

XLeaf: Twig Evaluation with Skipping Loop Joins and Virtual Nodes

Nils Grimsmo Truls A. Bjørklund
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
Email: {nilsgri,trulsamu}@idi.ntnu.no

Øystein Torbjørnsen
Fast Search and Transfer,
a Microsoft® subsidiary
Trondheim, Norway
Email: oysteint@microsoft.com

Abstract—XML indexing and search has become an important topic, and twig joins are key building blocks in XML search systems. This paper describes a novel approach using a nested loop twig join algorithm, which combines several existing techniques to speed up evaluation of XML queries. We combine structural summaries, path indexing and prefix path partitioning to reduce the amount of data read by the join. This effect is amplified by only reading data for leaf query nodes, and inferring data for internal nodes from the structural summary. Skipping is used to speed up merges where query leaves have differing selectivity. Multiple access methods are implemented as materialized views instead of succinct secondary indexes for better locality. This redundancy is made affordable in terms of space by using compression in a back-end with columnar storage. We have implemented an experimental prototype, which shows a speedup of two orders of magnitude on XPath queries with value predicates, when compared to existing open source and commercial systems using a subset of the techniques. Space usage is also improved.

Keywords—XML; query processing; column store;

I. INTRODUCTION

XML has evolved as the dominating data format for information exchange of structured and semi-structured data over the Internet. It may also be used for integration of data from disparate sources. When XML is generated from structured databases it is regular and has a known schema, while in other scenarios data is ad-hoc and without any schema.

XPath [31] and XQuery [32] are languages used to query XML data. XPath is a simple declarative language that supports matching of structure and content. XQuery is a more expressive iterative language, but uses XPath as a fundamental building block. This paper focuses on performance on a subset of XPath called *twig pattern matching*, which covers a majority of queries used in practice [11].

An XPath query is a tree, where the relation between the query nodes specify the relation between the data nodes in a possible match, and value predicates specify the contents of attributes and text nodes in the XML. The example query `/lib/book[author="Kant" and author="Gödel"]`, asks for all books coauthored by Kant and Gödel. A parent-child relation is denoted by `/`, an ancestor-descendant relation by `//`, and `[]` encloses

a predicate. Figure 1 shows a document where this query would have a match.

```
<lib>
  <book>
    <title>Kritik der Unvollständigkeit</title>
    <author>Kant</author>
    <author>Gödel</author>
    ...
  </lib>
```

Figure 1. Example XML document.

A typical system supporting XPath parses the XML and stores some information about each data node, usually its name, type and value, and its relation to other nodes. This can be stored in a table, and is usually sorted on document and node order for faster structural joins on the nodes in the query tree. When evaluating queries, it may be more advantageous to access nodes either by name, value, natural order, or other fields. Multiple secondary indexes can be added for direct lookup on all the fields stored for a node.

To find all matches for the example query, many current systems would read six lists of nodes from indexes, looking up the four XML element nodes on name (`lib`, `book`, `author` and `author`), and the two text nodes on value (`"Kant"` and `"Gödel"`). These would then be joined to give full matches, using the structural information stored about each node. In a library with millions of books, there would be just as many `book` nodes, and even more `author` nodes to join.

The main contributions of this paper are: 1) A twig join algorithm combining previous techniques in a novel way; 2) Implementation of an experimental prototype; 3) Experiments showing two orders of magnitude speedup over other systems on for queries with value predicates, and reduced space usage.

II. BACKGROUND

Indexing and querying XML has been a major research area both in the database and information retrieval community the last ten years, resulting in many different approaches.

A. Schema Agnostic XML Indexing

An early approach to XML indexing, which is usable with simple schema, is to translate the XML schema to a relational schema, and put shredded XML data into an RDBMS [28]. However, the XML schema can be complex, subject to updates, unknown, or non-existing. This flexibility may be considered a strength.

In *schema agnostic* XML indexing, all element-, attribute- and text nodes are extracted, and stored with information about their type, name, value and position in the tree. During query evaluation a list of matching data nodes is read for each query node, and these are joined into complete matches for the query tree. To allow joining matches for individual query nodes into tree pattern matches, the positional information must allow decision of node relationships, such as ancestor-descendant and parent-child. The most well known tree position encoding is the regional BEL encoding [37], which gives a node's *begin* and *end* position in the document, and its *level* (depth) in the tree.

Figure 2 shows an example using BEL encoding on the data extracted for the XML document in Figure 1. The data is usually indexed on name for XML element nodes, and on value for text nodes. When querying, one list of nodes is typically read for each node in the query. Retrieving element nodes on name (tag) is called *tag partitioning* (or tag streaming [6]). To evaluate the XPath query `//lib/book`, the nodes u and v (for `lib` and `book`) satisfying $u.begin < v.begin \wedge v.end < u.end \wedge u.level + 1 = v.level$, in addition to the type and name requirements, must be found.

<i>doc</i>	<i>begin</i>	<i>end</i>	<i>level</i>	<i>type</i>	<i>name</i>	<i>value</i>
1	1	...	1	Elem.	lib	
1	2	12	2	Elem.	book	
1	3	5	3	Elem.	title	
1	4	4	4	Text		"Kritik..."
1	6	8	3	Elem.	author	
1	7	7	4	Text		"Kant"
1	9	11	3	Elem.	author	
1	10	10	4	Text		"Gödel"
1	13	...	2	Elem.	article	
				...		

Figure 2. Data extracted for *tag partitioning*.

B. Tree-aware Joins

If the node lists are sorted on *doc* and *begin* values, a linear merge can be used when the schema and query are simple. But in other cases, matches may be formed from the lists out of order. Consider the document in Figure 3, and the query `//a[c and d]`. A linear merge of lists would miss one of the two pairs of *c* and *d* nodes usable together.

```

<a>
  <c/>
  <a> <c/> <d/> </a>
  <d/>
</a>

```

Figure 3. Breaking linear merge for `//a[c and d]`.

Using a full cross join on lists of data nodes matching each query nodes, and checking the output for legal matches, is not feasible for large data, as it can give intermediate results exponential in the size of the query, even when the final answer is small.

The first specialized tree joins focused on splitting the query tree into binary relationships and stitching the results into complete matches. Specialized loop joins gave optimal $\mathcal{O}(I + O)$ complexity for ancestor-descendant relationships [37], where I is the size of the input lists, and O is the size of the output. When joining an ancestor and a descendant list, a "marker" is left in the descendant list on matching positions, and the descendant list is "re-winded" to this position when the ascendant list is forwarded.

Later stack-based joins gave $\mathcal{O}(I + O)$ complexity also for parent-child relationships [2]. These maintained a stack of nested ancestor candidates in a single traversal of the two lists. Any current descendant candidate would be a descendant of all the nodes on the stack, and possibly the child of the top node.

As evaluating the query node relationships separately still gave useless intermediate results of exponential size, multi-way path and twig join algorithms were introduced [4]. By maintaining multiple stacks, these achieved optimal $\mathcal{O}(I + O)$ complexity using only $\mathcal{O}(d)$ memory, for path queries and twig queries using only the descendant axis. Later algorithms, which break the $\mathcal{O}(d)$ memory bound, achieve optimal complexity for all combinations of ancestor-descendant and parent-child edges [5].

C. Skipping Tree Joins

When the lists of matches for the different query nodes have similar size, regular tag partitioning is a fairly efficient solution, but that is often not the case. Take for example the query `//book[author="Kant"]`, evaluated on a library with millions of books. The leaf predicate probably has good selectivity, while the other nodes do not.

Skipping can be used to improve efficiency in such cases. Consider the query `//a//d`, and the problem of forwarding in the lists for the two query nodes to the first position where a match for *a* is an ancestor of a match for *d*. Skipping forward in the list for *d* to find the first d_j which can be a descendant of the current node a_i for *a*, is trivially done by finding the first d_j with $d_j.begin > a_i.begin$. If d_k is the last *d* node with $d_k.end < a_i.end$, then all *d* descendants of a_i (if any) are stored contiguously from d_j to d_k .

However, this technique cannot be used to skip in the ancestor list, as any node with a lower *begin* value is a candidate, and the actual ancestors can be spread evenly among a large number of such candidates. Specialized data structures can be used to skip efficiently in both ancestor and descendant lists [33].

D. Adding Structural Summaries

A different way of avoiding many useless nodes in the query node match lists is to do some partial matching in a preprocessing step. In the approach called *path indexing*, some of the structure of the data is extracted and put in a *path summary* [10], which is a tree containing all label-unique root-to-leaf paths seen in the data. This is used in a query preprocessing step for partially identifying structural matches for the query.

Figure 4 shows the structural data extracted from the example in Figure 1. Note that each path seen in the data (e.g. `/lib/book/author`) is only added once to the summary. The data stored for each document node is then linked to nodes in the summary in some way. One approach is shown in Figure 5. Summary tree nodes are given unique integer path IDs, which are referenced by the indexed data nodes. Here the Dewey encoding [30] is used to enumerate data node positions.

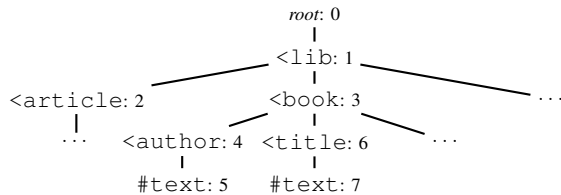


Figure 4. Structural summary.

The structural summary in the example contains information about single paths only. It is possible to store more structural information, for example about relations to other branches in the tree, at the expense of using more space. Note that even when a summary only gives matches for root-to-leaf paths in the query, some holistic pruning can be performed [34], removing matches that can never be part of a complete match for the whole query.

doc	Dewey	pathID	value
1	1	1	
1	1.1	3	
1	1.1.2	6	
1	1.1.2.1	7	"Kritik..."
1	1.1.3	4	
1	1.1.3.1	5	"Kant"
1	1.1.4	4	
1	1.1.4.1	5	"Gödel"
1	1.2	2	
		...	

Figure 5. Extracted for *prefix path partitioning*.

When searching using structural summaries and path indexing, the summary is first consulted to find structural matches for the root-to-node (prefix) paths of branching nodes and leaves in the query. Lists of data nodes matching the related summary nodes can be read. This is called *prefix path partitioning* (or prefix path streaming [6], as opposed to tag streaming). It usually results in reading less

data, as the set of matching paths is often much more selective than the node *name*.

E. Virtual Nodes

The Dewey encoding, which is used in Figure 5, has an advantage over the BEL encoding when combined with structural summaries, because it allows *ancestor reconstruction*. Only data node lists for branching and leaf query nodes need to be read, because matching of non-branching internal nodes is implied by the structural matching of the prefix paths of the checked nodes.

This approach is taken one step further by generating *virtual nodes* for the internal query nodes from the lists of leaf node matches [36]. Given the Dewey and pathID of a data node matching a leaf query node, the Dewey and pathIDs of all above data nodes can be inferred. The Dewey of an ancestor with depth d is a length d prefix of the Dewey of any descendant of the node, and the pathID can be found by going up to depth d in the summary tree.

Using Dewey to enumerate nodes requires non-linear space, and shows poor performance for some degenerate cases, but is commonly used in practice, such as in Microsoft SQL Server [22]. Space usage can be improved by using various compression schemes [13], and updatability can be attained [21].

F. Column Storage

A trend in database systems research the last decade has been investigating how performance can be improved for read intensive workloads. An important contribution in this field is *column store databases* [29], where each column in a table is stored separately. MonetDB/XQuery [20] is a well known XPath/XQuery system using columnar storage.

Advantages of column stores include reading less data, if not all columns in a table are needed for a query, better cache behavior, if *block processing* is used in a column, better compression, as you easily can use techniques such as *run length compression* (RLE) [1,14]. And as tuples can be materialized late, there can be less computational work, especially if block processing is used [1].

Using compression can give benefits beyond the obvious reduced space usage. Compression of inverted lists is commonly done in regular search engines to reduce disk I/O [35], and using compression can reduce the memory bus bottle-neck in database systems [23].

III. THE XLEAF SYSTEM

XLeaf is a novel combination of many previous techniques for evaluating twig queries. It combines structural summaries, prefix path partitioning, virtual node lists for internal query nodes, skipping joins, multiple access methods, compression and column storage. As most other academic XML search prototypes, our system supports the descendant and child axis, and simple value predicates.

The main difference between our approach and the majority of work on twig joins, is that we use a nested loop join, where the size of the state is linear in the size of the query. Most approaches read input lists once (streaming), and maintain larger intermediate results.

A. Query Evaluation

Algorithm 1 gives an overview of the process of evaluating a query in our system. First the query is matched against the summary, as described in Section III-B. Then an access method for candidate matches for each leaf query node is chosen, as described in Section III-C.

For queries with a single return node, such as in XPath, it is also often possible to avoid looping, and use a simple linear join. Such a join is correct when the depth of a branching node is fixed, as the query will not have out of order matches. The simple linear join is described in Section III-D, and the general looping join in Section III-E.

Algorithm 1 Overall query evaluation

```

1: procedure EVALUATE( $Q$ )
2:   SUMMARYMATCH( $Q$ )
3:   for  $l_q \in Q$ .leaves
4:     CHOOSEVIEW( $l_q$ )
5:   if no matches possible
6:     return
7:   if  $\forall b_q \in Q$ .branching :  $b_q$ .minDepth =  $b_q$ .maxDepth
8:     LINEARJOIN( $Q$ )
9:   else
10:    LOOPINGJOIN( $Q$ )

```

Note that in the following, only output, branching, leaf query nodes need to be considered. For internal query nodes with one child, matching is implied by the matching of the root-to-node paths of the nearest branching and leaf node descendants in the query tree. On the other hand, for a parent query node with multiple children, it is essential that all of these can use the same node for the parent in the data to construct a legal matching.

B. Summary matching

Our summary is indexed using an inverted index over the root-to-node paths in the summary tree, similar to what is described in [12]. Paths in the document tree are viewed as strings of node names, where the names are dictionary coded and stored as integers. The first step of the summary matching is finding the matches for the individual root-to-leaf paths in the query. The most selective element name in each path is looked up in the inverted index, and a list of candidate paths is retrieved. This list is then matched against the pattern.

For each query node, the matching path IDs are saved, along with their respective possible candidates for the parent branching node. This is more robust than storing all legal combinations of matches for above query nodes, as their

total number can be exponential in the size of the query. This typically happens with deeply recursive schemas.

After finding individual root-to-leaf path matches, the query tree is traversed bottom up and then top-down to prune away node matches which can never be part of a complete match.

C. Data access methods

It is common in XML indexing systems to index element nodes on name and text nodes on value. For *prefix path partitioning*, nodes must also be accessible on path. In our system we use multiple access methods for all node types, and attempt to choose the most efficient for each query leaf node during twig evaluation. Lists for internal nodes are virtual (see Section III-D and III-E).

One approach for implementing multiple access methods is storing a table with all nodes in the data, and having multiple indexes point into this storage. But if reading matches from such an index means following pointers into a node table, this will give bad spatial locality. It also makes it inefficient to use compression schemes with expensive random lookups. To avoid this, we have used multiple materialized views of the data table, with different sorting orders. The views we create are shown in Figure 6, where underlined fields give the sort order.

t_base	(<u>doc</u> , dewey, nameID, pathID, value)
t_value	(value, <u>doc</u> , dewey, nameID, pathID)
t_path	(pathID, <u>doc</u> , dewey, value)
t_name	(nameID, <u>doc</u> , dewey, pathID, value)

Figure 6. Materialized views used

The base table is built while scanning the data, while the other tables are sorted using a stable ternary quicksort. As the sorting is stable, and the source is sorted on the first two columns, the target table is in order after only sorting on the first target column.

The mappings for nameID and pathID are stored in tables, which are read into separate in-memory data structures for fast lookup. Note that the field nameID also indicates the type of the node (element, attribute, text, etc..), and that for text nodes, the name of the parent element node is stored. In cases where there are multiple pathID matches for a leaf in a query, the hits have to be merged when read from the table t_path. In many cases, it is cheaper to scan the t_name matches, and filter out non-matching nodes on pathID.

D. Simple Linear Join

Assume that as in XPath the query has one output node, and that there is one list of data node matches for each leaf node q in the query, which can be read with READ(q) and moved one data node forward with a call to ADVANCE(q).

The linear join shown in Algorithm 2 is used when for each branching query node, there is a fixed tree depth for the matching summary nodes. The correctness of using a linear

join in this case is trivial from the proofs for TwigStack [4]. There, matches for root-to-leaf paths are output from the first step of their join algorithm sorted root-to-leaf on the document order of the matched nodes, and fed to a linear merge, which is the second and last step. Since the depths of all branching nodes are fixed, the lists for the leaf node matches will already be in the correct order, from the fact that the data is a tree.

Algorithm 2 Simple linear join

```

1: procedure LINEARJOIN( $Q$ )
2:    $q_s := Q.selectingNode$  ▷ Assume a leaf
3:   while SIMPLENEXTMATCH( $Q.root$ )
4:     OUTPUT( $q_s$ )
5:     ADVANCE( $q_s$ )

   ▷ Note non-branching, non-leaf, non-selecting nodes ignored.
6: procedure SIMPLENEXTMATCH( $q$ )
7:   if ISLEAF( $q$ )
8:      $q.d := READ(q).dewey$ 
9:     return  $q.d$ 
10:   $m := q.minDepth$  ▷ Equals  $q.maxDepth$ 
11:   $x := \max_{q_c \in q.children} q_c.d$ 
12:  while true
13:    for  $q_c \in q.children$ 
14:      ALIGN( $q_c, x, m$ )
15:       $x_c := SIMPLENEXTMATCH(q_c)$ 
16:      if  $x_c = \emptyset$  return  $\emptyset$ 
17:       $x := \max x_c, x$ 
18:  if No change since last step
19:     $q.d := (x_1, \dots, x_m)$ 
20:    return  $q.d$ 

21: procedure ALIGN( $q, x, m$ )
22:   while READ( $q$ ).dewey <  $(x_1, \dots, x_m)$  ▷ Lexicographic
23:     if not ADVANCE( $q$ ) return

```

The reason no *pathID* ever needs to be read in Algorithm 2, is that the incoming leaf lists only contain structural matches for the root-to-leaf paths, with above branching node matches fixed in depth. A simple comparison of Deweys determine a common branching node in the data.

E. Loop Join

For cases where a simple linear join cannot guarantee a correct result, a nested loop join is used. There are two main modifications from the simple linear join. Markers are left in the lists, to which the list cursors are re-winded when necessary. And for a given candidate alignment of the leaf lists, it must be checked whether it is possible to choose candidates for the branching query nodes which are usable for all the leaf matches.

In previous loop join approaches [4], where there are explicit lists for internal query nodes, child list markers are updated when the parent’s list cursor is forwarded, but this method cannot be used directly in our approach. First we advance the list cursors to alignment based on the Deweys

matched down to the minimal possible depth of the above branching nodes. This depth is the maximal among the leaf matches minimal depths for a candidate for the branching node (*max-of-the-min*) from the summary matching phase. Then list positions are marked, before we iterate through the possible alignments. When list number i in the order is forwarded, list $i + 1$ is re-winded to the mark, before it is aligned based on Dewey, and the new mark is saved if differing from the previous. To speed up the iteration, the lists for all leaf query nodes are ordered on the expected number of hits.

The procedure depicted in Algorithm 3 checks whether a given alignment is a match. First, the query is traversed bottom-up, and common candidates for the branching nodes are chosen. Then it is traversed top-down, choosing the uppermost common candidate for each branching node. Finally, it is checked that the chosen branching nodes are the same physical nodes, by comparing the Dewey’s. Note that the top-down pass and the final bottom-up pass can be completed in one top-down traversal, as the Dewey for an internal node need not be materialized, but is given from the Dewey of any leaf descendant and the chosen depth.

Even though bit-vectors are used to implement the sets of candidates, using Algorithm 3 is expensive. For cases where many alignments are mismatches, it is cheaper to check the Deweys to the *max-of-the-min* depth first. This gives false positives, and matches must be checked with Algorithm 3. The *max-of-the-min* used here can be calculated from the branching nodes usable for the current leaf node matches, instead of from *all* branching node candidates from the summary matching.

In cases where the output node is not a leaf value node, but matches XML element nodes, care must be taken iterate through all candidates, and not to chose duplicates (line 20 in Algorithm 3). Also, to output entire data subtrees as matches, the table t_base (see Section III-C) must be scanned or searched during the query process to retrieve all nodes which have the given Dewey as a prefix.

F. Skipping

Skipping is crucial for efficient merges, and is included in Algorithm 2 by modifying the procedure ALIGN to use underlying data structures to skip forward to the next item matching the parameter x . Skipping is implemented in our system by combining a “one level skip list” [35] and search. Every k -th value is extracted from the columns on which merges will be performed (doc and $dewey$), and put in a separate column. k is chosen to be a power of two (usually 32 in our system) to allow for arithmetics using shifts. Note that this column can also double as check-points in compression (see Section III-G). When forwarding lists to values, the first few values in the smaller column are checked, and if no match is found, a binary search

Algorithm 3 Checking leaf alignment

▷ Note non-branching, non-leaf, non-selecting nodes ignored.

```
1: procedure CHECKLEAFALIGNMENT( $Q$ )
2:    $q_t := Q.topBranchingNode$ 
3:   CANDIDATES( $q_t$ ) ▷ Bottom-up
4:   if  $q_t.C = \emptyset$ 
5:     return false
6:   CHOOSEUPPERMOST( $q$ ) ▷ Top-down
7:   if not CHECKMATCH( $q$ ) ▷ Bottom-up
8:     return false

9: procedure CANDIDATES( $q$ )
10: if ISLEAF( $q$ )
11:    $q.p := READ(q).pathID$ 
12:   return  $q.C := \{q.p\}$ 
13: else
14:   return  $q.C := \bigcap_{q_i \in CHILDREN(q)} PARENTCAND(q_i)$ 

15: procedure PARENTCAND( $q$ )
16:   return  $\bigcap_{c_i \in CANDIDATES(q)} q.parMatchCand[c_i]$ 

17: procedure CHOOSEUPPERMOST( $q$ )
18:   if ISLEAF( $q$ )
19:     return
20:    $q.p := \arg \min_{c_i \in q.C} c_i.depth$ 
21:   for  $q_i \in q.children$ 
22:      $q_i.C = \{c_i \mid c_i \in q_i.C \wedge q.p \in q.parMatchCand[c_i]\}$ 
23:     CHOOSEUPPERMOST( $q_i$ )

24: procedure CHECKMATCH( $q$ )
25:   if ISLEAF( $q$ )
26:      $q.d := READ(q).dewey$ 
27:     return true
28:   for  $q_i \in q.children$ 
29:     if not CHECKMATCH( $q_i$ )
30:       return false
31:    $x := q.children[1].d$ 
32:    $m := q.p.depth$ 
33:    $q.d := (x_1, \dots, x_m)$ 
34:   for  $q_i \in q.children$ 
35:     if  $LCP(q.d, q_i.d) < m$ 
36:       return false
37:   return true
```

is performed there. Then a segment of the full column is scanned linearly.

G. Storage Back-end

The first iteration of this prototype used MonetDB/SQL [20] as a back-end for storing the data. This was changed for our own column store back-end because the overhead of using the communication interface to the server back-end was significant, and because of the lack of compression in the open source version of MonetDB.

Our minimalistic implementation uses memory mapped files to write to and read from disk, and uses compression to reduce the space usage. A column adapter chooses which compression method to use based on statistics from the data. Different compression schemes are favorable depending on

whether the columns are self-ordered, and whether they have few or many distinct values.

Supported storage methods for integer types are raw, bit packing, run length encoding (RLE), delta encoding and dictionary encoding. The last two are more expensive, and are used only in the maximum compression variant in the experiments (see Section IV). Typically RLE or delta coding is chosen for (partially) sorted columns, and bit packing or dictionary encoding for unsorted columns. The delta encoding is done using VByte [27], with checkpoints for faster random access. The dictionary encoded column sorts values on frequency, and uses VByte to store the coded values. Many column types are stored as multiple physical columns. For example is an RLE column stored as separate “values” and “cumulative count” columns.

Strings are stored raw, or using a dictionary. For the raw strings, a column of pointers is used for random access. Dictionaries are shared between the different materialized views. For logical columns consisting of several physical columns, these are compressed recursively if this is considered favorable.

In the current prototype, no explicit indexing is done, and a simple binary search is used. In the following experiments, all queries are run warm to simplify the experimental setup. Then the poor spatial locality of the binary search is not so much an issue, but for later experiments, running large batches of queries over more data, indexing should be considered.

However, some of the column types have specialized search implementations. For RLE, the “values” column is searched, and the “cumulative count” column is used to translate to row numbers. For the delta encoding, a search in the checkpoints is done first, before decoding the final block of coded values. For the dictionary encoding, the dictionary is searched first, before searching for the coded value in the data column.

A note should be given on the encoding of Deweys. A variable number of bytes are used per element in the Dewey, and the first bits in an element number give the number of bytes in the element. This allows comparing two Deweys with a simple string comparison to find which is smaller. This is useful when merging lists of nodes. To check if two Deweys match to a given depth in the document tree, the codes must be parsed when compared.

IV. EXPERIMENTAL RESULTS

This section presents results for query performance evaluation of various XPath implementations. Experiments were performed on an AMD Athlon 64 Processor 3500+ at 2.2 GHz, with 4GB main memory, running Linux 2.6.28.

The open source MonetDB/XQuery (MoXQ) [3] and an anonymous commercial system (SysA) were tested. They were included because they feature some, but not all, of the techniques from Section III-A. SysA uses path indexing and

multiple access methods, but reads lists for internal query nodes. MoXQ uses tag partitioning with no skipping, and has a column store back-end. The release from August 2008 was used. We have also tested the systems Berkeley DB XML, X-Hive and eXist, but the results are omitted, as these systems had poorer performance than MoXQ and SysA, and the results did not yield further insights.

For our own prototype we have included performance results for the compression schemes none (XLeaf_{none}), lightweight (XLeaf_{light}) and maximum (XLeaf_{max}). The latter includes delta and dictionary encoding for integer types.

It should be noted that a comparison of a minimalistic prototype with full fledged systems with features like transaction management is not fair. Ideally algorithms should be compared implemented in the same framework, but this was not done in these preliminary experiments, due to time constraints.

Some queries have been rewritten to counting queries to avoid measuring the overhead of printing answers, and because outputting hundreds of thousands of results is not a probable user scenario. All queries were given 2 warm-up runs, and were executed 10 times.

A. Indexing Performance and Space Usage

Figure 7 shows the indexing performance of the tested methods on the DBLP corpus [17], and the artificial benchmark XMark [26]. The table shows megabytes indexed per second, and the size of the index divided by the size of the original unparsed data.

	MoXQ	SysA	XLeaf _{none}	XLeaf _{light}	XLeaf _{max}
DBLP (440 MB)					
MB/s	1.92	0.27	0.43	0.45	0.30
Space	2.6	14.1	5.0	1.84	1.59
XMark (1118 MB)					
MB/s	8.8	0.45	1.39	1.41	1.06
Space	1.78	10.8	4.2	1.32	1.20

Figure 7. Indexing performance

MoXQ has the fastest indexing, and is fairly space efficient. The increased space requirements for adding the additional access methods in SysA may make it unusable in some scenarios. The space usage for our uncompressed variant (XLeaf_{none}) may also be too high, but the two compressed variants are very space efficient, and use less space than MoXQ, even though they feature multiple access methods like SysA, and uses the Dewey encoding, which has more redundancy than the BEL encoding used in MoXQ. Note that building the lightweight compressed index XLeaf_{light}, is faster than building the uncompressed index, while the heavily compressed variant is much slower, but only reduces storage requirements by 10 – 15%.

B. DBLP Queries

Figure 8 shows some queries on DBLP taken from [25], and table 9 gives the running times.

#	Query	Hits
P1	//inproceedings[author="Jim Gray"] [year="1990"]/@key	6
P2	//www[editor]/url/text()	5
P3	//book/author[text()="C. J. Date"]/text()	13
P4	//inproceedings [title/text()="Semantic Analysis Patterns."] /author/text()	2

Figure 8. DBLP queries.

	P1	P2	P3	P4
MoXQ	1815	131	257	1151
SysA	972	2.5	0.53	7.2
XLeaf _{none}	0.126	0.064	0.039	0.058
XLeaf _{light}	0.130	0.064	0.044	0.058
XLeaf _{max}	0.42	0.123	0.044	0.067

Figure 9. Query performance. Run-time in milliseconds.

The results for MoXQ are worse than one might expect, especially on P1 and P4. This is because the data has a very flat and repetitive structure, with many node matches for //inproceedings. SysA has better performance, due to the use of path indexing. But it is still slow on P1, probably because merging the three branches is expensive. In P2, the total number of matches for //www is low.

Our implementations perform very well in this experiment, because only data for leaf nodes are read, and skipping is applied in the join. The differences between XLeaf_{none} and XLeaf_{light} are almost negligible. We expected the lightweight compression to give better performance due to lower data bandwidth requirements. One reason this is not the case may be that queries are run hot, which reduces the bandwidth bottleneck due to caching effects, in combination with slightly more expensive computation for XLeaf_{light}. XLeaf_{max} has worse performance, due to even more computation, and perhaps worse memory access patterns, because of dictionary compression for integer types.

C. XPathMark A Queries

Queries A1-A6 on the XMark data are from XPathMark [8,9], an XPath equivalent of the XQuery benchmark XMark. XMark is artificially generated, and has properties rather different from DBLP, which has a flat and repetitive structure. XMark is a deeper tree, following a recursive schema.

MoXQ and SysA seem to have about equal performance, with the former in the lead. In these tests, the former performs relatively much better than on DBLP, because value predicates are not used, and the data tree is differently shaped. The number of matches for nodes near the root of the query are usually low, and the query leaf matches make up the majority. This gives a much cheaper join relative to the total number of matches.

Note the performance of MoXQ and XLeaf_{light} (or SysA) on queries A1 and A2, which highlights a key difference of the two systems. MoXQ is twice as fast on A2 as on A1, even though it has three times as many hits. On A1 it must

#	Query	Hits
A1	count (/site/closed_auctions/closed_auction/annotation/description/text/keyword)	40726
A2	count (//closed_auction//keyword)	124843
A3	count (/site/closed_auctions/closed_auction//keyword)	124843
A4	count (/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date)	26570
A5	count (/site/closed_auctions/closed_auction[descendant::keyword]/date)	53342
A6	count (/site/people/person[profile/gender and profile/age]/name)	32242
V1	... keyword[text ()=" preventions "] ...	55
V2	... keyword[text ()=" preventions "] ...	145
V3	... keyword[text ()=" preventions "] ...	145
V4	... keyword[text ()=" preventions "] ... date[text ()="06/27/1998"] ...	11
V5	... keyword[text ()=" tempests "] ... date[text ()="04/18/1999"] ...	12
V6	... gender[text ()="male"] ... age[text ()="18"] ... name[text ()="Mehrdad Takano"] ...	19

Figure 10. XMark queries from XPathMark. Queries V1-V6 are equal to queries A1-A6 with added value predicates.

	A1	A2	A3	A4	A5	A6	V1	V2	V3	V4	V5	V6
MoXQ	214	85	110	263	156	348	272	249	262	323	294	456
SysA	3.9	352	348	178	2128	446	11.8	398	371	30	46	97
XLeaf _{none}	9.1	72	73	56	153	181	0.160	0.27	0.29	0.32	0.144	4.3
XLeaf _{light}	8.5	94	94	57	180	196	0.24	0.38	0.28	0.58	0.20	4.6
XLeaf _{max}	9.6	93	94	166	190	708	0.21	0.32	0.34	3.6	0.70	24

Figure 11. Query performance. Run-time in milliseconds.

merge matches for seven internal nodes, but on A2 only two of these are involved. XLeaf_{light}, on the other hand, looks up the first query on *pathID*, and all nodes are matches. The second query is looked up on *nameID* and filtered on *pathID*, as there are multiple matches for the latter. Note that the performance on A3 is the same as on A2, as the executions are identical on our system. The fact that MoXQ is almost as fast as XLeaf_{none} on A2, even though it reads more data and does more work, indicates a better implementation.

XLeaf_{light} is faster on A4 than A3, even though the former is branching. The reason is that the leaves can be looked up directly on *pathID*. Hits are found efficiently using skipping. A5 is three times as slow, even though there are only two times as many hits. This is because the predicate leaf on *keyword* has more matches. A6 is also much slower than A4, even though the number of hits is similar, again because the individual query leaves have more matches.

Comparing SysA and XLeaf_{light} on queries A2 and A3 may show the benefit of using a column store back-end. SysA also looks up nodes on path in these queries, but is more than three times slower. Note that for the branching queries A4-A6, its disadvantage of reading matches for branching nodes does not show as much as on the DBLP queries, as the data is deeper and more “tree shaped”, with more matches for the query leaves than for the branching nodes.

D. XPathMark A Queries, with Value Predicates

Queries V1-V6 are the same as A1-A6, but with added value predicates. These were chosen to give as many hits as possible.

MoXQ is slightly slower with value predicates, probably because of the extra text nodes to be read and merged. SysA is also slower on the first three queries, but faster on the last three, because looking up the leaves on value is much more selective. There are however still many matches

for the branching node, giving an unnecessarily expensive join. A comparison of SysA and XLeaf_{light}, both using path indexing, shows the benefits of our systems features. Our implementation avoids handling the large number of matches for the internal query nodes, and is 20-200 times faster.

Query V6 is more expensive than the others for XLeaf_{light}, because the result for one of the leaves is large, and the merge is more expensive, even when skipping is used and the simple linear merge is allowed.

E. Queries with Decreasing Path Specification

Queries S1-S4 in Table 13 gives some queries on DBLP with decreasingly specified paths, using an increasing number of wild-cards.

This test shows that our systems are more robust for partially specified paths. With MoXQ and SysA, the query cost increases greatly from query S2 to S3, because a larger number of nodes become candidates for the branching node. Query S4 is a degenerate query which probably never would be seen in a real system, but it can be argued that S3 is not unrealistic. Our implementation is very fast on all queries. One leaf is looked up on *value* consistently, while the other on *pathID* for S1 and S2, and *nameID* for S3 and S4, which is the reason the last two queries are slower.

F. DBLP Queries with Increasing Branching Complexity

Queries B0-B5 in Table 13 show performance results on queries with increasing branching complexity.

Comparing B0 and B1, the main cost of MoXQ seems to be merging with the lists for */dblp/**, which is large. Additional branches give no significant extra cost. Note that this node is critical for the semantics of queries B2-B5. The poor results for SysA on B2-B5 are due to a “performance bug”. The system sometimes chooses to use nested loop lookups, even though other approaches would be more efficient.

#	Query	Hits
S1	/dblp/inproceedings[@key="conf/3dica/RohalyH00"]/booktitle/text()	1
S2	//inproceedings[@key="conf/3dica/RohalyH00"]/booktitle/text()	1
S3	//*[@key="conf/3dica/RohalyH00"]/booktitle/text()	1
S4	//*[@*="conf/3dica/RohalyH00"]/booktitle/text()	1
B0	/dblp//author[text()="Michael Stonebraker"]/text()	215
B1	/dblp/*/author[text()="Michael Stonebraker"]/text()	215
B2	/dblp/*[author/text()="Michael Stonebraker"]/title/text()	215
B3	/dblp/*[author/text()="Michael Stonebraker"][author/text()="Hector Garcia-Molina"]/title/text()	4
B4	/dblp/*[author="Michael Stonebraker"][author="Hector Garcia-Molina"] [@key="journals/corr/cs-DB-0310006"]/title/text()	1
B5	/dblp/*[author="Michael Stonebraker"][author="Hector Garcia-Molina"] [@key="journals/corr/cs-DB-0310006"][year/ > 1950]/title/text()	1

Figure 12. Queries with decreasing path specification, and queries with increasing branching complexity, on DBLP.

	S1	S2	S3	S4	B0	B1	B2	B3	B4	B5
MoXQ	1549	1387	4553	4694	1464	2609	2704	2940	2948	3029
SysA	160	289	21911	21932	1450	1247	146273	146353	149293	19127
XLeaf _{none}	0.054	0.054	0.59	0.93	0.084	0.085	0.63	0.25	0.159	0.20
XLeaf _{light}	0.055	0.054	0.50	0.96	0.086	0.088	1.00	0.26	0.156	0.192
XLeaf _{max}	0.060	0.062	0.51	0.98	0.114	0.104	2.2	0.78	0.21	0.27

Figure 13. Query performance, with run-time in milliseconds.

Our implementation is also affected by the added branches in the queries, with some interesting effects. B3 is faster than B2, because the join algorithm takes advantage of the two `author` lists being more selective. The common result is smaller, and more can be skipped when joining with the title. A similar argument holds for B4 vs B3, but for B5, the added predicate on year has low selectivity, and only adds to the cost.

V. RELATED WORK

Loop joins have been used previously in MPMGJN [37] for pairs of nodes, and in PathMPMJ [4] for path queries, while in our system loop joins are used for full twigs, and with with no physical lists of matches for internal nodes.

Most twig join algorithms are non-looping, and read the input lists once. The original TwigStack [4] maintains one stack of nested matches for each query node, and output individual root-to-leaf path matches, which are merged in a second phase. More complex intermediate result management are used by the later single phase join algorithms, such as Twig²Stack [5], HolisticTwigStack [16], TwigList [24] and TwigFast [18].

Structural summaries [10] are a prerequisite for prefix path partitioning, which is used previously in iTwigJoin [6]. A difference between iTwigJoin and our approach is that iTwigJoin uses materialized lists for all query nodes, and uses a specialized join to combine pairs of lists for directly related query nodes.

Generating matches for internal query nodes on the fly is done in the Virtual Cursors [36] algorithm and in TJFast [19], which use tag partitioning. A disadvantage of TJFast is that instead of using a structural summary, the name path and Dewey are stored compressed in the leaf lists. This makes reads unnecessarily expensive and increases space usage. A disadvantage of Virtual Cursors is that non-branching

internal nodes are not ignored during evaluation, because generated internal nodes are not guaranteed to have matching prefix paths. This also causes useless node candidates for branching internal nodes. A shortcoming of both previous approaches, and also ours, is that much work is repeated during construction of internal nodes.

Skipping joins has been used previously with tag partitioning in for example TSGeneric+ [15] and TwigOptimal [7]. When physical lists are used for the internal query nodes, skipping in a list to catch up with a descendant is not trivial, and specialized data structures like XR-trees [33] must be used. An advantage with virtual node lists is that skipping for internal nodes can be implicit. This is used previously Virtual Cursors [36], where B-trees were used to skip through leaf node matches.

Multiple access methods for query node matches, implemented through materialized views, is used for XML search in Microsoft SQL Server [22]. Their system uses prefix path partitioning similar to ours, but reads data node lists for internal query nodes, and uses row oriented storage. MonetDB/XQuery [3] has columnar storage, but it uses tag partitioning, and does not feature compression.

VI. CONCLUSION AND FUTURE WORK

This paper has investigated how to combine various techniques for twig matching. A prototype was developed to investigate possible benefits. When compared with two existing systems, which use only some of these techniques, a speedup of two orders of magnitude was shown for queries with value predicates.

Future work includes modifying the experimental prototype to allow switching features on and off individually, to investigate both their individual and combined benefit thoroughly. This may give more insight than comparing with

other full systems. More features, like optimal twig join algorithms, should also be added to our system.

ACKNOWLEDGMENT

Thank you to Felix Weigel for fruitful discussions. The first author was supported by the Research Council of Norway under grant NFR 162349.

REFERENCES

- [1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. SIGMOD*, 2008.
- [2] S. Al-Khalifa, HV Jagadish, N. Koudas, JM Patel, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE*, 2002.
- [3] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. SIGMOD*, 2006.
- [4] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, 2002.
- [5] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig²Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. VLDB*, 2006.
- [6] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. SIGMOD*, 2005.
- [7] Marcus Fontoura, Vanja Josifovski, Eugene Shekita, and Beverly Yang. Optimizing cursor movement in holistic twig joins. In *Proc. CIKM*, 2005.
- [8] M. Franceschet. XPathMark. <http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/> (Accessed: 2009-03-06).
- [9] M. Franceschet. XPathMark: an XPath benchmark for XMark generated data. In *Proc. XSYM*, 2005.
- [10] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, 1997.
- [11] Gang Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.
- [12] Nils Grimsmo. Faster path indexes for search in XML data. In *Proc. ADC*, 2008.
- [13] T. Härder, M. Haustein, C. Mathis, and M. Wagner. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowl. Engineering*, 2006.
- [14] Allison L. Holloway and David J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 2008.
- [15] H. Jiang, W. Wang, H. Lu, and J.X. Yu. Holistic twig joins on indexed XML documents. In *Proc. VLDB*, 2003.
- [16] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, and Qiang Zhu Dunren Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *Proc. DEXA*, 2007.
- [17] Michael Ley. DBLP XML Records, Jan. 2007. <http://www.informatik.uni-trier.de/~ley/db/> (Accessed: 2008-10-22).
- [18] Jiang Li and Junhu Wang. Fast matching of twig patterns. In *Proc. DEXA*, 2008.
- [19] J. Lu, T.W. Ling, C.Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In *Proc. VLDB*, 2005.
- [20] Monet. MonetDB web page. <http://monetdb.cwi.nl/> (Accessed: 2009-02-14).
- [21] Patrick O’Neil, Elizabeth O’Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: insert-friendly XML node labels. In *Proc. SIGMOD*, 2004.
- [22] Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML data stored in a relational database. In *Proc. VLDB*, 2004.
- [23] Niels Nes Peter Broncz, Marcin Zukowski. MonetDB/X100: Hype-pipelining query execution. In *Proc. CIDR*, 2005.
- [24] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make twig pattern matching fast. In *Proc. DASFAA*, 2007.
- [25] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using prüfer sequences. In *Proc. ICDE*, 2004.
- [26] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proc. VLDB*, 2002.
- [27] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.
- [28] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. VLDB*, 1999.
- [29] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amereson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *Proc. VLDB*, 2005.
- [30] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proc. SIGMOD*, 2002.
- [31] W3C. XPath 1.0, 1999. <http://www.w3.org/TR/xpath> (Accessed: 2009-04-29).
- [32] W3C. XQuery 1.0, 2007. <http://www.w3.org/TR/xquery/> (Accessed: 2009-04-29).
- [33] H.J.H.L.W. Wang and BC Ooi. XR-tree: Indexing XML data for efficient structural joins. In *Proc. ICDE*, 2003.
- [34] Felix Weigel. *Structural summaries as a core technology for efficient XML retrieval*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
- [35] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. 1999.
- [36] Beverly Yang, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. Virtual cursors for XML joins. In *Proc. CIKM*, 2004.
- [37] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 2001.