= max(total number of children of query nodes with the same labels in* Q*). Obviously,* |B| <= |Q|.

PROOF [10].

We show the case when the query nodes have distinct labels. There are two costs in Twig²Stack, namely, the cost for merging stacks and the cost for checking all branches of one query node. For the merge cost, assume that there are n *elements in one hierarchical stack. It is easy to show that the merge cost is* O(n)*, since once the stack trees are merged, they need not be considered for merging again. When all query nodes have distinct labels, obviously the total merge cost is* O(|D|)*. The branch checking cost is bounded by the maximum fan-out of the query nodes.*

The memory requirement of Twig²Stack is higher than in TwigStack. In the worst case Twig²Stack may keep the full document in memory. This will practically never happen, because only document elements will be stored which satisfy some part of the query twig.

5.3 Summary

The TwigStack was inspired by the PathStack algorithm. It uses the same stack encoding, but it checks every node if it is part of any solution. This check decreases the size of the intermediate results which increases the performance.

The Twig²Stack uses a much more complex stack-encoding which decreases intermediate results. It can also handle optional query nodes.

Another algorithm which is not covered in this work is the MyTwigStack [17]. It advances the TwigStack with an effective path merging scheme to gain more performance.

The TwigStackList [18] uses a read-ahead state to read the elements from the input data streams and cache them into main memory.

Another optimization is the TwigOptimal [20] which tries to reduce the cursor movements to gain more performance.

6 Indexes

All presented algorithms are using the introduced numbering scheme which helps to determine structural relationships like parent-child or ancestor-descendant. This is an effective index over all nodes in the XML repository. There are a lot of other operations in the algorithms which can be optimized by an index structure. One example is to read out all children from a node. In relational database systems there exist a lot of different index structures which are optimized for a specific use case. Nearly all structures can be adapted and used for XML databases. The rapid growth of XML databases has led to much research on this sector.

The most important index structures for the join algorithms are the B+-tree, XB-tree and the XR-tree. The B+-tree is the basic structure for the XR and the XB-tree. Each data structure has special advantages and tradeoffs for the join algorithms.

6.1 B+ Tree

The B+ tree is an enhancement of the B-tree and stores all records in the lowest level of the tree. Only keys are stored in interior blocks. It supports efficient insertion, retrieval and removal of records. Each record is identified by a key. In the case of the join algorithms the key is the numbering scheme. The tree is always balanced which is the reason for the high performance of any operation on the tree like searching.

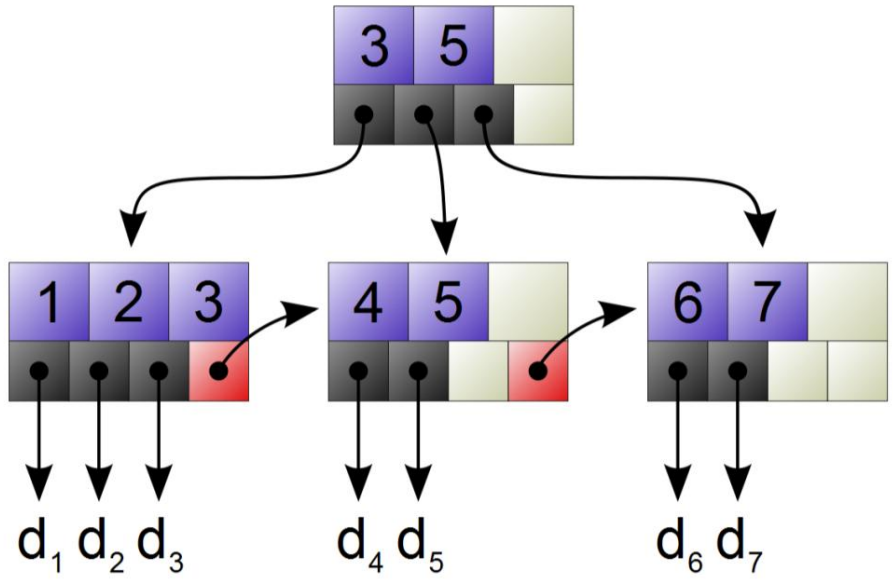


Figure 33 [5]

The linked list (colored red) allows a rapid in-order traversal.

The data is stored in a block-oriented context. When a storage system with a block size b is given the B+ tree stores a number of keys equal to a multiple of b. This is very efficient compared to a binary search tree. [5]

B+ tree is used in various file systems and database systems for block indices. For example it is used in ReiserFS or NTFS for Microsoft Windows.

The maximum number of keys in a tree node is called the order of the B+ tree. It is denoted by the variable b. The minimum number is 1 / 2 of the maximum. [5] It can be rounded up or down, but it must be done consistently throughout the tree. The minimum number restriction is lifted for the root node. There may be the case that the tree contains too few entries to fill the index.

Example: The order of a B+ tree is b. Each node has at least (b/2) + 1 keys and at most b keys. The root element has 2 to b keys.

This results that the number of keys that may be indexed in a B+ tree is a function of the order b and the height of the tree.

A B+ tree with order b and h levels has the following characteristics:

- maximum number of records stored is $n = b^h$
- minimum number of keys is $2(b / 2)^{h-1}$
- space complexity of the tree is $O(n)$

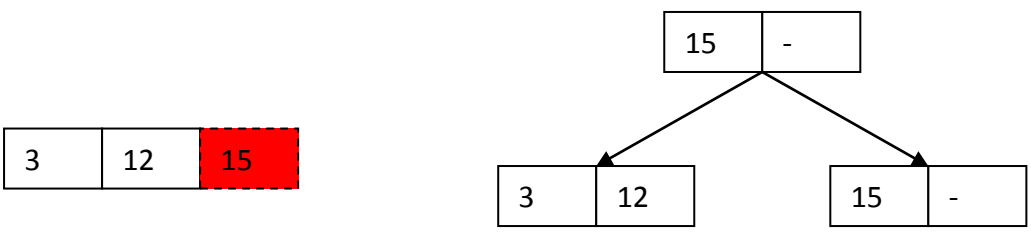
A search operation has the same complexity of $O(\log_b n)$ operations.

An insert operation requires $O(\log_b n)$ operations in the worst case. First the position for the insert has to be searched and then an overflow of the node can occur. Then a split of the node happens.

The basic B+ tree operations will be needed for the containment forest extension and will be shortly presented. These operations are the basics for every following index structure which is important for joins.

Simple insertion of 12:

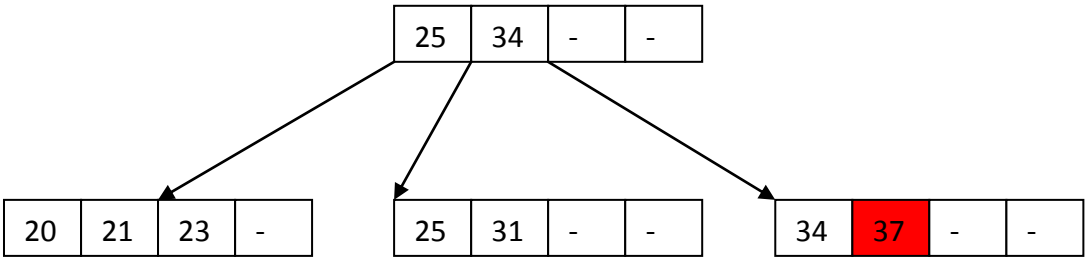
Insert of element 15 results in an overflow which causes a split:



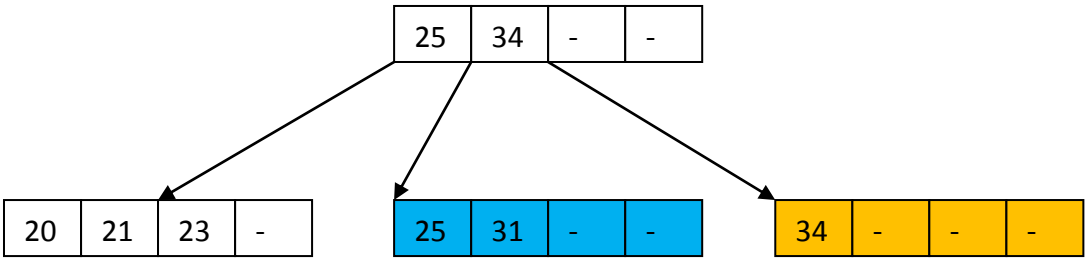
A split can also result into a new root node.

The remove operation requires $O(\log_b n)$ operations in the worst case. If the node has not enough elements after the remove it will be merged with a sibling node.

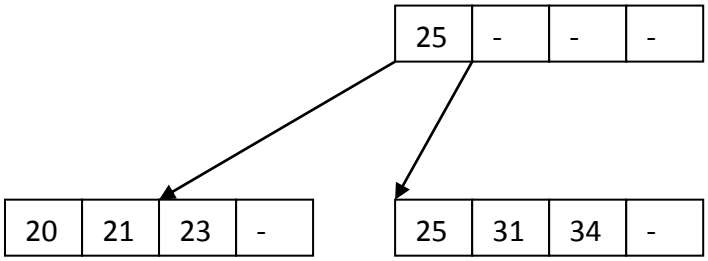
For example if 37 in this tree will be removed the node has only one key left.



This causes an under run of this node and a merge operation with its sibling has to be done.



Key 34 moves to the sibling node and the old node will be deleted.



The most important feature of the B+ tree is the ability to make efficient range queries. For this the leaf nodes are structured as a linked list. For example when searching all data nodes with key 2-6 the tree will be traversed till key 2 is found. Then a sequential scan from key 2 till key 6 collects the needed data.

For structural joins this range queries accomplish an efficient retrieval of descendants from a node. The numbering schema will be used as key.

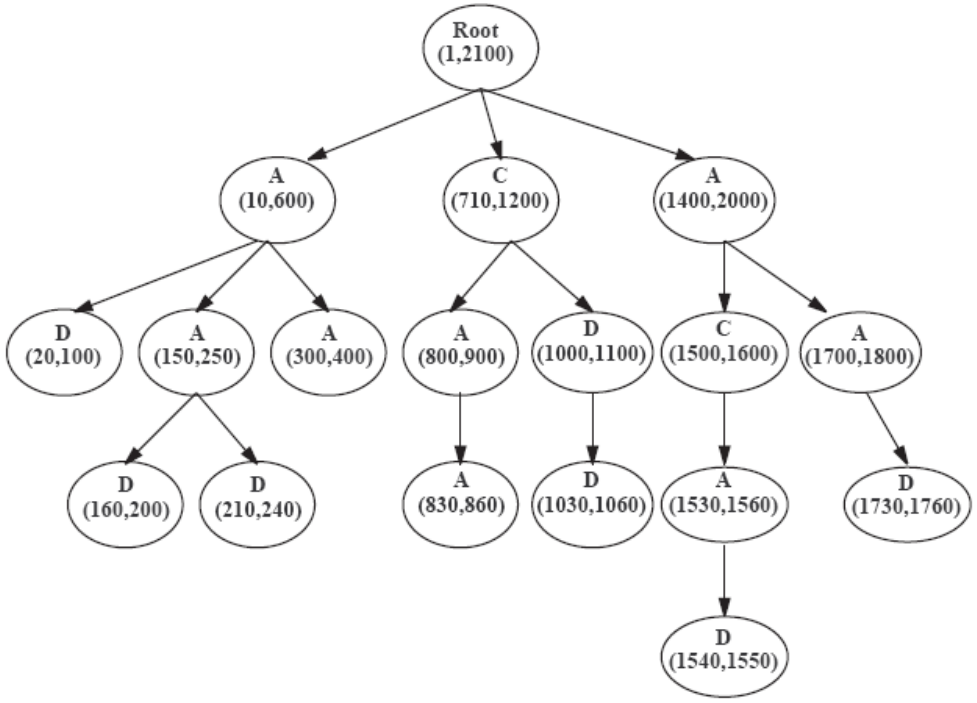


Figure 34 [6]

The graphic above shows a sample XML document. All nodes have been indexed with the introduced numbering scheme. This information can simply be indexed by a B+ tree as shown below.

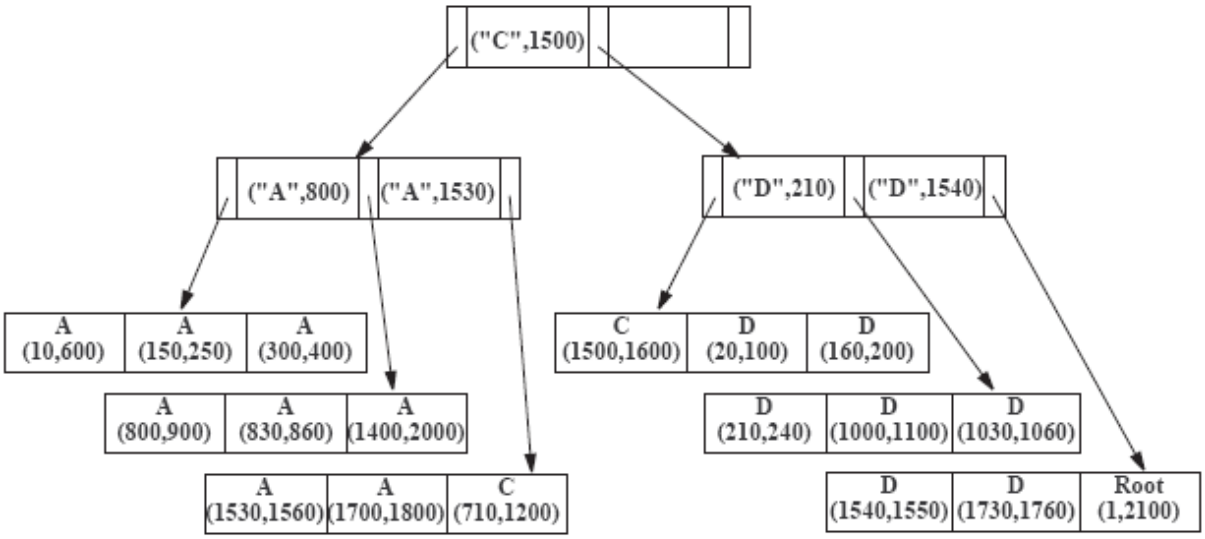


Figure 35 [6]

The pointers between sibling nodes are missing in the graphic for simplification. This index assumes the same DocumentId for all nodes.

A sequential scan of the input lists is not needed anymore. All the ancestor-descendant information can be retrieved efficiently by using the B+ tree. An example is shown in the following graphic.

Thin line segments represent the (start, end) intervals of nodes in the ancestor list and thick line segments represent the (start, end) intervals of elements in the descendant list.

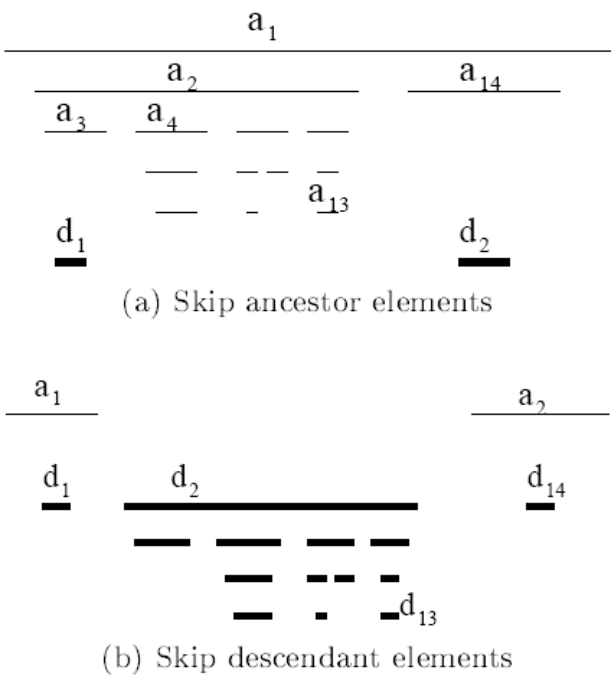


Figure 36 [6]

Figure 36 [6] a shows a scenario where ancestor elements should be skipped. For this example the Stack-Tree-Desc algorithm will run through the following steps:

1) push a_1, a_2, a_3 into the stack and join them with d_1
2) pop a_3 and a_2 from the stack
3) examine elements a_4 trough a_{13} from AList (pushing and popping each element)
4) push a_{14} into the stack and join a_{14} and a_1 with d_2

Obviously the third step is wasteful. The pushing and popping of the elements costs a lot of CPU resources.

With a B+ tree step 3 can be avoided. After a_2 is popped the algorithm directly jumps to a_{14} by using the B+ tree. a_{14} has the smallest start which is larger than the end from a_2.

Figure b shows a scenario where descendant elements should be skipped. After a_1 was joined with d_1 the Stack-Tree-Desc will scan the elements d_2 to d_{13}. Again this costs resources and can be avoided by a B+ tree. With the index the algorithm can directly jump to d_{14} which is the element from DList with the smallest start that is larger than the start from a_2.

6.2 B+ Tree with embedded containment forest

The containment forest is not directly a standalone index, but it is a powerful enhancement of the B+ tree which can gain a lot of performance.

A containment forest links elements from the same tag. Each element corresponds to a node in the structure and is linked to other elements from the same tag. The link is established via parent, first-child and right-sibling pointers.

Parent Pointer:

Given are nodes n and n_p from the same tag. Node n_p is called parent of node n when the conditions a and b are fulfilled:

a) n_p is n's ancestor in the document tree (n_p.start < n.start < n.end < n_p.end)
b) there is no other ancestor node n_a from n where n_p is the ancestor of n_a

First-child:

Given are nodes n and n_c from the same tag. Node n_c is called first-child of node n when the conditions a and b are fulfilled:

a) n_c is a child of n
b) there is no other same-tag node that is a child of n and is before n_c (node n_1 is before n_2 when n_1.end < n_2.end)

Right-sibling:

Given are nodes n and n_s from the same tag. Node n_s is called right-sibling of node n when the conditions a and b are fulfilled:

a) n and n_s have the same parent node
b) there is no same-tag node between them with the same parent (n_2 is between them when n_1.end < n_2.start and n_2.end < n_3.start)

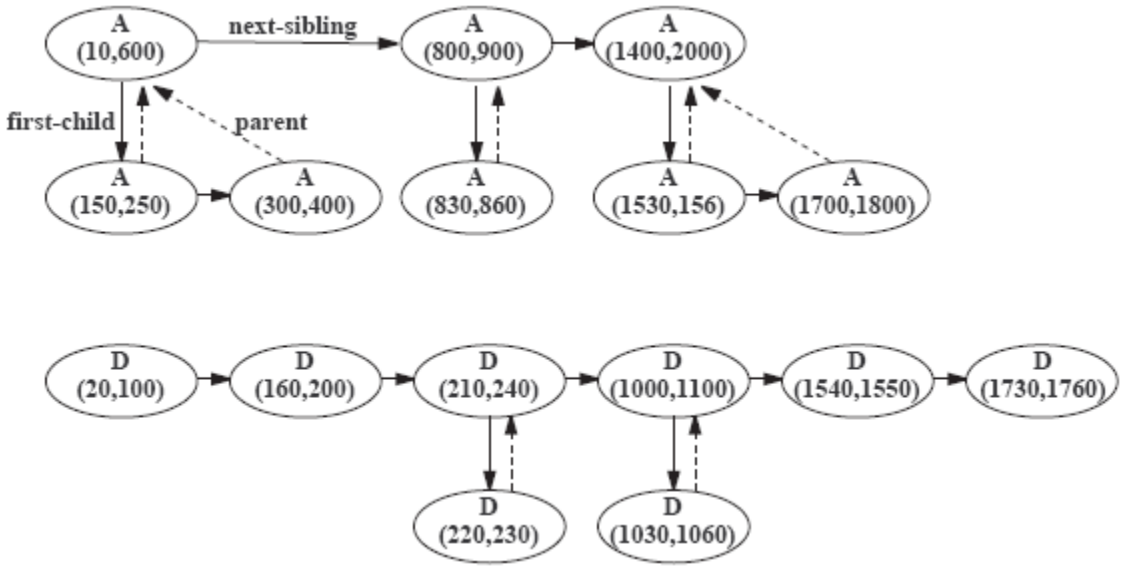


Figure 37 [6]

A containment forest has the following properties:

- The (start, end) interval of each node contains all intervals in its sub tree
- All start numbers follow a preorder traversal
- Start (end) numbers of sibling nodes are in increasing order

A containment forest for a given tag can easily be embedded in the B+ tree which indexes the tags. This will be accomplished by adding parent and next-sibling pointers among the leaf records. First-child pointers are always stored subsequently in the B+ tree. [6]

Obviously the operations of a B+ tree with an embedded containment forest must be extended too.

At each deletion and insertion the containment forest pointers have to be updated. The algorithm which handles inserts is shown below in pseudo code.

```
Algorithm Insert_B+sp (int start, Attribute attr)

01 Use the B+-tree insertion algorithm to insert a new
element (start, attr)
02 if ( leaf page overflow occours ) then
03   for ( each element i that is moved to a new page)
04        Adjust pointers pointing to i;
05   endfor
06 endif
07 Find the parent, left sibling, right sibling, and
first child of the new element and link them with it.
```

Figure 38 [6]

First the new element will be inserted into the B+ tree using start as key. Then the right-sibling and parent pointers will be adjusted among the leaf elements. Step 7 locates among the existing elements which element will be the parent, left sibling, right sibling and first child of the new element.

To locate the left sibling of the new element a the B+ tree will be used to locate element e with the largest start which is smaller than a.start. Either Element e is the parent of a or e is before a. If e is parent of element a then a does not have any left sibling. Otherwise consider the parent of e. Either e.parent is parent of or it is before a. In the first case e is left sibling of a. In the second case recursively consider the parent of e.parent and so on. If the highest ancestor of e in the containment forest is before a it becomes the left sibling of a.

The procedure of finding the left sibling also finds the parent of a.

To locate the right sibling the B+ tree is used to find the element e with the smallest start that is greater than a.end. If e.parent is the same as the parent of a then e is the right sibling of a. Otherwise a does not have a right sibling.

It is simple to determine the first-child of a. The leaf elements of the B+ tree are sorted by the StartPosition so the algorithm just has to examine the element stored right after a.

If a page overflow occurs on an insert, the B+ tree insertion algorithm will move some records to other pages. The pointers pointing to these elements have to be adjusted.

Again the algorithm, which handles delete operations, is shown in pseudo code below.

```
Algorithm Delete_B+sp ( Pointer a )
01 Locate the left sibling and the first child of a;
02 Adjust the C-forest pointers of the related elements;
03 Use the B+-tree deletion algorithm to delete element
a;
04 if ( a leaf page underflow occours ) then
05   for ( each element I that is moved to a new page )
06        Adjust pointers pointing to i;
07   endfor
08 endif
```

Figure 39 [6]

First the element a will be deleted from the containment forest. Then all related elements will be located. (parent, first-child, left sibling, right sibling)

Different from the insertion algorithm the parent and right sibling elements don't have to be located.

The parent pointers of all childrens from a will be set to a.parent. For the left sibling of a its right sibling has to be set to the first child of a. For the last child of a its right sibling pointer will be set to point to a.right. After this modifications element a can be deleted with the deletion algorithm of the B+ tree.

If a page underflow occurs its records will be moved to a sibling page. All pointers of the moved records have to be adjusted.

Analyzing the containment forest enhancement:

The depth of a node is the number of ancestors a node has.

The max-depth denotes the maximum depth of any node in the forest.

The max-span corresponds to the maximum number of children under any node.

These parameters are depending on the document characteristics. Elements with bigger intervals tend to have many children elements and create deeper subtrees. This increases the max-depth and max-span.

THEOREM 6.2.1 [6]:

The amortized insertion/deletion cost of a B+sp-tree is O(h + s + d), where h is the height of the B+-tree and s, d are the max-span and the max-depth of the embedded C-forest.

6.3 XR-tree

The XR-tree was especially designed for XML data. Again the numbering scheme is used to index a node.

Definition 6.2.1 [7]:

Given a key k *and an element with region* $E_i(s_i, e_i)$, E_i *is said to be* stabbed *by* k*, or* k stabs E_i*, if* $s_i \le k \le e_i$*. Given a set of ordered keys,* $k_j(0 \le j < n)$*, where* $k_x < k_y$ *if* $x < y$*, and an element* $E_i(s_i, e_i)$*,* E_i *is said to be* primarily stabbed *by* k_j*, or* k_j primarily stabs E_i*, if (1)* $s_i \le k_j \le e_i$*, and (2) for all* $l, l < j, k_l < s_i$*, that is,* k_j *is the smallest key that stabs* E_i*.*

Definition 6.2.2 [7]:

Given a set of ordered keys, $k_j(0 \le j < n)$*, where* $k_x < k_y$ *if* $x < y$*, and a set of elements* $E = U_i(s_i, e_i)$*, the* stab list *of a key* k_j *is the list of elements in* E *that are stabbed by* k_j *, denoted as* SL_j *or* $SL(k_j)$*. The* primary stab list *of a key* k_j *is the list of elements in* E *that are primarily stabbed by* k_j *, denoted as* PSL_j *or* $PSL(k_j)$*.*

The stabbing and primary stabbing relationships are shown in the Figure below.

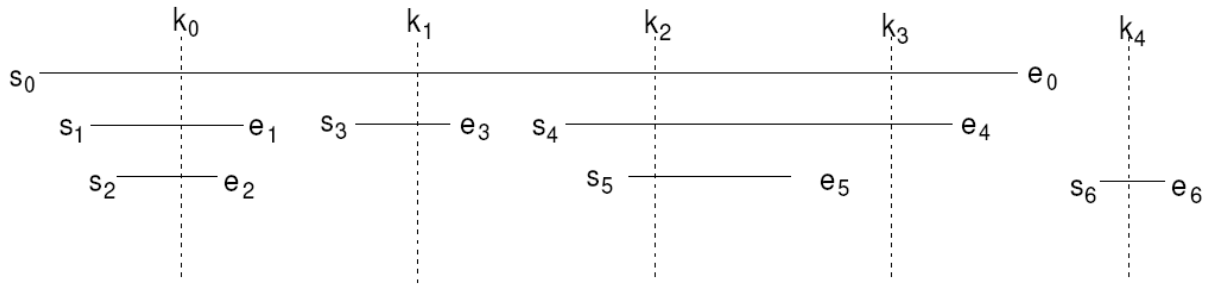


Figure 40 [7]

Given are the keys $k_0 < k_1 < k_2 < k_3 < k_4$ and seven regions (si, ei). The list of regions stabbed by kj are:

- $SL_0 = \{(s_0, e_0), (s_1, e_1), (s_2, e_2)\}$
- $SL_1 = \{(s_0, e_0), (s_3, e_3)\}$
- $SL_2 = \{(s_0, e_0), (s_4, e_4), (s_5, e_5)\}$
- $SL_3 = \{(s_0, e_0), (s_4, e_4)\}$
- $SL_4 = \{(s_6, e_6)\}$

The primary stab lists are:

- $PSL_0 = \{(s_0, e_0), (s_1, e_1), (s_2, e_2)\}$
- $PSL_1 = \{(s_3, e_3)\}$
- $PSL_3 = 0$
- $PSL_4 = \{(s_6, e_6)\}$

Strict ancestor-descendant relationship holds between each pair of neighboring elements in a primary stab list.

In case of k_0 and PSL_0 (s_0, e_0) is an ancestor of (s_1, e_1) which is an ancestor of (s_2, e_2). The first element of PSL_k is the ancestor of all other elements in the list and its region covers all other regions in PSL_k.

Definition 6.2.3 [7]:

> *The start and end positions,* ps_j, pe_j, *of key* k_j *with primary stab list* PSL_j *are defined the start and end positions of the first element of* PSL_j *when* PSL_j != ∅, *and* (nil, nil) *if* PSL_j = ∅.

In case of the example shown in Figure 40 [7] we get the results:

- $(ps_0, pe_0) = (s_0, e_0)$
- $(ps_1, pe_1) = (s_3, e_3)$
- $(ps_2, pe_2) = (s_4, e_4)$
- $(ps_3, pe_3) = (nil, nil)$
- $(ps_4, pe_4) = (s_6, e_6)$

Those shown definitions where summarized by [7] to a final structure description:

Definition 6.2.4 [7]:

An XR-tree for a set of region-encoded XML elements is a tree with the following properties:

1. An XR-tree is a balanced tree.
2. An internal node contains m key entries in the form of (k_i, ps_i, pe_i), with $k_0 < k_1 < \cdots < k_{m-1}$, and $d \le m \le 2d$, where d is the degree of the XR-tree.
3. An internal node with m keys also contains m + 1 pointers p_j , $(0 \le j \le m)$, pointing to the nodes in the next level of the tree, such that all keys in the node pointed by p_i are less than k_i, and all keys in the node pointed by p_i+1 are greater than or equal to k_i, respectively.
4. An internal node n is associated with a stab list, SL(n), which holds all elements E_i, such that E_i is stabbed by at least one key in n but not stabbed by any key of any ancestor of n. Each element in SL(n) is in the form of (s, e, pointer), where (s, e) is the region of the element and pointer points to the data entry of the element.

5. SL_j , PSL_j for the set of all keys k_j in an internal node n are defined on the list SL(n) by Definition 6.2.2. Each pair of (ps_j, pe_j) of k_j is defined by Definition 6.2.3.
6. Leaf nodes contain element entries, in the form of (s, e, InStabList, pointer), where (s, e) is the region of the element, and s is the index key. InStabList is a flag indicating whether the element is included in any stab list of internal nodes, and pointer points to the data entry of the element.
7. Leaf nodes are linked from left to right.

With these definitions an XR-tree is a B+ tree with a complex index key entry and extra stab lists associated with its internal nodes. This supports efficient structural joins which will be shown later.

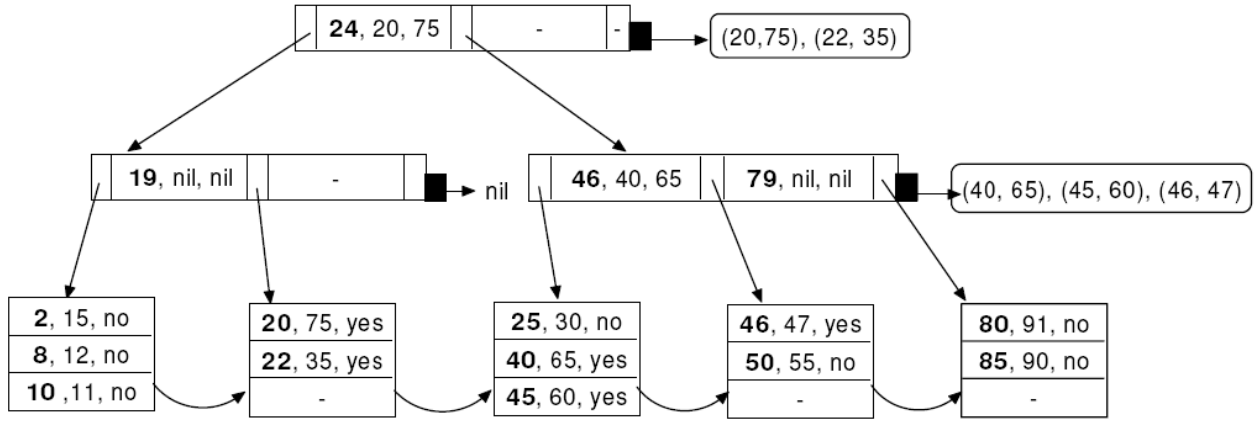


Figure 41[7]

Figure 41[7] shows an example. The left internal node has an empty stab list. Region (20, 75) is stabbed by key 46. It is not part of the stab list of the right node, because it is already stabbed by key 24 in the root. This shows that a region will always be included in the topmost nodes stab list. For key 19 $ps_{(19)}$ and $pe_{(19)}$ are set to nil, because $PSL_{(19)} = 0$. In case of key 46 $PSL_{(46)} = \{(40, 65), (45, 60), (46, 47)\}$. $ps_{(46)} = 40$ and $pe_{(46)} = 65$.

Indexing keys must not be start positions of element nodes. It is better to choose the key to minimize the size of the stab lists. If 80 will be taken as key (which is the start of element (80, 91)) instead of 79 the stab list would contain one more element.

6.3.1 Performance Analysis

Due to the stab lists maintaining the tree causes a bit more overhead.

6.3.1.1 Space complexity

The sizes of the stab lists may be a concern. Each element region can only be included at most once in one stab list. This results that the elements in the stab lists will not exceed the elements which are indexed in the tree. Each index key can at most stab h_d elements. h_d is the maximum number of nesting element nodes indexed.

The maximum number of pages for a stab list is [7]: $S_{max} = \frac{hd * BI * fmax}{Bs * fmin}$

B_l is the maximum number of entries in an internal node. B_s is the maximum number of tuples a stab list page can hold. When f_{min} is 0,5 and f_{max} is 1 then $S_{max} = 2h_d$ pages. Normally the number of pages is between zero and a few pages.

6.3.1.2 Access cost complexity

If an element set is highly nested the stab list of an internal node can span a few pages to tens of pages. In such extreme cases the access costs can be a concern.

For example the PSL_i often needs to be located for key k_j when searching or updating in the XR-tree. It is a problem when a lot of pages have to be scanned to locate PSL_i. This problem can be solved with a ps directory page that maps each ps_j to the location of PSL_j.

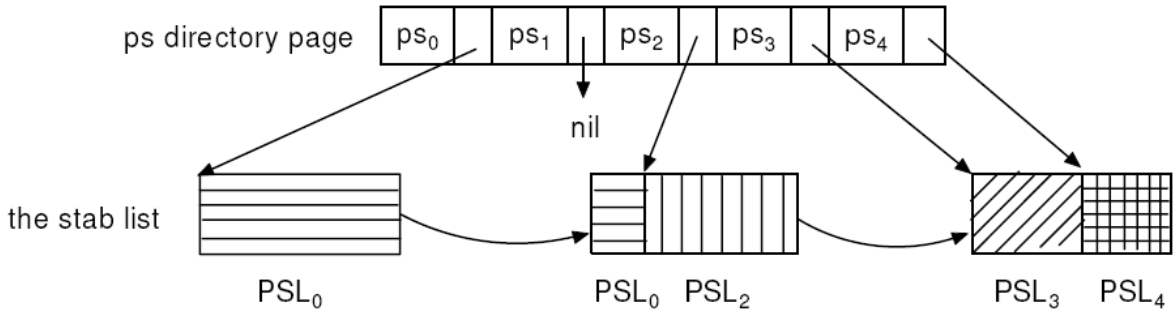


Figure 42 [7]

The ps directory page contains one entry for each index key. An entry has the format (ps_j, pointer) where ps_j is the ps field of key k_j. The pointer points to the head of PSL_j in the stab list. The pointer is set to nil when $PSL_j = 0$. This is the case for ps_1 in Figure 42 [7].

With this extension it takes only 1 or 2 disk I/O's to locate PSL_j[7].

6.3.2 Inserting

The inserting is equivalent to the B+ tree.

Algorithm Insert:
Input: A new element, E = (s, e, pointer), to be inserted.

I1 **[Find a leaf page for insertion]**
Navigate down to a proper leaf page for insertion. Insert E into the stab list of the highest internal node that stabs it, if any.

I2 **[Insert E into the leaf page L]**

I21 If L has room for another element, insert E and return.

I22 Otherwise, split L by moving second half entries to a new leaf page L_{new}. Insert E into the correct leaf page. Give up a new key entry k', pointer'), together with its StabSet'.

I3 **[Insert** (k', pointer', StabSet') **into internal node I]**

I31 If I has enough room, insert the new entry and its StabSet' into I.

I32 Otherwise, split I by moving the second half entries, together with their primary stab lists, to the new node I_{new}. Insert the new entry and its StabSet' into the proper internal node. Give up a new key entry(k', pointer'), together with its StabSet'. Repeat step I3, or go to I4 if I is the root node.

I4 **[Grow the XR-tree taller]**
If the node split propagation caused the root to split, create a new root whose children are the two resulting nodes.

Figure 43 [7]

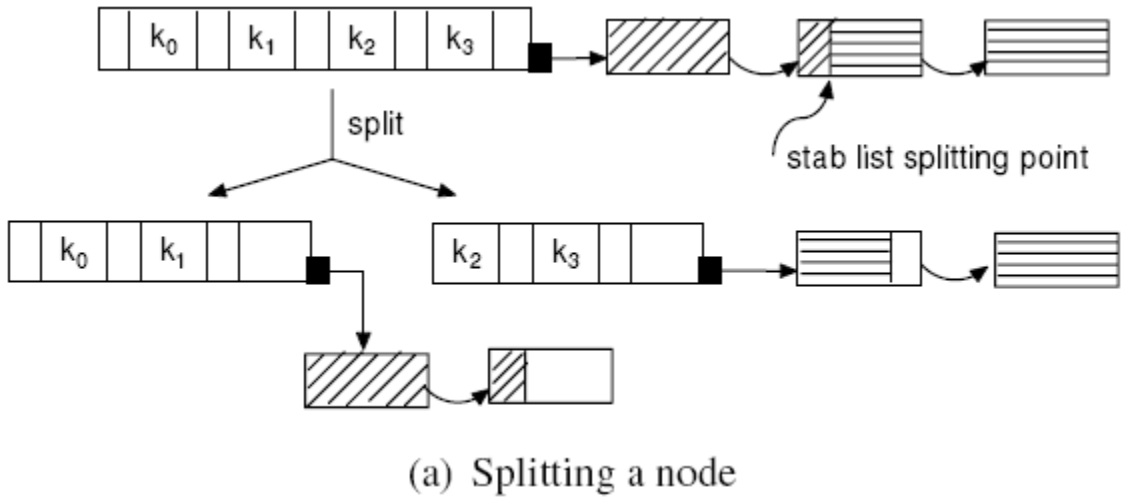
Step I1 navigates down to a leaf page and inserts element E. It gets also inserted into the stab list of the highest internal node that stabs it. If there is one the flag InStabList will be set to true.

In case of an overflow step I22 and I32 also split the stab lists. The split costs are independent of the size of the stab list, because for the split only access to the page which holds the splitting point is needed. No other pages of the stab list have to be touched.

After a split a new key k' is proposed to the upper level and the set of elements stabbed by k' will be proposed to the upper level. It will be denoted as StabSet'. See Figure 44 [7]b

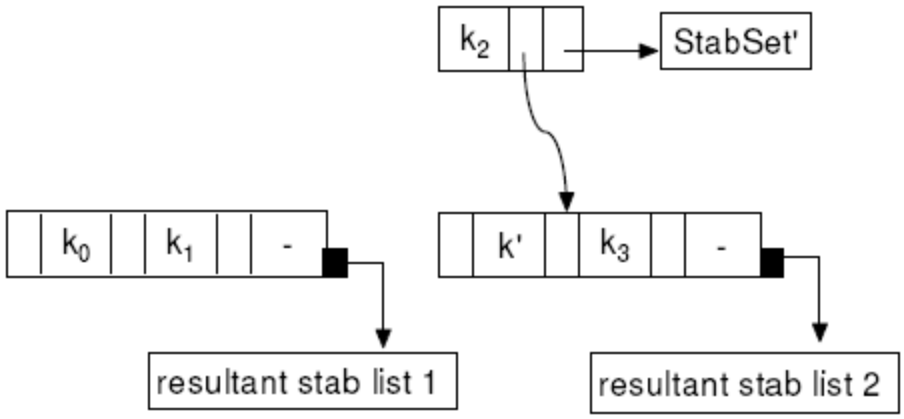
where k' is inserted after k_2 which is selected for being the key to be given up. StabSet' contains all elements from SL(I) and SL(I_{new}) that are stabbed by k'.

If the split occurs on a leaf page all elements from L and L_{new} are retrieved which are newly stabbed (and InStabList flag is false) and set the flag to true.



(a) Splitting a node

Figure 44 [7]



(b) Giving up a new key with StabSet'

Figure 45 [7]

The basic insertion has the complexity of $O(\log_F N)$ which is the same of the B+ tree. This does not include the maintenance of the stab lists.

The amortized I/O costs for the stab list maintenance is given by the worst case total I/O costs for inserting N elements and divide the result by N.

Worst case is given when all elements in leaf pages are stabbed by internal nodes. Each element at height h is given up from the lower level h-1. This will be done in step I22 and I32

in Figure 43 [7]. It is assumed that every element is given up from a lower level. An element at height h will be given up h times from leaf to its current position during a split. Cdp denotes the costs for one move. (E.g. moving from level h-1 to h) A stabbed element at height h causes C_{DP}*h displacement costs. The amortized I/O cost C_{DP} for each insertion is divided by the total number of insertions.

$\mathrm{Cdp} = \sum_{h=1}^{h=H} Nh * Cdp$*h/N [7]

N_h is the number of total stabbed elements at height h. Each internal node stabs at most h_d*BI*f elements. The maximum total number of stabbed elements with height>1 is $h_d/B_L*B_I*f^2$ * N.

$h_d/B_L*B_I*f^2 << 1$ implies that most stabbed elements are at height 1. In the worst case all elements are stabbed in height 1. This result to $\mathrm{Cdp} \approx \mathrm{N1} * \mathrm{Cdp/N} \leq \mathrm{Cdp}$[7]

THEOREM 6.2.2.1 [7]:

The amortized I/O cost for inserting a new element into an XR-tree is $O(\log_F N + C_{DP})$, where N is the number of elements indexed, F is the fanout of the XR-tree, C_{DP} is the cost for one displacement of a stabbed element, or the cost for deleting an element from a stab list and then inserting it into another stab list.

6.3.3 Deletion

Similar to deletion in a B+ tree redistribution or merging occurs in case of an underflow. Again here the maintenance of stab lists is interesting.

```
Algorithm Deletion:
Input: An element, E = (s, e, pointer), to be deleted.

D1 [Locate E in leaf page]
     Locate the leaf page containing E.
     Delete E from the internal node I if its stab list
     contains E.

D2 [Delete E from leaf page L]

     D21 Delete E from L. If L remains at least half full,
          return.

     D22 Otherwise, let S be a sibling of L. If S has extra
          entries, redistribute entries between L and S and
          update their parent entry.

     D23 Otherwise, merge S and L. Propagate the deletion up
          the tree with the key of their parent entry.

D3 [Delete the entry from an internal node I]

     D31 Suppose entry j is to be deleted. Delete entry j
          from I and "reinsert" elements in SL(I) that are
          no longer stabbed by I. If I remains at least half
          full, return.

     D32 Otherwise, let S be a sibling of I. If S has extra
          entries, redistribute entries between L and S.
          Update their parent entry properly and return.

     D33 Otherwise, merge L and S, and their stab lists.
          Propagate the deletion up the tree with the key of
          their parent entry.

D4 [Shorten the tree]
     If the root has only one child after the deletion, make
     the child as the new root.
```

Figure 46 [7]

Deleting an index entry will cause that some elements in SL(I) will not be stabbed any longer. For each element E' it has to be reinserted at the highest internal node that stabs it. If no such interval node exists the flag InStabList will be set to false.

After redistribution between two internal nodes the entry with key k in the parent node P has to be updated with key k'. SL(k') has to be removed from the two internal nodes and inserted into SL(P). Then some elements in SL(P) might not be stabbed any longer by P when k is replaced with k'. This will be handled similar to the entry deletion case. [7] Redistribution between two leaf pages works similar.

In case of merging two internal nodes from I to its left sibling S also SL(I) and SL(S) will be merged. For this SL(I) and SL(S) will be simply linked.

The I/O costs also include additional costs for maintaining the stab lists like in the insert operation.

THEOREM 6.2.3.1 [7]:

> *The amortized I/O cost for deleting an element from an XR-tree is* $O(\log_F N + 3 \cdot C_{DP})$*, where* N *is the number of elements indexed,* F *is the fanout of the XR-tree,* C_{DP} *is the cost for the displacement of an element from one stab list to another.*

6.4 XB-Tree

The XB-tree is a variant of the B+ tree which was especially designed for indexing positional representations in the format (DocumentId, LeftPosition, RightPosition). The structure in Figure 47 [8] assumes that the DocumentId of all nodes is equal. To extend the XB-tree to index nodes with different DocumentIds is straightforward.

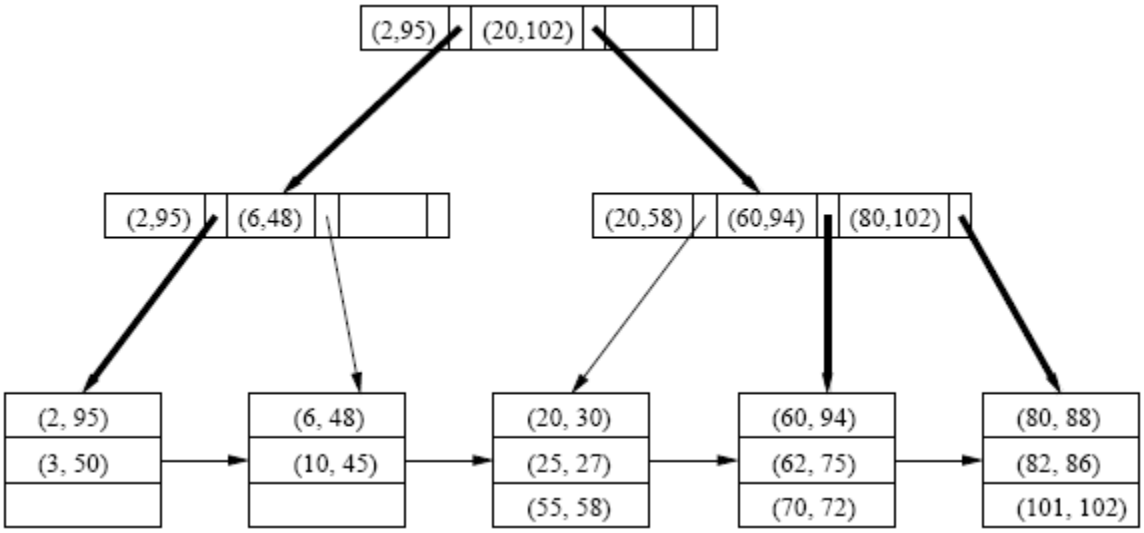


Figure 47 [8]

All nodes in the leaf pages are sorted by their LeftPosition (L). This is similar to a B+ tree on the L values. The difference between an XB-tree and a B+ tree lies in the data maintained at internal nodes. Each node N in an internal page consists of a bounding segment [N.L, N.R]. (L denoted the LeftPosition value and R the RightPosition value) Also a pointer to its child page N.page is maintained. This page contains all nodes with bounding segments included in [N.L, N.R]. Bounding segments of nodes in internal pages might partially overlap, but they are in increasing order. Each page P has a pointer to its parent page P.parent and an integer P.parentIndex which is the index of the node in P.parent which points back to P.

The maintenance is also quite similar to a B+ tree using the L value as key. The only difference is that R has to be propagated above.

To process the findAncestors queries the XB-tree starts from the root node and goes down to the leaf nodes. Only paths that are with intervals that can cover the given interval are searched. Data entries in leaf nodes which can cover the given interval are output as results. In the worst case the findAncestors operation covers the whole tree.

The findAncestors operation is optimal when the intervals in the index nodes are the minimum bounding intervals of the intervals in their child nodes. This is because every search path, except for the last search path, will yield at least one ancestor that contains the given descendant element. [8]

Definition (Valid Path) [8]:

> *A root-to-leaf search path in*
> *an XB-tree for a findAncestors operation is a valid path*
> *if the leaf node of the path contains at least one ancestor*
> *element interval that covers the given descendant interval.*

As example Figure 47 [8] is searched for element (90,92). The two root-to-leaf search paths (2; 95) → (2; 95) → leaf and (20; 102) → (60; 94) → leaf are valid paths, because they lead to leaf nodes that contain the searched data entries. Search path (20; 102) → (80; 102) → leaf is not a valid path.

THEOREM 6.4.1 [8]:

> *Let the intervals of the index nodes in an XB-tree be the minimum bounding intervals. Given an element interval, every search path of the findAncestors operation must be a valid path except for the last search path.*

An index entry consists of an interval and a child pointer. Assume Ki and Kj to be two consecutive intervals in an index node. Let Ci be a child node that is pointed by Ki like in Figure 48 [8] shown.

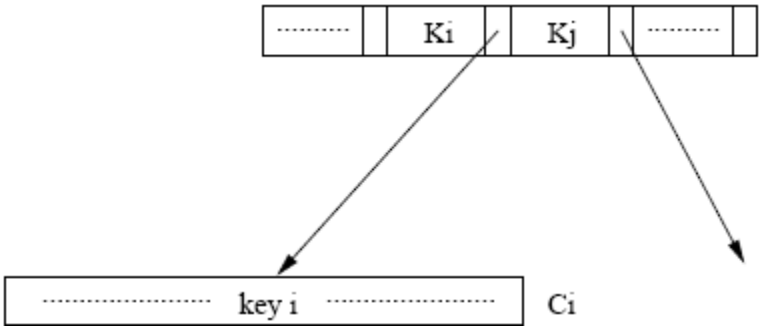


Figure 48 [8]

Suppose the intervals Ki and Kj overlap and the given descendant Id lies in the overlap. The path Ki → Ci is then not the last path. Because this is an XB-tree with minimum bounding intervals there must exist a key key_i in child node Ci that $key_i.e = K_i.e$. Consider $K_i.e > I_d.e$ and $key_i.s < Kj.s < I_d.s$ then $key_i.s < I_d.s < I_d.e < key_i.e$ implies that C_i has at least the interval key_i which contains I_d. [8] The same reasoning happens when traversing down the XB-tree. It can be guaranteed that every search path except the last one returns at least one ancestor element that contains I_d.

6.5 Summary

Index structures speed up operations like the search for ancestors or descendants of a node. Various structures have been developed. Classical structures like the B+-tree perform very well, but also new structures like the XR-tree have been proposed.

The B+ tree, XR and XB-tree are used in combination with the numbering schema. There are also indexing techniques which do not use the numbering schema. They create a structural summary of the XML document in the form of a labeled graph. [21] The idea is to preserve all paths while having fewer nodes and edges. [6]

The XB-tree is a variation of the B+-tree and is the most robust solution. Certainly there were much more index structures proposed. Another index structure which is not covered in this work is for example the Ctree [4].

7 Index based join algorithms

Processing of nodes which are not part of the solution decreases the join performance. The usage of index structures should help to avoid such nodes. This is illustrated in Figure 50 [6]. With such an index structure it is very efficient to find ancestors and descendants of a node.

This section shows some examples with the StackTree algorithm which uses the B+, XR and XB-tree to speed up the join operation. Figure 56 [8] shows how the different index structures process an ancestor and descendant search.

One example of the TwigStack shows the usage of an XB-tree to gain more performance.

The most important index operations are the search for ancestors and the search for descendants.

Benchmarking results can be found in [6] and are not covered in this work. The XB-tree seems to be the most robust solution.

7.1 Anc_Des_B+

```
Algorithm Anc_Des_B+(List A, List D)
01 Let a, d be the first elements of A and D;
02 while (not at the end of A or D) do
03   if( a is an ancestor of d) then
04        Locate all elements in A that are ancestors
          of d and push them into the stack;
05        Let a be the last element pushed;
06        Output d as a descendant of all elements in
          stack;
07        Let d be the next element in D;
08   else if (a.end < d.start) then
09        Pop all stack elements which are before d;
10        Let l be the last element popped;
11        Let a be the element in A (locate using B+-tree)
          having the smallest start that is larger than
          l.end;
12   else
13        Output d as decendant of all elements in stack;
14        if ( ancestor stack is empty ) then
15             Let d be the element in D (locate using B+-
               tree) having the smallest start that is
               larger than a.start;
16        else
17             Let d be the next element in D;
18        end if
19   end if
20 endwhile
```

Figure 49 [6]

The leaf pages in each B+ tree are linked together. Due to this the elements can be viewed as a sorted list. Variable a and d denote the first elements from AList and DList. The algorithm moves a and d forward and joining them until one of the lists is empty.

During runtime the algorithm keeps a stack of ancestor elements.

In Steps 11 and 15 use the B+ tree to skip elements from the ancestor and descendant list. To avoid unnecessary B+ tree accesses in practice, first it will be checked if the next ancestor element is on the same page p as the previous ancestor.

Step 14 shows that the stack has to be empty when the algorithm wants to skip elements in the descendant list. Otherwise it can happen that elements will be skipped which could be joined with an ancestor element on the stack. This will lead to an error.

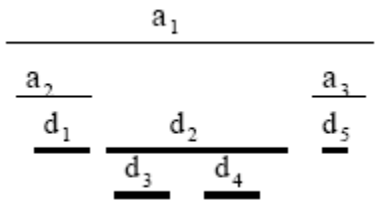


Figure 50 [6]

Figure 50 [6] shows this situation. The algorithm is currently checking ancestor a_3 against descendant d_2 while a_1 is on the stack. a_3 is after d_2 so the algorithm goes to step 13. Without step 14 it would have continued to step 15 and skip descendants d_3 and d_4. This would fail to join d_3 and d_4 with a_1.

7.2 Anc_Des_B+ using sibling pointers

The Anc_Des_B+ algorithm also works with B+ trees which are enhanced with a containment forest.

One difference appears in step 11 that finds the element a_{new} which has the smallest start larger than a.end. Due to the containment forest the relationship between a_{new} and a is as follows:

- If a has a right-sibling then a_{new}=a.sibling
- If a does not have a right-sibling, but a.parent has a sibling then a_{new} = a.parent.sibling.

This will go up in the tree until a sibling is found. If a and no ancestor of a has a right-sibling the algorithm finishes, because no other ancestor element needs to be examined.

At step 11 all ancestors of a are in the stack. Each element has now a right-sibling pointer which makes it possible to the address of a_{new} is identified directly without any extra I/O. This improves the algorithm, because the B+ tree traversal is avoided in this case.

In a plain B+ tree even if all pages are in memory the algorithm still has to search for the new ancestor element. Due to the sibling pointers this can be avoided.

7.3 Stack-based Structural Joins with XR-trees

To process the join operation two basic operations are needed:

- FindDescendants: searches all descendants for a given element E_a in an element set indexed by an XR-tree
- FindAncestors: searches all ancestors for a given element E_d in an element set indexed by an XR-tree

7.3.1 Searching for descendants

To find all descendants of a given element (s_a, e_a) is to find all elements E_i such that s_a > E_i.start < e_a. This can be accomplished by a simple range query over the start position of the elements. There is no need to access the stab lists.

```
Algorithm 3 FindDescendants

Description: find all descendant elements of EA = (sa, ea)
               in XR-tree T.

01 node := T.root;
02 while node is not a leaf page do
03   find the largest key ki in node, such that ki ≤ sa;
04   if found, let node := ki.rightChild; otherwise, let
     node := k0.leftChild;
05 end while

06 stop := FALSE;
07 while not stop do
08   for all entries Ei in node do
09        if sa < Ei.start < ea, output Ei;
10        if Ei.start > ea, let stop := TRUE;
11   end for
12   node := node.next;
13 end while
```

Figure 51 [7]

It is obvious that the algorithm retrieves all descendant elements for the given element.

THEOREM 7.3.1.1 [7]:

The operation FindDescendants over an XRtree
can be evaluated with optimal worst case I/O cost:
$O(\log_F N + R/B)$, where N is the number of elements indexed,
F is the fanout of the XR-tree, R is the output size, B
is the average number of element entries in each leaf page.

7.3.2 Searching for ancestors

A search for ancestor elements (s_d, e_d) means to find all elements E_i such that E_i.start < s_d < E_i.end. In case of nodes indexed by an XR-tree all nodes stabbed by s_d are searched. Elements stabbed by s_d could be scattered in any leaf page left to the leaf page on the search path of s_d. A sequential search over the leaf pages could be too costly.

Instead during the navigation from the root to the leaf page all stab lists of internal nodes are searched for elements stabbed by s_d.

After reaching the leaf page all elements are output which are not included in the stab lists of internal nodes. This is shown in the algorithm below.

Algorithm 4 FindAncestors

Description: Find all ancestors of ED = (s_d, e_d) in XR-tree T.

S0 Let node be the root of T;
S1 Search non-leaf pages

While node is not a leaf page **do**
S11 Retrieve all elements stabbed by s_d in its stab list, by calling *SearchStabList* (Algorithm 5).

S12 Find the largest key k_i, such that $k_i \leq s_d$.

S13 If found, let node be k_i.rightChild; otherwise, let node be the left child of the first index entry.

S2 [Search within the leaf page]
Search from the first element of node to output elements that are stabbed by sd but with InStabList flags being no, until an element whose start position is greater than sd is encountered.

Figure 52 [7]

The algorithm which searches elements in a stab list is shown below.

```
Algorithm 5 SearchStabList

Description: Search the stab list of an internal
node I for all elements stabbed by sd.

01 let ki be the key in I, such that ki ≤ sd <
     ki+1;
02 for c = i + 1 to 0 do
03   if psc _= nil and psc < sd < pec then
04        Scan PSLc and output the scanned elements
          until an element not stabbed by sd is
          encountered;
05   end if
06 end for
```

Figure 53 [7]

Assume s_d falls in $[k_i, k_{i+1})$. S_d cannot stab any element in PSL_j where j>i+1, because such elements have their start values larger than k_{i+1}. Only PSL_c has to be checked where c <=i+1. If s_d stabs an element in a PSL it stabs it ancestors in the PSL.

When searching a PSL the algorithm just scans from the beginning and stops when the current element is not stabbed by s_d. (line 4)

The I/O efficiency of SearchStabList is guaranteed because a stab list is only accessed when it contains at least one element which is stabbed. With the ps directory page it is possible to access the PSL of any key in 1-2 disk I/O's.

THEOREM 7.3.2.1 [7]:

The operation FindAncestors takes O(logF N + R) worst case I/O cost, where N is the number of elements indexed, F is the fanout of the XR-tree, R is the output size.

The correctness of FindAncestors is assured by the following lemma.

LEMMA 7.3.2.1 [7]:

Let T be an XR-tree of height H, ps be a query point. Let $I_{H-1} \rightarrow I_{H-2} \rightarrow \cdots \rightarrow I_1 \rightarrow L_0$ be the path for ps to navigate from I_{H-1}, the root of T, down to a leaf page, L_0. Let R be the set of elements stabbed by ps in T. Then, for each element $E \in R$, it must appear in L0 or the stab list of some I_i, where $i \in \{H-1, H-2, \cdots, 1\}$.

7.3.3 Structural joins on XR-tree indexed data

Input lists A and D are both indexed with an XR-tree. The join can be processed like a merge-join in database systems, because the leaf pages are sorted by the start positions. Different

from the typical merge-join algorithm the presented algorithm can skip elements which are not needed in the join. For every element it is possible, to retrieve its ancestors and descendants with the two basic operations described before. Elements which are no ancestors or descendants will not be touched. Again a stack will be used to store ancestors during its execution.

```
Algorithm 6 Stack-based Structural Joins with XR-trees

Input:A is the ancestor set and D is the descendant set.

01 CurA := First(A);
02 CurD := First(D);
03 stack := Ø;

04 while CurA != EndOf(A) and CurD != EndOf(D) do
05  if stack != Ø then
06        pop all elements that are not ancestors of CurD;
07  end if

08  if CurA.start < CurD.start then
09    Ad := FindAncestors(A, CurD.start);
10    for each aj ∈ Ad, if aj not ∈ stack, push it on the
      stack;
11    output pairs (a ∈ stack, CurD);
12    CurA := first element in A whose start > CurD.start;
13    CurD := next element in D after CurD;
14  else
15   if stack != Ø then
16        output pairs (a ∈ stack, CurD);
17        CurD := next element in D after CurD;
18   else
19        CurD := first element in D whose start >
          CurA.start;
20   end if
21 end if
22 end while
```

Figure 54 [7]

Pointers CurA and CurD always point to the current elements in A and D. Initially they are set to the first element in the lists. (line 1-2)

Like mentioned above a stack is maintained. Every element on the stack is a descendant of the elements below it. All elements in the stack are ancestors of CurD' where CurD' is the element before CurD in D. The stack is initially empty. (line 3)

At each iteration of the loop the algorithm tries to skip elements without matches based on current positions of CurA and CurD. If CurA is before CurD it tries to skip ancestors based on CurD, otherwise it skips descendants based on CurA. This is repeated until one of the lists is exhausted.

The algorithm in details:

The stack keeps the ancestors of an element CurD' preceding CurD. Some top elements of the stack may not be ancestors of CurD, because CurD.start > CurD'.start. Those elements cannot be ancestors of any element after CurD and they will be popped from stack. (line 5-7)

In case where CurA is before CurD is coped with from line 9 to 13. All ancestors of CurD are retrieved and pushed onto the stack. All matched pairs for CurD will be output and CurD and CurA forwarded. Only ancestors which are not already onto the stack are pushed. (line 9-10)

Line 15-19 deals with the case where CurA is behind CurD which means that no elements will be skipped when the stack is not empty. If the stack is not empty CurD will be preceded by one. Otherwise CurD is moved to the element right after CurA. (line 19)

7.4 Anc-Desc Structural Join

A generic index based algorithm was proposed by [8] which can make use of a B+, XR and a XB-tree.

```
Algorithm 1 Ancestor-Descendant Structural Join

Input: Lists A and D

Output: All matching (a_i; d_j), such that ai covers dj

01 a = A.first
02 d = D.first
03 stack = 0;

04 while !eof(D) and !(eof(A) and isEmpty(stack))

05   pop all a_i from stack, such that a_i can't cover d
06   let al be the last element (if any) popped

07   next = Max(a_l.e; a.s)

08   if stack = 0 and d.s < a.s then
09        d = first di in D, such that di:s > a:s
10   else
11        if di:s > a:s then
12             push findAncestors(Ta; d; next) into
               stack
13             a = first an in A, such that a_n.s > d.s
14        output (a_i, d) for all a_i element stack
15        d = D.getnext()
```

Figure 55 [8]

Also this algorithm takes two input lists with ancestors and descendant elements which are ordered by their starting positions. The input lists are scanned with help of the indexes. A stack is also used to store ancestor elements for later use.

Initially the stack is empty and the cursors are set to point to the first element in the lists. (lines 1-3)

In each iteration the algorithm checks the current descendant for ancestors. If no ancestors are found d points to the next element whose start is greater than the start point of the current ancestor element. (line 8-9)

Otherwise all ancestor elements for the current descendants which are not already in the stack are retrieved. This will be done by calling the function findAncestors. The results are pushed onto the stack. (line 12)

Then pointer a is updated to point to the next element which likely has descendant elements. (line 13)

The new elements which are pushed onto the stack together with the ancestors which already were on the stack yield all ancestors for the current descendant. (line 14)

Finally d is set to the next descendant element in D.

The algorithm terminates if:

- List D is exhausted
- List A is exhausted and the stack is empty

7.4.1 Searching for ancestors

The algorithm has to execute findAncestors (D_i) and findAncestors (D_{i+1}) consecutively. D_i and D_{i+1} are the descendant intervals.

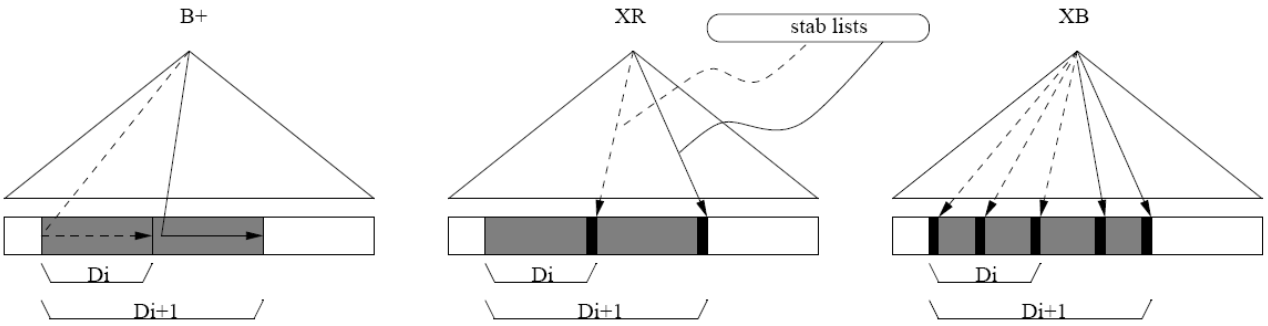


Figure 56 [8]

The figure above illustrates how each index structure handles the searches. The grey areas labeled D_i and D_{i+1} denote the interval ranges of the ancestor elements in the leaf pages that contain D_i and D_{i+1}. The search paths for the two searches are denoted by dashed and solid lines.

The B+ tree supports consecutive findAncestors efficiently, because the scan for the second search starts from the leaf page where the first search ends.

The XR-tree cannot distinguish the ancestor elements which have been used in the first search when the second one is performed. This is because the majority of the matching ancestor nodes are obtained from the stab lists.

The XB-tree can support consecutive queries efficiently, because it stores the interval information into the index nodes.

7.5 TwigStack XB

The TwigStack XB is an extended version of the TwigStack algorithm which uses an XB-tree to gain more performance. It maintains a pointer act = (actPage, actIndex) to the actIndex'th node in page actPage of the XB-tree.

There are two operations that affect this pointer:

advance: If act = (actPage, actIndex) does not point to the last node in the current page actIndex will be advanced. Otherwise act will be replaced with (actPage.parent, actPage.parentIndex) and recursively advanced.

drilldown: If actPage is not a leaf page and N is the actIndex'th node in actPage it will be replaced with (N.page, 0) that it points to the first node in N.p.

Initially act is set to (rootPage, 0). It points to the first node in the root page of the XB-tree. The traversal is finished when act points to the last node in rootPage and gets advanced.

```
Algorithm TwigStackXB(q)

01 while !end(q)
02   q_act = getNext(q)
(03) if (isPlainValue(T_qact))
04        if (!isRoot(q_act))
05             cleanStack(parent(q_act), next(q_act))
06   if(isRoot(q_act) || !empty(S_parent(q_act)))
07        cleanStack(qact, next(qact))
08        moveStreamToStack(Tqact,Sqact , pointer to
               top (S_parent(q_act) ) )

09        if (isLeaf(q_act))
10             showSolutionsWithBlocking(S_qact,1)
11             pop(S_qact)

12        else advance(T_qact)
(13) else if (!isRoot(q_act) && empty(S_parent(q_act))&&
          nextL(Tparent (q_act)) > nextR(T_qact))

(14)      advance (T_qact) // Not part of a solution
(15) else // Might have a child in some solution
(16)      drillDown (T_qact)

// Phase 2
17 mergeAllPathSolutions()

Function getNext(q)
01 if (isLeaf(q)) return q
02 for q_i in children(q)
03   n_i = getNext(q_i)
(04) if (q_i != n_i || !isPlainValue(T_ni) ) return n_i
05 n_min = minarg_ni nextL(T_ni)
06 n_max : maxarg_ni nextL(T_ni)
07 while (nextR(T_q) <nextL(Tn_max))
08   advance(T_q)
09 if (nextL(T_q) <nextL(T_nmin) ) return q
10   else return n_min

Procedure cleanStack(S, actL)
01 while (!empty(S) && (topR(S) < actL))
02   pop(S)
```

Figure 57 [3]

The changes of TwigStack to TwigStack XB are indicated with line numbers in brackets. The function isPlainValue returns true if the actual pointer in the XB-tree is pointing to a leaf node.

THEOREM 7.5.1 [3]

Given a query twig pattern q and an XML database D, Algorithm TwigStackXB *correctly returns all answers for q on D.*

7.6 Summary

This section shows how index structures can be used to speed up the join operation. The StackTree has been modified to work with a B+ tree and a XR-tree. These structures avoid unnecessary processing of elements which are not part of the final solution. A detailed speed comparison can be found in [7].

State of the art is the TwigStackXB algorithm which uses a XB-tree.

8 Implementation and experimental results

In this section the TreeMerge, StackTree and the PathStack algorithm will be tested on a given test dataset. An important difference between these algorithms is that the TreeMerge and the StackTree algorithm have to split a query into binary subsets and merge the results afterwards. The PathStack is able to process the query at once which avoids the merging phase.

This section tries to show how the different strategies of the algorithms increased the performance. The simplest algorithm is the TreeMerge join algorithm. It uses no stack to cache nodes and simply uses two nested loops to compare the ancestor and the descendant nodes.

An optimization is represented by the StackTree algorithm. It stores ancestors in a stack to gain more performance and avoid the skipping of elements which are not participating in the join. It will be shown that this simple enhancement brings a huge performance gain.

The PathStack uses more stacks to store intermediate results. Every query node has its own stack to store ancestor nodes and encode the results. It is obvious that the stack-encoding causes more overhead, but this will be amortized by not needing a merge phase.

8.1 Dataset

For the test data i chose the book example from [1] and extended it to run some test queries. I decided to write my own generator to be more flexible and generate the numbering schema information directly into the XML tags.

```
<author level="3" leftPos="8" rightPos="12">
     <name level="4" leftPos="9" rightPos="11">
          Peter1
     </name>
</author>
```

Figure 58

Especially I wanted to simulate different worst case situations in every query to see how each algorithm behaves and how the runtime was affected.

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT title (#PCDATA)>
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT illustrator ((name))>
<!ELEMENT chapters ((chapter+))>
<!ELEMENT chapter ((title, subtitle?))>
<!ELEMENT books ((book+))>
<!ELEMENT book (((illustrator, chapters?) | (author,
                        illustrator, chapters)))>
<!ELEMENT author ((name))>
```

Figure 59

Figure 59 shows the DTD for the test set. 5000 book entries have been generated with various chapters. Details about the generation can be found in the implementation section.

8.2 Queries

The queries were chosen to create various scenarios which should increase the runtimes of the algorithms.

8.2.1 Query 1: //chapters/chapter

This query handles a lot of nodes since every book with chapters can have from 5 up to 20 chapters. Especially the TreeMerge algorithm would have to skip a lot of descendent elements (chapter).

8.2.2 Query 2: //book/title

This is a simple query which has not that many elements. It should be a test for the PathStack algorithm which is more complex and will show how the overhead with the stacks influences the runtime.

8.2.3 Query 3: //book/subtitle

Query 3 returns no results because a book does not have a subtitle. Only chapters can have a subtitle. This should test the algorithms how they behave with elements they cannot match.

8.2.4 Query 4: //title/chapter

This query also does not return any elements. It has a big number of nodes which have to be tested.

8.2.5 Query 5: //book/chapters//subtitle

This query has to be split into book/chapters and chapters//subtitle. Book/chapters will generate much more results which are not part of the final result.

8.2.6 Query 6: //book//chapters//chapter//title

This query causes very much overhead, because it has to be spit into book//chapters, chapters//chapter and chapter//title. Afterwards the results have to be merged which increases the runtime of the TreeMerge and the StackTree. Three merge operations are needed to get the final results.

8.2.7 Query 7: //book/chapters/chapter/title

This query is equal to Query 6, but has only parent-child relationships. I wanted to test if a parent-child relationship causes a different runtime behavior.

8.3 Implementation

8.3.1 Node

Class node is a container class which keeps all information about the numbering schema like the positions or the level of the node.

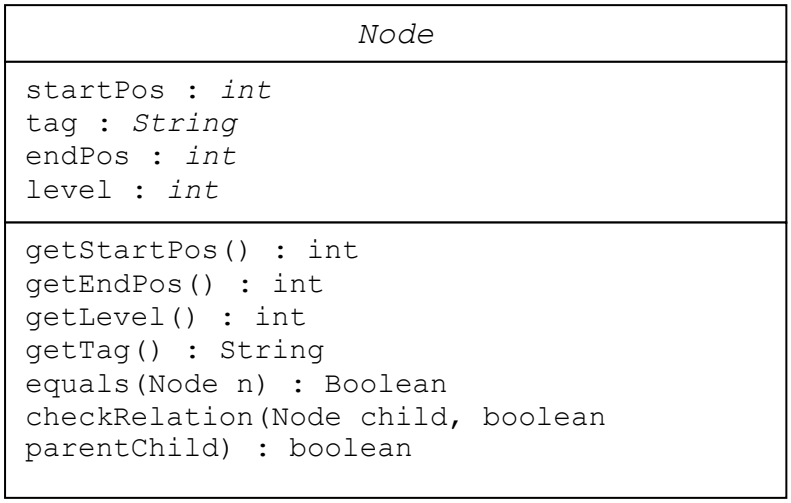


Figure 60

8.3.2 Pair

The TreeMerge and the StackTree generate Pair classes as results.

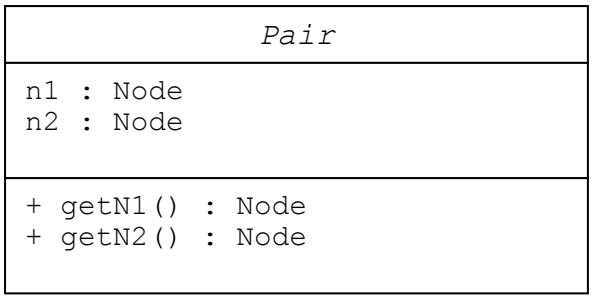


Figure 61

8.3.3 Tuple

The PathStack algorithm organizes all nodes with stacks of Tuples. Attribute idx keeps the index of the parent stack.

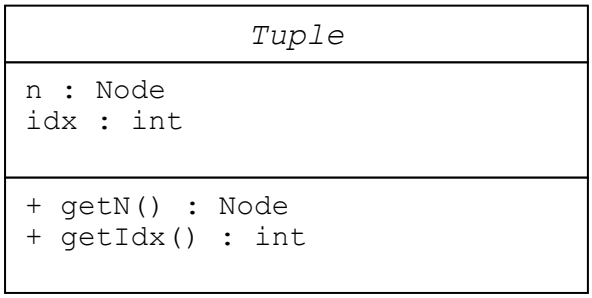


Figure 62

8.3.4 PStack

For every query node the PathStack has a PStack which is needed to encode the nodes. Each PStack has a parent PStack except for the PStack of the root query node.

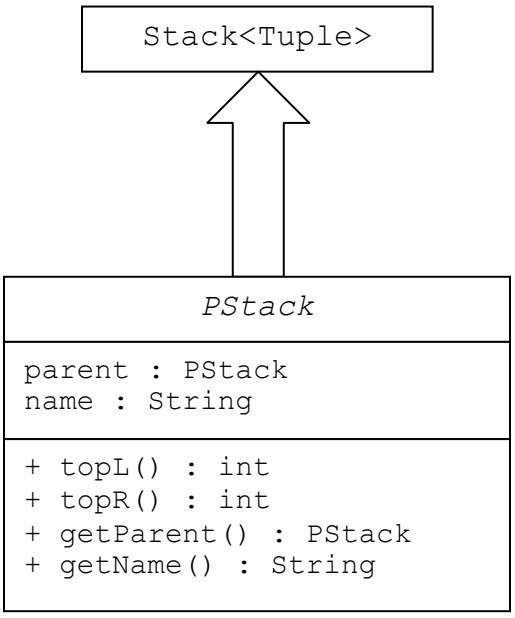


Figure 63

8.3.5 Query

The TreeMerge and the StackTree algorithm can only handle one relationship at one time. The PathStack can resolve the full query and needs all relationships of the query nodes during the join. The class query contains all information about the query node relationships.

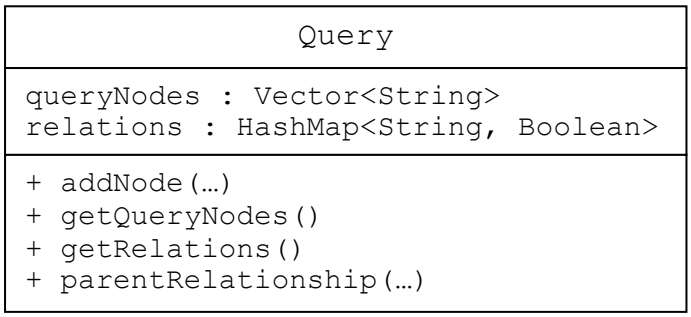


Figure 64

8.3.6 Generator

For fast data generation XMark [11] is a good choice, but to gain more flexibility in the data generation i decided to write my own data generator. I randomized specific tags like the author tag in a book to create special result sets for my queries. It is a straight forward implementation which also generates the numbering schema for each tag.

8.3.7 Parser

With the SAX parser I was able to quickly parse the needed nodes from the example repository. It returns a vector of nodes sorted by the start positions of the tags. I generated the numbering schema directly into my XML repository. Otherwise I would have to iterate through the XML tree to generate the numbering schema.

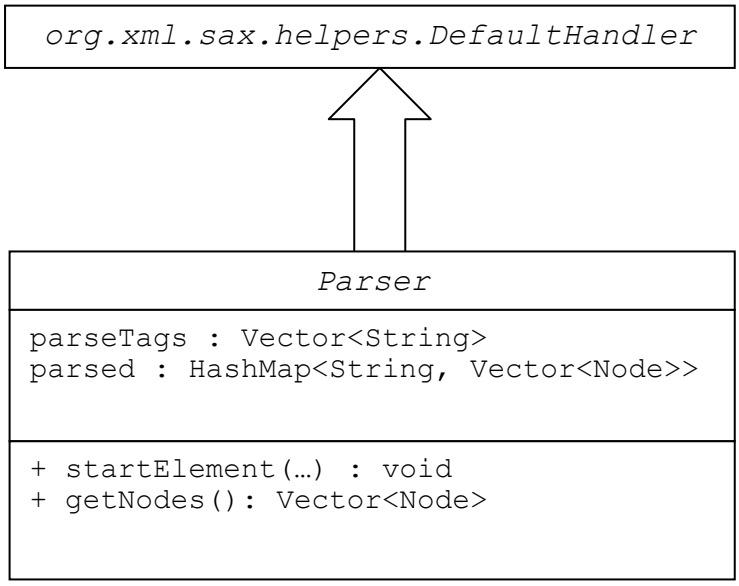


Figure 65

8.3.8 TreeMerge

This algorithm takes the two sorted lists of ancestors and descendants as input. These lists can be directly used from the parser. Each node contains its numbering information which is needed to check ancestor-descendant und parent-child relationships.

It iterates over the ancestor nodes and skips all descendants which are not joinable with the current ancestor. This skipping is very costly which can be seen very well in Query 2. It has to skip a lot of title nodes which increases the runtime.

Operation merge and mergeIntermediate merge the results from the sub queries in nested loop manner.

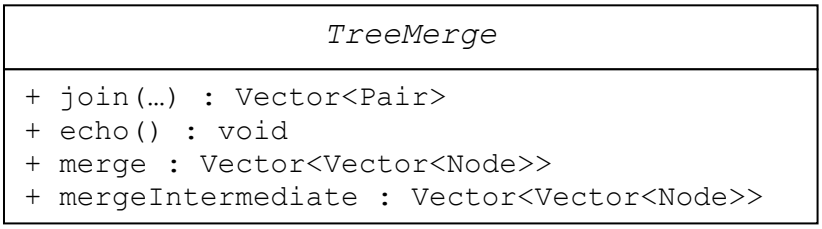


Figure 66

8.3.9 StackTree

For the stack tree a stack has to be maintained to cache ancestor nodes. In my test it shows very good results. The stack increases performance and does not cause too much overhead.

The algorithm terminates when the stack is empty and no ancestor elements in the input list are left or no descendant elements where left.

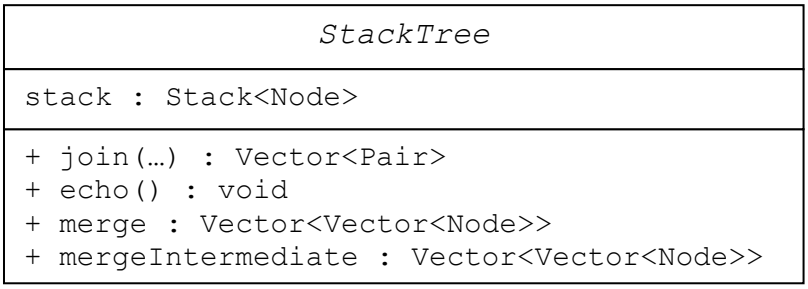


Figure 67

8.3.10 PathStack

PathStack expects for every query node a stream which returns the nodes in sorted manner. For my purposes it was better to lead all data into memory, because it is much faster than the latency of my hard drive.

The PathStack uses more than one stack to encode its intermediate results. For each query node one stack is maintained which is linked to the stack of the parent query node. I decided to manage all stacks and streams in a hash map.

PathStack
Query query; streams : HashMap<String, Vector<Node>> stacks : HashMap<String, PStack>
+ join(…) : Vector<Pair> + echo() : void + merge : Vector<Vector<Node>> + mergeIntermediate : Vector<Vector<Node>> - end : Boolean - getMinSource : Node - getSubtreeNodes : Vector<String> - showSolutions : Vector<Vector<Node>>

8.4 Results

Every algorithm handled a test repository with 5000 books. The StackTree showed very good performance on queries with two nodes. It is also faster than the PathStack, because it has less overhead.

PathStack outperforms the other two algorithms when the query has more than three nodes. The merge of the intermediate results from the TreeMerge and StackTree is extremely costly.

Especially in query 1 the TreeMerge algorithm has to skip a lot of chapter nodes and causes an increased runtime. Here the StackTree performs best, because it has less overhead than the PathStack.

Query 2 handles fewer nodes than query 1. Here the difference between StackTree and PathStack is much higher.

Query 5, 6 and 7 have more than 2 query nodes. Here the merge of the intermediate results has a huge impact on the runtime of the TreeMerge and StackTree. The difference to the PathStack is very high which shows that for more complex queries PathStack performs best.

Query 6 and 7 has nearly the same runtime. The only difference between these two queries is the relationship between the query nodes. Query 6 resolves only parent-child relationships and query 7 only ancestor-descendant relationships.

Query	1	2	3	4	5	6	7
TreeMerge [ms]	20703	5047	1094	33765	19641	161640	159688
StackTree [ms]	297	140	47	250	10656	130234	124813
PathStack [ms]	1703	1875	281	3312	343	3532	3484

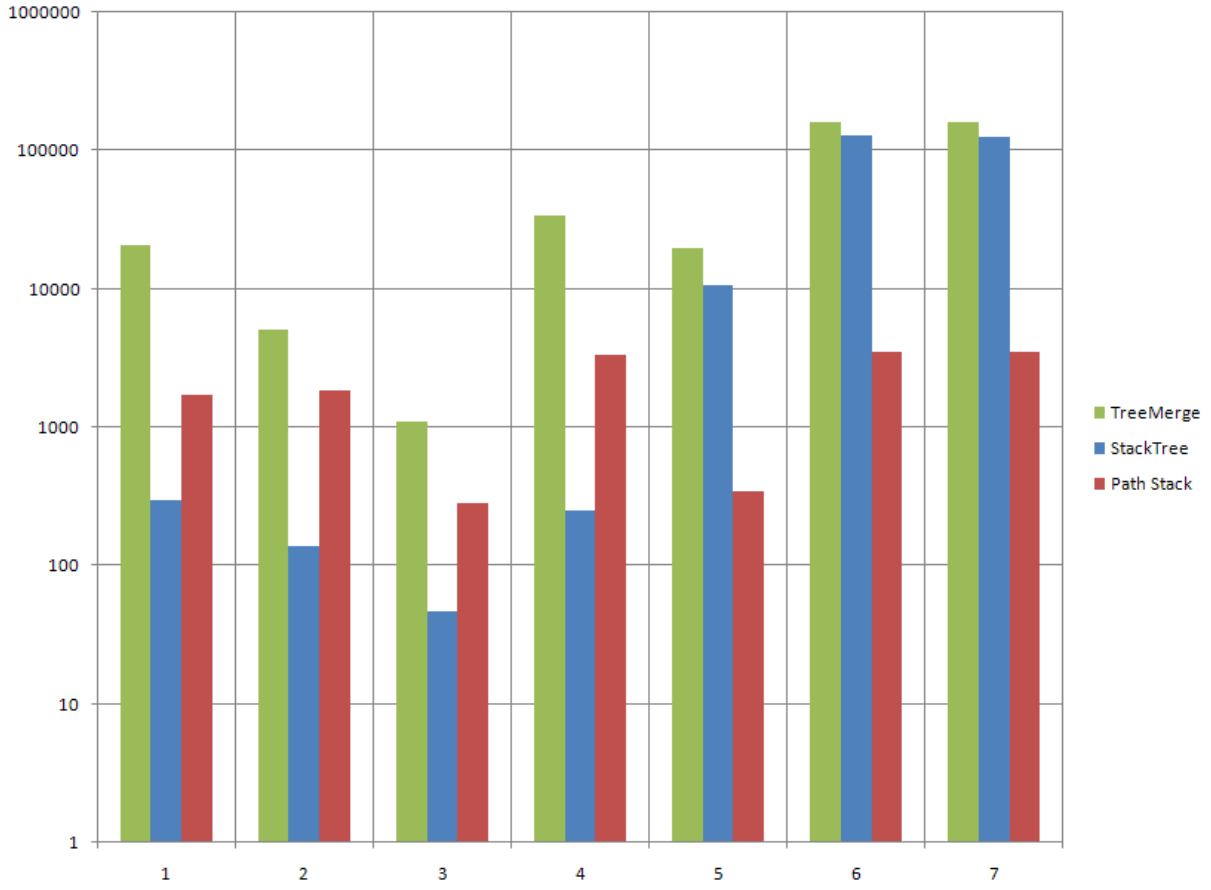


Figure 68

8.5 Optimal merge of intermediate results

The tests have shown that the merge process of intermediate results is very costly. Many intermediate results may not be part of the final result. Especially when a query is longer the intermediate results which are useless for the final solution can get very large.

To reduce the costs of the merge phase where the results get merged it would be possible to use nodes from a previous processing step. This would reduce the size of the intermediate result and increase the runtime performance. The simple optimization would speed up the MergeTree and the StackTree if the query twig has more than two nodes. The merge phase can be completely avoided.

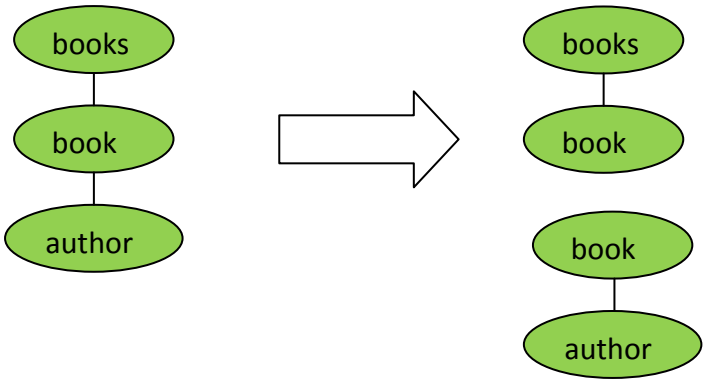


Figure 69

Figure 69 shows an example query twig which would be decomposed into books/book and book/author. This causes that the results of books/book and book/author has to be merged.

The optimization would first compute the results to books/book and pass this results to the next processing step book/author. The book nodes from the first subquery will be used as ancestors for the next subquery.

This is certainly also possible from the other side. That first book/author will be computed and the results given up to provide the descendants for the next subquery. A query planer could help to determine the optimal processing queue of the subqueries.

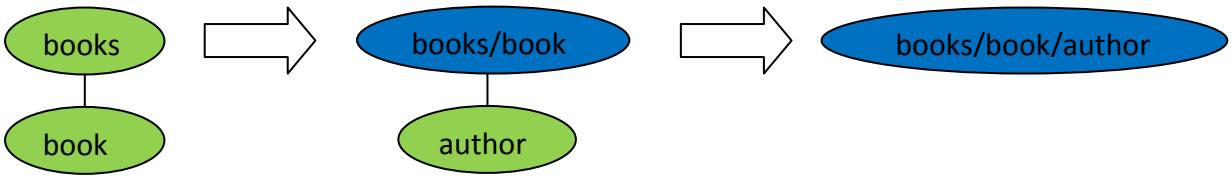


Figure 70

9 Conclusion

Many structural join algorithms have been proposed with different ideas and strategies. The key of every algorithm is the numbering schema, which helps to determine structural relationships efficiently.

Some algorithms process a query at once and others have to split it to process only two query nodes at a time. State of the art is the Twig^{2}Stack which is able to process GTPs and can for example also handle optional query nodes.

My tests have shown that the StackTree delivers very good results if the query has only two nodes. It has not very much overhead and the stack is a good strategy to increase the performance. The disadvantage is that if the query has more than two nodes the intermediate results have to be merged. This happens in nested loop manner and costs a lot of processing time. A possible optimization is shown in 8.5. Some detailed tests with real world data would be very interesting and show how the optimization works.

Future work could be a query planner which chooses the algorithm depending on its statistics over the data. Like in relational database systems where a query planner decides to use an index or a sequential table scan. A combination of different algorithms could use the advantage of every algorithm. One example could be the usage of the StackTree algorithm if the query has only two nodes.

10 References

[1] Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M., Srivastava, D., and Wu, Y. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the IEEE International Conference on Data Engineering,* 141--152.

[2] Zhang, C., Naughton, J., DeWitt, D., Luo, Q., and Lohman, G. 2001. On supporting containment queries in relational database management systems. *SIGMOD Rec.* 30, 2 (Jun. 2001), 425-436. DOI= http://doi.acm.org/10.1145/376284.375722

[3] Bruno, N., Koudas, N., and Srivastava, D. 2002. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data* (Madison, Wisconsin, June 03 - 06, 2002). SIGMOD '02. ACM, New York, NY, 310-321. DOI= http://doi.acm.org/10.1145/564691.564727

[4] Zou, Q., Liu, S., and Chu, W. W. 2004. Ctree: a compact tree for indexing XML data. In *Proceedings of the 6th Annual ACM international Workshop on Web information and Data Management* (Washington DC, USA, November 12 - 13, 2004). WIDM '04. ACM, New York, NY, 39-46. DOI= http://doi.acm.org/10.1145/1031453.1031462

[5] Wikipedia B+ tree: http://en.wikipedia.org/wiki/B%2B_tree Last visited: 29.5.2008

[6] Chien, S., Vagena, Z., Zhang, D., Tsotras, V. J., and Zaniolo, C. 2002. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th international Conference on Very Large Data Bases* (Hong Kong, China, August 20 - 23, 2002). Very Large Data Bases. VLDB Endowment, 263-274.

[7] Haifeng, J., Lu, H., Wei, W. & Ooi, B. C. 2003. *XR-Tree: Indexing xml data for efficient structural joins*. In IEEE International Conference on Data Engineering, 253--264.

[8] Li, H., Lee, M. L., Hsu, W., and Chen, C. 2004. An evaluation of XML indexes for structural join. *SIGMOD Rec.* 33, 3 (Sep. 2004), 28-33.
DOI=http://doi.acm.org/10.1145/1031570.1031576

[9] Chen, Z., Jagadish, H. V., Lakshmanan, L. V., and Paparizos, S. 2003. From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In *Proceedings of the 29th international Conference on Very Large Data Bases - Volume 29* (Berlin, Germany, September 09 - 12, 2003). J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Very Large Data Bases(Vldb) Series. VLDB Endowment, 237-248.

[10] Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D., and Candan, K. S. 2006. Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd international Conference on Very Large Data Bases* (Seoul, Korea, September 12 - 15, 2006). U. Dayal, K. Whang, D. Lomet, G. Alonso, G. Lohman, M. Kersten, S. K. Cha, and Y. Kim, Eds. Very Large Data Bases. VLDB Endowment, 283-294.

[11] XMark. Available from http://monetdb.cwi.nl/XML/ Last visited: 09.07.2008

[12] W3C Recommendation XML: http://www.w3.org/TR/REC-xml/ Last visited: 09.07.2008

[13] W3C Recommendation DTD: http://www.w3.org/TR/REC-xml/#dt-doctype Last visited: 09.07.2008

[14] W3C Recommendation XPATH: http://www.w3.org/TR/xpath20/ Last visited: 09.07.2008

[15] Wan, C. and Liu, X. 2007. Structural join and staircase join algorithms of sibling relationship. *J. Comput. Sci. Technol.* 22, 2 (Mar. 2007), 171-181. DOI= http://dx.doi.org/10.1007/s11390-007-9023-9

[16] Wang, W. 2004 *Structural Join: Processing Algorithms and Size Estimation*. Doctoral Thesis. UMI Order Number: AAI3137064., Hong Kong University of Science and Technology (People's Republic of China).

[17] Che, D. 2006. MyTwigStack: A Holistic Twig Join Algorithm with Effective Path Merging Support. In *Proceedings of the Seventh ACIS international Conference on Software Engineering, Artificial intelligence, Networking, and Parallel/Distributed Computing* (June 19 - 20, 2006). SNPD-SAWN. IEEE Computer Society, Washington, DC, 184-189. DOI= http://dx.doi.org/10.1109/SNPD-SAWN.2006.51

[18] Son, S., Shin, H., and Xu, Z. 2007. Structural Semi-Join: A light-weight structural join operator for efficient XML path query pattern matching. In *Proceedings of the 11th international Database Engineering and Applications Symposium* (September 06 - 08, 2007). IDEAS. IEEE Computer Society, Washington, DC, 233-240. DOI= http://dx.doi.org/10.1109/IDEAS.2007.43

[19] Lu, J., Chen, T., and Ling, T. W. 2004. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proceedings of the Thirteenth ACM international Conference on information and Knowledge Management* (Washington, D.C., USA, November 08 - 13, 2004). CIKM '04. ACM, New York, NY, 533-542. DOI= http://doi.acm.org/10.1145/1031171.1031272

[20] Fontoura, M., Josifovski, V., Shekita, E., and Yang, B. 2005. Optimizing cursor movement in holistic twig joins. In *Proceedings of the 14th ACM international Conference on information and Knowledge Management* (Bremen, Germany, October 31 - November 05, 2005). CIKM '05. ACM, New York, NY, 784-791. DOI= http://doi.acm.org/10.1145/1099554.1099741

[21] Chen, Q., Lim, A., and Ong, K. W. 2003. D(k)-index: an adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD international Conference on Management of Data* (San Diego, California, June 09 - 12, 2003). SIGMOD '03. ACM, New York, NY, 134-144. DOI= http://doi.acm.org/10.1145/872757.872776

[22] Shanmugasundaram, J., Shekita, E., Kiernan, J., Krishnamurthy, R., Viglas, E., Naughton, J., and Tatarinov, I. 2001. A general technique for querying XML documents using a relational database system. *SIGMOD Rec.* 30, 3 (Sep. 2001), 20-26. DOI= http://doi.acm.org/10.1145/603867.603871