Hash-Based Structural Join Algorithms

Christian Mathis and Theo Härder

University of Kaiserslautern[**]

Abstract. Algorithms for processing Structural Joins embody essential building blocks for XML query evaluation. Their design is a difficult task, because they have to satisfy many requirements, e. g., guarantee linear worst-case runtime; generate sorted, duplicate-free output; adapt to fiercely varying input sizes and element distributions; enable pipelining; and (probably) more. Therefore, it is not possible to design *the* structural join algorithm. Rather, the provision of different specialized operators, from which the query optimizer can choose, is beneficial for query efficiency. We propose new hash-based structural joins that can process unordered input sequences possibly containing duplicates. We also show that these algorithms can substantially reduce the number of sort operations on intermediate results for (complex) tree structured queries (twigs).

1 Introduction

Because XML data is based on a tree-structured data model, it is natural to use path and tree patterns for the search of structurally related XML elements. Therefore, expressions specifying those patterns are a common and frequently used idiom in many XML query languages and their effective evaluation is of utmost importance for every XML query processor. A particular path pattern—the *twig*—has gained much attention in recent publications, because it represents a small but frequently used class of queries, for which effective evaluation algorithms have been found [1, 3, 7, 11, 14, 16].

Basically, a twig, as depicted in Fig. 1, is a small tree, whose nodes *n* represent simple predicates p_n on the content (text) or the structure (elements) of an XML document, whereas its edges define the relationship between the items to match. In the graphical notation, we use the double line for the descendant and the single line for the child relationship. For twig query matching, the query processor has to find all possible embeddings of the given twig in the queried document, such that each node corresponds to an XML item and the defined relationship among the matched items is fulfilled. The result of a twig is represented as an ordered[1] sequence

a) **Queries**
Q1) //book[title="XML"]//author[.//city="Rome"]/name
Q2) **for** \$b **in** //book, \$a **in** \$b//author
where \$b//title="XML" **and** \$a//city="Rome"
return (\$a, \$a/name)

b) **Twig for Q1 and Q2**

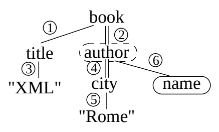


Fig. 1. Sample Query and Twig

[**] Database and Information Systems, D-67653 Kaiserslautern, Germany. {mathis|haerder}@informatik.uni-kl.de

[1] Here, “ordered” means: sorted in document order from the root to the leaf items.

of tuples, where the fields of each tuple correspond to matched items. Usually, not all nodes of a twig generate output, but are mere (path) predicates. Therefore, we use the term *extraction point* [7] to denote twig nodes that do generate output (the boxed nodes in Fig. 1).

1.1 Related Work

For twig query matching, a large class of effective methods builds on two basic ideas: the *structural join* [1] and the *holistic twig join* [3]. The first approach decomposes the twig into a set of binary join operations, each applied to neighbor nodes of the twig (for an example, see Fig. 2). The result of a single join operation is a sequence of tuples S_{out} whose degree (number of fields) is equal to the sum of the degrees of its input tuples from sequences S_{inA} and S_{inB}. S_{out} may serve as an input sequence for further join operations. In the following, we denote the tuple fields that correspond to the twig nodes to join as the *join fields*. The underlying structural join algorithms are interchangeable and subject to current research (see the discussion below).

In [3], the authors argue that, intrinsic for the structural join approach, intermediate result sizes may get very large, even if the final result is small, because the intermediate result has to be unnested. In the worst case, the size of an intermediate result sequence is in the order of the product of the sizes of the input sequences. To remedy this drawback, twig join algorithms [3, 7] evaluate the twig as a whole, avoiding intermediate result unnesting by encoding the qualifying elements on a set of stacks.

Of course, holistic twig join algorithms are good candidates for physical operators supporting query evaluation in XDBMSs. However, they only provide for a small fraction of the functionality required by complete XPath and XQuery processors (e. g., no processing of axes other than *child* and *descendant*; no processing of order-based queries). Therefore, the development of new structural join algorithms is still valuable, because they can act as complemental operators in case the restricted functionality of twig joins is too small, or as alternatives if they promise faster query evaluation.

Existing structural join approaches can roughly be divided into four classes by the requirements they pose on their input sequences: A) *no requirements* [8, 11, 14]; B) *indexed input* [16], C) *sorted input* [1, 10, 16]; D) *indexed and sorted input* [4]. Especially for classes C and D, efficient algorithms have been found that generate results in linear time depending on the size of their input lists. In contrast, for class A, there is—to the best of our knowledge—no such algorithm. All proposed approaches either sort at least one input sequence [11], or create an in-memory data structure (a heap) requiring $O(nlog_2n)$ processing steps [14]. By utilizing hash tables that can be built and probed in (nearly) linear time, the algorithms we introduce in this paper can remedy this problem. Note, the strategies in [11, 14] elaborate on partition-based processing schemes, i. e., they assume a small amount of main memory and large input sequences, requiring their partition-wise processing. Their core join algorithm, however, is main-memory–based, as ours is. Therefore, our new join operators can be—at least theoretically[2]—combined with the partitioning schemes proposed in these earlier works.

[2] [14] uses a perfect binary tree (PBiTree) to generate XML identifiers. In real-world scenarios, we assume document modifications that can hardly be handled with PBiTrees. Therefore, we

Answering twig (and more complex queries) using binary structural join algorithms imposes three non-trivial problems: selecting the best (cheapest) join order (*P1*) to produce a sorted (*P2*) and duplicate-free (*P3*) output. P1 is tackled in [15], where a dynamic programming framework is presented that produces query executions plans (QEPs) based on cost estimations. The authors assume class C (and D) algorithms, which means that even intermediate results are required to be in document order on the two join fields. As a consequence, sort operators have to be embedded into a plan to fulfill this requirement. Consider for example the twig in Fig. 1. Let the circled numbers denote the join order selected by an algorithm from [15]. Then, three sort operators have to be embedded into the QEP (see[3] Fig. 2). Sort operators are expensive and should be avoided whenever possible. With structural join algorithms not relying on a special input order—like those presented in this paper—we can simply omit the sort operators in this plan. However, a final sort may still be necessary in some cases.

Problem P3 was studied in [8]. The authors show that duplicate removal is also important for intermediate results, because otherwise, the complexity of query evaluation depending on the number of joins for a query Q can lead to an exponential worst-case runtime behavior. Therefore, for query evaluation using binary structural joins, tuplewise duplicate-free intermediate result sequences have to be assured after each join execution. Note, due to result unnesting, even a (single) field in the tuple may contain duplicates. This circumstance is unavoidable and, thus, we have to cope with it. Because duplicate removal—like the sort operator—is an expensive operation, it should be minimized. For example in [6], the authors present an automaton that rewrites a QEP for Q, thereby removing unnecessary sort and duplicate removal operations. Their strategy is based on plans generated by normalization of XPath expressions, resulting in the XPath core language expressions. However, this approach does not take join reordering into account, as we do. Our solution to P3 is a class of algorithms that do not produce any duplicates if their input is duplicate free.

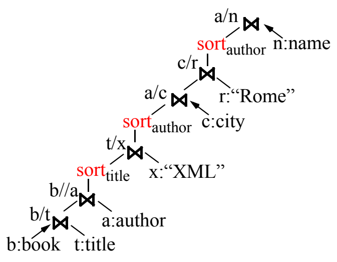


Fig. 2. Sample Plan

1.2 Contribution

We explore the use of hash-based joins for path processing steps of XML queries and identify the selectivity ranges when they are beneficial. In particular, we propose a class of hash-based binary structural join operators for the axes *parent*, *child*, *ancestor*, *descendant*, *preceding-sibling*, and *following-sibling* that process unordered input sequences and produce (unordered) duplicate-free output sequences. Furthermore, we show by extensive tests using the XTC (XML Transaction Coordinator)—our prototype of a native XDBMS—that our approach leads to a better runtime performance than sort-based schemes.

The remainder of this paper is organized as follows: Sect. 2 briefly describes some important internals of XTC, namely our node labeling scheme and an access method for

used SPLIDs (Sect. 2.1) instead. As a consequence, this "gap" had to be bridged to support the proposed partition schemes with our ideas.

[3] An arrow declares the input node of a join by which the output is ordered, where important. Possible are *root to leaf*, e. g., between "book" and "title", and *leaf to root*, e. g., the final join.

element sequences. Sect. 3 introduces new hash-based algorithms. In Sect. 4 we present our quantitative results before we conclude in Sect. 5.

2 System Testbed

XTC adheres to the well-known layered hierarchical architecture: The concepts of the storage system and buffer management could be adopted from existing relational DBMSs. The access system, however, required new concepts for document storage, indexing, and modification including locking. The data system available only in a slim version is of minor importance for our considerations.

2.1 Path Labels

Our comparison and evaluation of node labeling schemes in [9] recommends node labeling schemes which are based on the Dewey Decimal Classification [5]. The abstract properties of Dewey order encoding—each label represents the path from the documents root to the node and the local order w. r. t. the parent node; in addition, sparse numbering facilitates node insertions and deletions—are described in [13]. Refining this idea, similar labeling schemes were proposed which differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of these schemes are ORDPATH [12], DeweyID [9], or DLN [2]. Because all of them are adequate and equivalent for our processing tasks, we prefer to use the substitutional name *stable path labeling identifiers* (SPLIDs) for them.

Here we only summarize the benefits of the SPLID concept which provides holistic system support. Existing SPLIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. Comparison of two SPLIDs allows ordering of the respective nodes in document order, as well as the decision of all XPath axis relations. As opposed to competing schemes, SPLIDs easily provide the IDs of all ancestors to enable direct parent/ancestor identification or access. This property is very helpful for navigation and for fine-grained lock management in the XML documents. Finally, the representation of SPLIDs, e. g., label 1.3.7 for a node at level 3 and also used as an index reference to this node, facilitates the application of hashing in our join algorithms.

2.2 Accessing Ordered Element Sequences

A B*-tree is used as a document store where the SPLIDs in inner B*-tree nodes serve as fingerposts to the leaf pages. The set of doubly chained leaf pages forms the so-called document container where the XML tree nodes are stored using the format (SPLID, data) in document order. Important for our discussion, the XDBMS

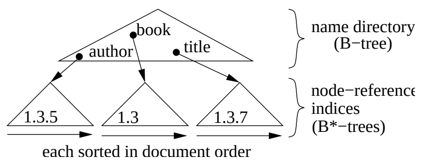


Fig. 3. Element Index

creates an *element index* for each XML document. This index consists of a *name directory* with (potentially) all element names occurring in the XML document (Fig. 3). For each specific element name, in turn, a *node-reference index* is maintained which addresses the corresponding elements using their SPLIDs. Note, for the document store and the element index, prefix compression of SPLID keys is very effective because both are organized in document order directly reflected by the SPLIDs [9].

The leaf nodes in our QEPs are either element names or values. By accessing the corresponding node reference indexes, we obtain for them ordered lists of SPLIDs and, if required lists of nodes in document order by accessing the document store.

3 Hash-Based Structural Join Algorithms

To be able to compete with existing structural join algorithms, we had to design our new algorithms with special care. In particular, the use of semi-joins has several important benefits. The processing algorithms become simpler and the intermediate result size is reduced (because the absolute byte size is smaller and we avoid unnesting). Several important design objectives can be pointed out:

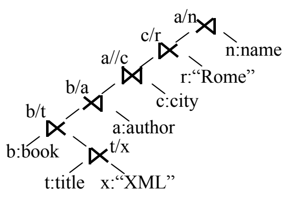


Fig. 4. Plan for Query 1

- *Design single-pass algorithms*. As in almost all other structural join proposals, we have to avoid multiple scans over input sequences.
- *Exploit extraction points*. With knowledge about extraction points, the query optimizer can pick semi-join algorithms instead of full joins for the generation of a QEP. For example, consider the plan in Fig. 4 which embodies one way to evaluate the twig for the XPath expression in Fig. 1. After having joined the *title* elements with the content elements "XML", the latter ones are not needed anymore for the evaluation of the rest of the query; a semi-join suffices.
- *Enable join reordering*. Join reordering is crucial for the query optimizer which should be able to plan the query evaluation with any join order to exploit given data distributions. As a consequence, we need operators for the reverse axes *ancestor* and *parent*, too (e. g., the semi-join operator between *title* and "XML" in Fig. 4 actually calculates the parent axis).
- *Avoid duplicate removal and sort operations whenever possible*. By using only algorithms that do not generate duplicates and operate on unordered input sequences, the query optimizer can ignore these problems. However, the optimizer has to ensure the correct output order, requiring a final sort operator. In some cases, this operator can be skipped: If we assume that the element scans at the leaf nodes of the operator tree in Fig. 4 return the queried element sequences in document order (as, for example, our element index assures), then, because the last semi-join operator is simply a filter for name elements (see Sect. 3.1), the correct output order is automatically established.
- *Design dual algorithms that can hash the smaller input sequence*. The construction of an in-memory hash table is still an expensive operation. Therefore, our set of algorithms should enable the query optimizer to pick an operator that hashes the smaller of both input sequences and probes the other one, yielding the same result.

Table 1. Classification of Hash-Join Operators

	Output		
Hashed	*ancestor/parent*	*descendant/child*	*full join*
	Class 1: UpStep	Class 2: TopFilter	Class 3: FullTopJoin
parent	//**a**[b]	//**a**/b	//**a**/b, //**a**[b]
	ParHashA	ChildHashA	ChildFullHashA
ancestor	//**a**[.//b]	//**a**//b	//**a**//b, //**a**[.//b]
	AncHashA	DescHashA	DescFullHashA
	Class 4: BottomFilter	Class 5: DownStep	Class 6: FullBottomJoin
child	//a[**b**]	//a/**b**	//a/**b**, //a[**b**]
	ParHashB	ChildHashB	ChildFullHashB
descendant	//a[.//**b**]	//a//**b**	//a//**b**, //a[.//**b**]
	AncHashB	DescHashB	DescFullHashB

3.1 Classification of Algorithms

We can infer three orthogonal degrees of freedom for structural hash-join algorithms: the *axis* that has to be evaluated (*parent*/*child*/*ancestor*/*descendant*); the *mode* of the join (*semi*/*full*); and the choice of which input sequence to *hash* (*A* or *B*)[4]. The following naming scheme is used for our operators: `<axis> + <mode> + <hash>:` {`Par|Child|Anc|Desc`} {`Semi|Full`} `Hash`{`A|B`} ("Semi" is omitted for brevity). For example, the join operator between *title* and "XML" in Fig. 4 is a `ParHashB` operator, because it calculates the parent axis, is a semi-join operator, and hashes the sequence of possible children.

For an overview of all possible operators refer to Table 1: The column header defines the input to be hashed, whereas the row header defines the output. For clarification of the semantics, each operator is additionally described by an XPath expression where the input sequence to hash is marked in bold face. The names of the operator classes describe the evaluation strategy of the join. They will be discussed in the following. Note, class 1–3 algorithms are dual to class 4–6 algorithms, i. e., they calculate the same result as their corresponding algorithms, but hash a different input sequence.

3.2 Implementation

To abstract from operator scheduling and dataflow control, we let all operators act in the same operating system thread and use the well-known iterator-based *open-next-close* protocol as a basis for the evaluation. Each algorithm receives two input sequences of tuples, where, due to intermediate result unnesting, duplicates on the join fields have to be expected.

All proposed algorithms in this paper consist of two phases. In phase one, a hash table `ht` is constructed using the join field of the tuples of one input sequence (either sequence A or B). In phase 2, the join field of the other input sequence is probed against `ht`. Depending on how a result tuple is constructed, the operators can be assigned to one of the six classes: *Full*Join* operators return a sequence of joined result tuples just as earlier proposals for structural join algorithms (e. g., [1]). Note, the qualifiers "Top" and "Bottom" denote which input sequence is hashed. The remaining classes contain

[4] Note, in the following, *A* denotes the sequence of possible ancestors or parents (depending on the context), whereas *B* denotes descendants or children.

```
Input: TupSeq A,B, Axis aixs, bool hashA
Output: TupSeq results,Local:HashTable ht

1  // phase 1: build hash table
2  if (hashA)
3    foreach (Tuple a in A)
4      hash a.jField() in ht;
5  else if (axis is 'Par' or 'Child')
6    foreach (Tuple b in B)
7      hash b.jField().parent() in ht;
8  else if (axis is 'Anc')
9    List levelOcc = getLevels(A);
10   foreach (Tuple b in B)
11     foreach (level in levelOcc)
12      hash b.jField().anc(level) in ht;
13
14 // phase 2: probe
15 foreach (Tuple t in ((hashA) ? B : A)
16   if (! hashA and
17      t.jField() in ht) results.add(t);
18   else if (axis == 'Child' or 'Par')
19      if (t.jField().parent() in ht)
20       results.add(t);
21
22   else if (axis == 'Desc' or 'Anc')
23     List levelOcc = getLevelsByProb(A);
24     foreach (level in levelOcc)
25       if (t.jField().anc(level) in ht)
26         results.add(t);
27         break inner loop;
28
29 function hashEnqueue
30       (SPLID s, Tuple t, HT ht)
31   Queue q = ht.get(s);
32   q.enqueue(t);
33   hash (s, q) in ht;
34
35 function hashDelete (SPLID s, HT ht)
36   Queue q = ht.get(s);
37   foreach (Tuple t in q)
38     results.add(t);
39   ht.delete(s);
40
41 function hashFull
42       (SPLID s, Tuple current, HT ht)
43   Queue q = ht.get(s);
44   foreach (Tuple t in q)
45     results.add(new Tuple(t, current));
```

Fig. 5. *Filter Operator and Auxiliary Functions for *Step and Full*Join

semi-join algorithms. **Filter* operators use the hash table, constructed for one input sequence to filter the other one, i. e., tuples are only returned from the probed sequence. **Step* operators work the other way around, i. e., they construct the result tuples from the hashed input sequence.

**Filter Operators* (see Fig. 5): In phase one, for `ChildHashA` and `DescHashA`, the algorithm simply hashes the SPLID of the elements of the join fields (accessed via method `jField()`) into `ht` (line 4). Then, in phase two, the algorithm checks for each tuple *t* in B, whether the parent SPLID (line 19 for `ChildHashA`) or any ancestor SPLID (line 25 for `DescHashA`) of the join field is contained in `ht`. If so, *t* is a match and is appended to the result. Actually, for the descendant operator, we had to check all possible ancestor SPLIDs which could be very costly. To narrow down the search, we use the meta-information, at which levels and by which probability an element of the join field of A occurs (line 23). This information can be derived dynamically, e. g., when the corresponding elements are accessed via an element index scan, or kept statically in the document catalog.

The strategy for `ParHashB` and `AncHashB` is similar, with the difference, that in the hash phase the algorithm uses the join fields of input B to precalculate SPLIDs that might occur in A (lines 7 and 12). Again for the descendant operator, we use the level information (line 9), but this time the probability distribution does not matter. In the probing phase it only has to be checked, whether the current join field value is in `ht`.

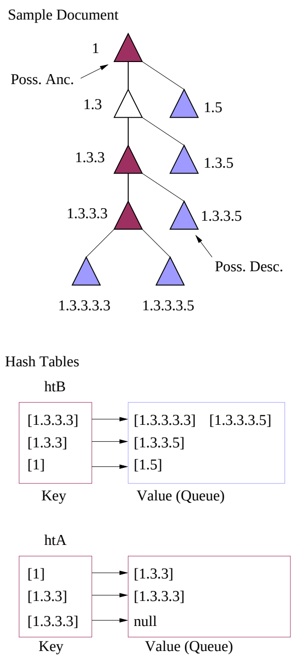
Obviously, the output order of the result tuples is equal to the order of the probed input sequence. Furthermore, if the probed input sequence is tuplewise duplicate free, the algorithm does not produce any duplicates. The *hashed* input sequence may contain duplicates. However, these are automatically skipped, whereas collisions are internally resolved by the hash table implementation.

**Step Operators* conceptually work in the same way as their corresponding **Filter* operators. However, they do not return tuples from the probed, but from the hashed input sequence. Accordingly, tuples that have duplicates on the *key* they use for hashing (e. g., TupSeq A of Fig. 7a) may not be skipped (as above) but have to be memorized for later output. The new algorithms work as follows: In the hash phase, the function `hashEnqueue()` (Fig. 5 line 29) is called instead of the simple hash statements in lines 4, 7, and 12). The first argument is the SPLID *s* of the join field (or its *parent*/*ancestor* SPLID). Function hashEnqueue() checks for *s* whether or not an entry is found in hash table `ht` (line 31). If so, the corresponding value, a queue *q*, is returned to which the current tuple is appended (line 32). Finally, *q* is written back into the hash table (line 33).

In the probing phase, we substitute the hash table lookup and result generation (lines 17, 19–20, 25–26) with the `hashDelete()` method (Fig. 5 line 35). For the given SPLID s to probe, this method looks up the corresponding tuple queue in the hash table and adds each contained tuple *t* to the result. Finally, the entry for *s* and its queue are removed from the hash table, because the result tuples have to be returned exactly once to avoid duplicates. The sort order of these algorithms is dictated by the sort order of the input sequence used for probing. If the hashed input sequence did not contain any duplicates, the result is also duplicate free.

The technique to memorize tuples with the same hash key works fine for the step operators `ParHashA`, `ParHashB`, and `AncHashA`. For `DescHashB`, however, the following problem occurs: In phase 1, the algorithm has to anticipate for each node b in tuple sequence B, on which level the ancestor nodes in the tuple sequence A can possibly reside. Then—following the technique above—it had to insert b into all queues of possible ancestors. As an example, consider the document shown in Fig. 6a and the element with the SPLID 1.3.3.5. In phase 1, the algorithm inserts 1.3.3.5 in the queue for the possible ancestor elements 1.3.3 and 1 (1.3 does not belong to any input). This is not only unfavorable because of the redundant storage of element 1.3.3.5 and the implied higher memory consumption, but it may also lead to duplicates in the final result: in the probing phase, the algorithm 1) *checks* for each a in input sequence A, if there is a corresponding tuple queue in the hash table, 2) *appends* all elements in the queue to the result sequence, and 3) *removes* the matched entry from the hash table. In the example, if the algorithm processes the possible ancestor 1.3.3, it would return the result tuple containing 1.3.3.5 and remove this entry for 1.3.3 from the hash table. If later on, possible ancestor 1 is probed, 1.3.3.5 is again returned. Thus, a duplicate would be generated.

To remedy these problems, a distinguished algorithm for the `DescHashB` operator is designed (see Fig. 6c). In the first phase, the operator builds two hash tables, named `htB` and `htA`, instead of only one. Hash table `htB` has the same function as in the other step operators: it keeps track of mappings from possible ancestor elements to queues of possible descendants. To avoid redundant storage, the possible descendant element b is only stored in the queue for the anticipated *lowest ancestor element* (line 4 to 6), which corresponds to the SPLID with the highest level that can still be an ancestor of b. For example, the tuple corresponding to 1.3.3.5 is only stored in the queue for 1.3.3, because 1.3.3 is the anticipated lowest ancestor (the storage of 1.3.3.5 in the



```
Input: TupSeq A,B
Output: TupSeq results
Local: HashTable htB,htA

1  // phase 1: build hash table
2  List levelOcc = getLevels(A);
3  foreach (Tuple b in B)
4    SPLID lowestAnc =
5      b.jField().lowestAnc(levelOcc);
6    hashEnqueue(lowestAnc, b, htB);
7    SPLID highestAnc =
8      b.jField().highestAnc
9                        (levelOcc,htA);
10
11   foreach (level in levelOcc between
12       highestAnc.level() and
13       lowestAnc.level() descending)
14     SPLID ancAnc =
15             b.jField().anc(level);
16     SPLID anc =
17             b.jField().anc(level+1);
18     Queue ancQ = htA.get(ancAnc);
19     ancQ.enqueue(anc);
20     ancQ = htA.get(anc);
21     ancQ.enqueue(null);
22
23 // phase 2: probe
24 foreach (Tuple a in A)
25   hashDelete(a, htB);
26   Queue q = new Queue();
27   q.addAll(htA.get(a.jField());
28   htA.remove(a.jField());
29   while (!q.isEmpty())
30     SPLID id = q.removeFirst();
31     hashDelete(id, htB);
32     q.addAll(htA.get(id));
33     htB.remove(id);
```

Fig. 6. a) Sample Document, b) Hash Tables, c) DescHashB Operator

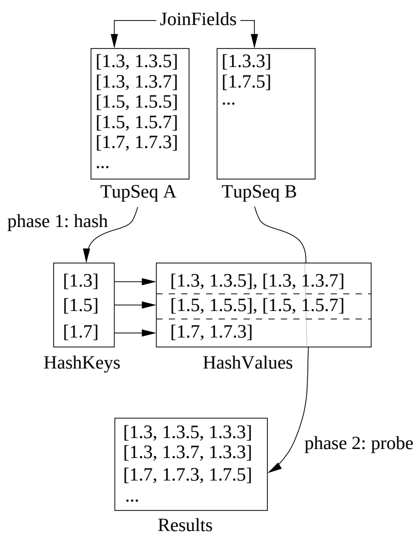
queue of element 1 is thus omitted). Then, another hash table (`htA`) is built which keeps track of ancestor/descendant relationships among the possible ancestor elements. In essence, `htA` stores a forest of trees. In the example, when element b=1.3.3.5 is processed, the key-value pairs (1, 1.3.3) and (1.3.3, null) are inserted into `htA`. Later on, when for example b=1.3.3.3.3 is processed, only the pair (1.3.3, 1.3.3.3) has to be inserted into `htA`, because the relationship of their ancestors is already contained in `htA`. This functionality is implemented in lines 7 to 21. First the highest possible ancestor SPLID, whose relationship is not yet contained in `htA` is computed. This can easily be done by comparing the keys already contained in `htA`. In the above example, when (1, 1.3.3) and (1.3.3, null) are present in `htA`, the highest possible ancestor for b=1.3.3.3.3 is 1.3.3. Afterwards the structural relationships down to the lowest possible ancestor are inserted into `htA` (lines 11 to 21).

In the probing phase, the algorithm calls `hashDelete()` (line 25). I. e., it probes each element a of the ancestor sequence A against `htB`. If there is a queue for the key a, the found tuples are written to the result and the matched key-value pair is removed from `htB`. For example, the lookup of a=1.3.3 immediately returns the tuple corresponding to 1.3.3.5 and the pair (1.3.3, 1.3.3.5) is removed from `htB`. Because the algorithm has to return *all* descendants, it follows the tree stored in `htA` rooted at a and calls `hashDelete()` for all possible ancestors found (lines 26 to 33). In the example, the algorithm looks up a=1.3.3 in `htA`, finds x=1.3.3.3, and calls hashDelete for x, which returns the descendants 1.3.3.3.3 and 1.3.3.3.5. All touched entries are removed from `htA`. Note, the operator fulfills our requirements: it does not produce any duplicates and can operate on unsorted input sequences.

*Full*Join Operators* resemble the **Step* operators. The only difference is the result generation. While **Step* algorithms are semi-join operators that do not produce a joined result tuple, *Full*Join* operators append the current result tuple with all tuples matched (as depicted in method `hashFull()`, Fig. 5 line 41). Note, opposed to `hashDelete()`, in `hashFull()` no matched entries from `ht`/`htA`/`htB` are deleted. For a brief *full join* example see Fig.7a: input sequence A for the `ChildFullHashA` operator is hashed on join field 1, thereby memorizing tuples with duplicates in the related queues. Then, the tuples from sequence B are probed against the hash table. For each match, each tuple in the queue is joined with the current tuple from B and appended to the result.

Space and Time Complexity. The space complexity (number of tuples stored) and time complexity (number of hashes computed) of the operators depend on the axis to be evaluated. Let $n = |A|$ and $m = |B|$ be the sizes of the input sequences. For the *parent*/*child* axis, the space and time complexity is $O(n+m)$. For the *ancestor*/*descendant* axis, the height h of the document also plays a role. Here the space complexity for classes 1–3 is also $O(n + m)$, whereas the time complexity is $O(n + h * m)$ (for each tuple in sequence B up to h hashes have to be computed). For classes 4–6, both space and time complexity are $O(n+h*m)$, except for the `DescHashB` operator, where the time complexity is $O(h * (n + m))$.

Beyond Twig Functionality: Calculation of Sibling Axes. With hash-based schemes and a labeling mechanism enabling the parent identification, the *preceding-sibling* and the *following-sibling* axes are—in contrast to holistic twig join algorithms—computable, too. Due to space restrictions, we can only show filtering algorithms, corresponding to the **Filter* classes above: In phase 1 operators `PreSiblHashA` and `FollSiblHashA` (see Fig. 7b) create a hash table `ht` to store key-value pairs of *parent*/*child* SPLIDs. For each element in A, parent *p* is calculated. Then the following-sibling (preceding-sibling) axis is evaluated as follows: For each parent SPLID *p*, the smallest (largest) child SPLID *c* in A is stored in `ht`. This hash table instance is calculated by successive calls to the `checkAndHash()` method (lines 14 to 21). While probing a tuple *b* of input B, the algorithm checks whether the SPLID on the join field of *b* is a following-sibling (preceding-sibling) of *c*, that has the same parent (lines 6 to 12). If so, the current *b* tuple is added to the result. Clearly, these algorithms reveal the same characteristics as their corresponding **Filter* algorithms: They do not produce any tuplewise duplicates and preserve the order of input sequence B.



```
Input: TupSeq A, B, Axis aixs
Output: TupSeq results, Local:HashTable ht

// phase 1: build hash table
foreach (Tuple a in A)
  checkAndHash(a.jField(), axis)

// phase 2: probe
foreach (Tuple b in B)
  SPLID s = ht.get(b.parent());
  if( (axis == 'PreSibl' and
       b.jField().isPreSibl(s)) or
      (axis == 'FollSibl' and
       b.jField().isFollSibl(s)) )
    results.add(b);

function checkAndHash(SPLID a, Axis axis)
  SPLID s = ht.get(a.parent());
  if( (s is NULL) or
      (axis == 'PreSibl' and
       not s.isPreSibl(a)) or
      (axis == 'FollSibl' and
       not s.isFollSibl(a)) )
    ht.put(a.parent(), a);
```

Fig. 7. a) Full*Join Example and b) Sibling Operator

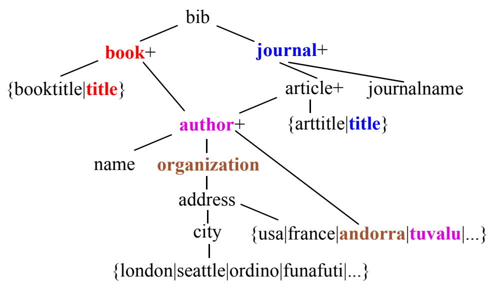
4 Quantitative Results

To substantiate our findings, we compared the different algorithms by one-to-one operator comparison on a single-user system. All tests were run on an Intel XEON computer (four 1.5 GHz CPUs, 2 GB main memory, 300 GB external memory, Java Sun JDK 1.5.0) as the XDBMS server machine and a PC (1.4 GHz Pentium IV CPU, 512 MB main memory, JDK 1.5.0) as the client, connected via 100 MBit ethernet to the server.

To test the dependency between runtime performance and query selectivity, we generated a collection of synthetic XML documents, whose structure is sketched in Fig. 8. Each document has a size of 200 MB and contains bibliographic information. Because we were mainly interested in structural join operators for element sequences, the generated documents do not contain much text content. The schema graph is a directed acyclic graph (and not a tree), because an author element may be the child of either a book or an article element. We generated the documents in such a way, that we obtained the following selectivity values for the execution of structural joins between input nodes: 1%, 5%, 10%, 50%, and 100%. For example, for the query `//book[title]`, selectivity 1% means that 1% of all *title* elements have a *book* element as their parent (all others have the *article* element as parent). Additionally, we created 10% noise on each input node, e. g., 10% of all *book* elements have the child *booktitle* instead of *title*.

4.1 Join Selectivity Dependency of Hash-Based Operators

In a first experiment, we want to explore the influence of the join selectivities of the input sequences and, in case of varying input sequence sizes, their sensitivity on the



*Filter Queries:
a) `//book[title]` **or** `//book/title`
b) `//journal[.//title]` **or** `//journal//title`

*Step and Full*Join Queries:
a) `//author[tuvalu]` **or** `//author/tuvalu`
b) `//organization[.//andorra]` **or** `//organization//andorra`

Fig. 8. Document Schema and Sample Queries

hash operator performance. All operators presented in Table 1 revealed the same performance characteristics as a function of the join selectivity. Hence, it is sufficient to present an indicative example for which we have chosen the `DescFullHash*` operators. For the query `//journal//title`, the size of the input sequence containing *journal* elements varies from around 2,000 to 200,000 elements, whereas the size of the *title* sequence remains stable (roughly 200,000 elements). Fig. 9a illustrates the runtime performance of the *DescFullHashA* operator and the *DescFullHashB* operator for the same query. For selectivities smaller than 10%, the runtime of each operator remains quite the same, because in these cases external memory access costs for the node reference indexes (column sockets) dominate the execution time, whereas the time for the hash table creation and probing remains roughly the same. However for selectivities $>$ 10%, the runtime increases due to higher CPU costs for hashing and probing of larger input sequences. The gap between the `DescFullHashA` and the `DescFullHashB` operator results from hashing the wrong—i. e., the larger—input sequence (*title*) instead of the smaller one (in operator `DescFullHashB`). Therefore, it is important that the query optimizer chooses the right operator for an anticipated data distribution.

4.2 Hash-Based vs. Sort-Based Schemes

In the next test, we want to identify the performance differences of our hash-based schemes as compared to sort-based schemes. For this purpose, we implemented the *StackTree* algorithm [1] and the structural join strategy from [14] called *AxisSort** in the following. Both operators work in two phases: In phase 1, input sequences are sorted using the *QuickSort* algorithm. While *StackTree* needs to sort both input sequences, *AxisSort** only needs to sort the smaller one. In phase 2, *StackTree* accomplishes its ordinary join strategy, while *AxisSort** performs a binary search on the sorted input for each element of the other input sequence. To compare our operators with minimal-cost sort-based schemes, we introduce hypothetical operators which also sort the smaller input sequence, but omit the probing phase. Thus, so-called **Fake* operators do not produce any output tuples. The result comparison is presented in Fig. 9b. Having the same join selectivity dependency, our hash-based operators are approximately twice as fast as the sort-based operators (with result construction). The figures for the *StackTree* algorithm impressively demonstrate that sort operations on intermediate results in query plans should really be avoided if possible. Finally, the hash-based operators—with their "handicap" to produce a result—match the sort-based fake operators.

4.3 Memory Consumption

Finally, we measured the memory consumption of hash-based and sort-based operators. On the generated document collection, we issued the query `//organization[.//andorra]`, where the number of andorra elements varied from 2000 to 200.000, whereas organization elements remained stable (at roughly 200.000). For comparison, we used the `DescFullHashB`[5] and the `DescFullSortB` operator. In all selectivity ranges, the internal hash table of the hash-based operator consumed three to four times more memory than the plain array of the sort-based one. To reduce this gap, a space optimization for hash-based operators is possible: Each key contained in the hash-table (as depicted in Fig. 7a) is repeated (as a prefix) in the join field value of the tuples contained in the key's queue. This redundant information can safely be disposed for a more compact hash table.

In a last experiment, we compare `DescFullHashB` with `AncHashB`. Here, the semi-join alternative required around three times fewer memory than the full join variant on all selectivities. This circumstance is also a strong argument for our proposal, that the query optimizer should pick semi-join operators whenever possible.

5 Conclusions

In this paper, we have considered the improvement of twig pattern queries—a key requirement for XML query evaluation. For this purpose, we have substantially extended the work on structural join algorithms thereby focussing on hashing support. While processing twig patterns, our algorithms, supported by appropriate document store and

[5] Note, regarding the space complexity, `DescFullHashB` is one of the more expensive representative among the hash-based operators (see 3.2).

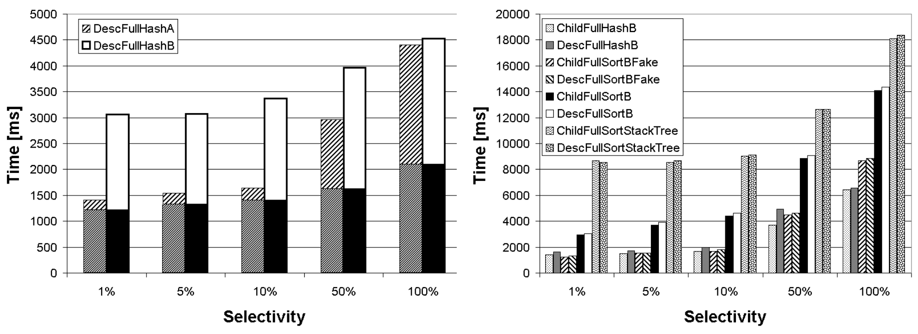


Fig. 9. a) DescFullHash* Characteristics, b) Operator Comparison

index structures, primarily rely on SPLIDs which flexibly enable and improve path processing steps by introducing several new degrees of freedom when designing physical operators for path processing steps.

Performance measurements approved our expectations about hash-based operators. They are, in the selectivity range 1%–100%, twice as fast as sort-based schemes and not slower than the **Fake* operators. As another beneficial aspect, intermediate sorts in QEPs can be drastically reduced. Such hash-based operators should be provided—possibly with other kinds of index-based join operators—in a tool box for the cost-based query optimizer to provide for the best QEP generation in all situations.

References

1. S. Al-Khalifa et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. ICDE: 141-152 (2002)
2. T. Böhme, E. Rahm: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd DIWeb Workshop: 70-81 (2004)
3. N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: optimal XML pattern matching. Proc. SIGMOD: 310-321 (2002)
4. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo: Efficient Structural Joins on Indexed XML Documents. Proc. VLDB: 263-274 (2002)
5. M. Dewey: Dewey Decimal Classification System. http://www.mtsu.edu/ vvesper/dewey.html
6. M. Fernandez, J. Hidders, P. Michiels, J. Simeon, R. Vercammen: Optimizing Sorting and Duplicate Elimination. Proc DEXA: 554-563 (2005).
7. M. Fontoura, V. Josifovski, E. Shekita, B. Yang: Optimizing Cursor Movement in Holistic Twig Joins, Proc. 14th CIKM: 784-791 (2005)
8. G. Gottlob, C. Koch, R. Pichler: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. 30(2): 444-491 (2005)
9. T. Härder, M. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered, accepted for Data & Knowledge Engineering (2006)
10. Q. Li, B. Moon: Indexing and Querying XML Data for Regular Path Expressions. Proc. VLDB: 361-370 (2001)
11. Q. Li, B. Moon: Partition Based Path Join Algorithms for XML Data. Proc. DEXA: 160-170 (2003)
12. P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury: ORDPATHs: Insert-Friendly XML Node Labels. Proc. SIGMOD: 903-908 (2004)
13. I. Tatarinov et al.: Storing and Querying Ordered XML Using a Relational Database System. Proc. SIGMOD: 204-215 (2002)
14. Z. Vagena, M. M. Moro, V. J. Tsotras: Efficient Processing of XML Containment Queries using Partition-Based Schemes. Proc. IDEAS: 161-170 (2004)
15. Y. Wu, J. M. Patel, H. V. Jagadish: Structural Join Order Selection for XML Query Optimization. Proc. ICDE: 443-454 (2003).
16. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohmann, On Supporting Containment Queries in Relational Database Management Systems. Proc. SIGMOD: 425-436 (2001)