Pushing XPath Accelerator to its Limits

Christian Grün Alexander Holupirek Marc Kramis
Marc H. Scholl Marcel Waldvogel
Department of Computer and Information Science
University of Konstanz, Germany
<Firstname>.<Lastname>@uni-konstanz.de

ABSTRACT

Two competing encoding concepts are known to scale well with growing amounts of XML data: XPath Accelerator encoding implemented by MonetDB for in-memory documents and X-Hive's Persistent DOM for on-disk storage. We identified two ways to improve XPath Accelerator and present prototypes for the respective techniques: BaseX boosts in-memory performance with optimized data and value index structures while Idefix introduces native block-oriented persistence with logarithmic update behavior for true scalability, overcoming main-memory constraints.

An easy-to-use Java-based benchmarking framework was developed and used to consistently compare these competing techniques and perform scalability measurements. The established XMark benchmark was applied to all four systems under test. Additional fulltext-sensitive queries against the well-known DBLP database complement the XMark results.

Not only did the latest version of X-Hive finally surprise with good scalability and performance numbers. Also, both BaseX and Idefix hold their promise to push XPath Accelerator to its limits: BaseX efficiently exploits available main memory to speedup XML queries while Idefix surpasses main-memory constraints and rivals the on-disk leadership of X-Hive. The competition between XPath Accelerator and Persistent DOM definitely is relaunched.

1. INTRODUCTION

XML has become the standard for universal exchange of textual data, but it is also increasingly used as a storage format for large volumes of data. One popular and successful approach to store and access XML data is to map document nodes to relational encodings. Based on the XPath Accelerator encoding [11], the integration of the Staircase Join [12], an optimizing to-algebra translation [4], and an underlying relational engine that is tuned to exploit vast main memories, the MonetDB system currently provides unrivalled benchmark performance for large XML documents [3]. This paper describes and evaluates two lines of research aiming at further improvements. We implemented two experimental prototypes, focusing on current weaknesses of the MonetDB system: the lack of value-based indexes and the somewhat less thrilling performance once the system gets I/O-bound. Separate prototypes have been chosen because they allow us to apply independent code optimizations.

Evaluation. The widely referenced XMark benchmark [20] and the well-known DBLP database [16] were chosen for evaluation to get an insight into both the current limitations and the proposed improvements of XPath Accelerator. BaseX is directly compared to MonetDB [2] as both are main-memory based and operate on a similar relational encoding. Idefix is compared to X-Hive [23] as both are mainly based on secondary storage. The main difference to Idefix is that X-Hive stores the XML as a Persistent DOM [13]. Both MonetDB and X-Hive were chosen because current performance comparisons [3] suggest them to be the best available solutions for either in-memory or persistent XML processing. In the course of this work, published results about MonetDB and X-Hive could be revisited. Both BaseX and Idefix strive for outperforming their state-of-the-art competitors.

Contributions. Our main contribution is the evaluation of our approaches against the state-of-the-art competitors, using our own benchmarking framework Perfidix. In more detail: The first prototype, BaseX, pushes in-memory XML processing to its limits by applying thorough optimizations and an index-based approach for processing predicates and nested loops. The second prototype, Idefix, linearly scales XPath Accelerator beyond current memory limitations by efficiently placing it on secondary storage and introducing logarithmic update time[1]. Finally, the use of our benchmarking framework Perfidix assures a consistent and reproducible benchmarking environment.

Outline. The improvements to the XPath Accelerator encoding and its implementation are described by introducing our two prototypes (Section 2). We intensively evaluated and compared the state-of-the-art competitors in the field of XML-aware databases against our optimized prototypes (Section 3). In Section 4, we summarize our findings and have a look at our future work.

2. PUSHING XPATH ACCELERATOR

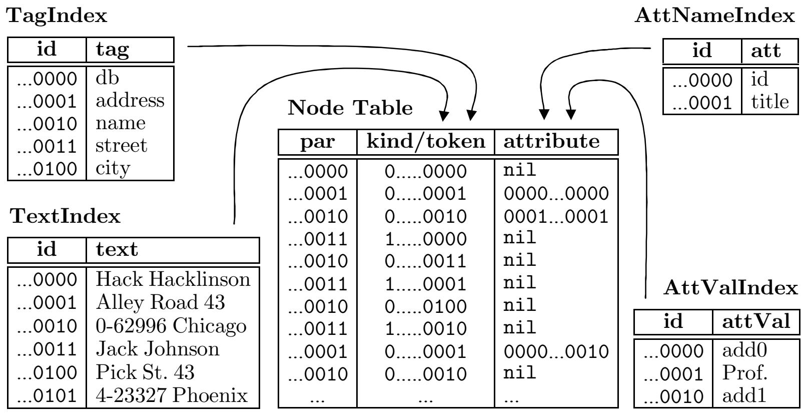
Attracted by the approved and generic XPath Accelerator concept, we were initially driven by two basic questions: how far can we optimize main-memory structures and algorithms

Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006), June 30, 2006, Chicago, Illinois, USA
Copyright 2006 ACM 1-59593-463-4 ...$5.00.

[1] The update performance of Idefix was not yet measured due to the current alpha-level update implementation.

XML Document Mapping

pre	par	token	kind	att	attVal
0	0	db	elem		
1	1	address	elem	id	add0
2	2	name	elem	title	Prof.
3	3	Hack Hacklinson	text		
4	2	street	elem		
5	3	Alley Road 43	text		
6	2	city	elem		
7	3	0-62996 Chicago	text		
8	1	address	elem	id	add1
9	2	name	elem		
10	3	Jack Johnson	text		
11	2	street	elem		
12	3	Pick St. 43	text		
13	2	city	elem		
14	3	4-23327 Phoenix	text		



TagIndex

id	tag
...0000	db
...0001	address
...0010	name
...0011	street
...0100	city

TextIndex

id	text
...0000	Hack Hacklinson
...0001	Alley Road 43
...0010	0-62996 Chicago
...0011	Jack Johnson
...0100	Pick St. 43
...0101	4-23327 Phoenix

Node Table

par	kind/token	attribute
...0000	0.....0000	nil
...0001	0.....0001	0000...0000
...0010	0.....0010	0001...0001
...0011	1.....0000	nil
...0010	0.....0011	nil
...0011	1.....0001	nil
...0010	0.....0100	nil
...0011	1.....0010	nil
...0001	0.....0001	0000...0010
...0010	0.....0010	nil
...	...	...

AttNameIndex

id	att
...0000	id
...0001	title

AttValIndex

id	attVal
...0000	add0
...0001	Prof.
...0010	add1

Figure 1: Relational mapping of an XML document (left), internal table representation in BaseX (right)

to efficiently work with the relational encoding? Next, which data structures are suitable to persistently map the encoding on disk and allow continuous updates? Two different implementations are the result of our considerations: BaseX creates a compact main-memory representation of relationally mapped XML documents and offers a general value index, and Idefix offers a sophisticated native persistent data structure for extending the XML storage to virtually unlimited sizes.

2.1 BaseX – Optimized in-memory processing

BaseX was developed to push pure main memory based XML querying to its limits both in terms of memory consumption and efficient query processing. Main memory is always limited, compared to the size offered by secondary storage media, so we aimed at optimizing the main-memory representation of the XML data to overcome the limitations. Main-memory XQuery processors such as Galax [7] or Saxon [15] work efficiently on small XML files, but querying gets troublesome for larger files as the temporary XML representations occupy 6 to 8 times the size of the original file in main memory. MonetDB [2] is designed as a highly efficient relational main memory database, and its Pathfinder/ XQuery extension [3] applies relational operations to process XML node sets at an amazing speed.

Our first Java prototype can be seen as a hybrid between relational and native XML processing. It uses a relational, edge-based `pre/parent` encoding for XML nodes, but issues arising from the set-oriented approach, such as the orderedness of XML node sets, can be evaded as all the data can be processed sequentially, thus simplifying orderedness for location step traversals. The current implementation can be used both as a real-time XML query tool and as a query application. XPath expressions can be passed on via the command line. Alternatively, an interactive mode is available to directly enter queries which are processed locally or by a BaseX server instance. XML files can be saved as database files to avoid future parsing of the original documents.

Data structures. The system is mainly built on two simple yet very effective data structures that guarantee a compact mapping of XML files: an XML node table and a slim hash index structure. The relational structure is represented in the node table, storing the `pre/parent` references of all XML nodes; the left table of Figure 1 displays a mapping for the XML snippet shown in Figure 2. The table further references the token of a node (which is the tag name or text content) and the node kind (`element` or `text`). Attribute names and values are stored in a two-dimensional array; a `nil` reference is assigned if no attributes are given. All textual tokens – tags, texts, attribute names and values – are uniformly stored in a hash structure and referenced by integers. To optimize CPU processing, the table data is exclusively encoded with integer values. (see Figure 1, right tables).

```
<db>
  <address id='add0'>
    <name title='Prof.'>Hack Hacklinson</name>
    <street>Alley Road 43</street>
    <city>0-62996 Chicago</city>
  </address>
  <address id='add1'>
    <name>Jack Johnson</name>
    <street>Pick St. 43</street>
    <city>4-23327 Phoenix</city>
  </address>
</db>
```

Figure 2: XML snippet, mapped in Figure 1

To save memory, some table values store more than one attribute. Based on the extensive evaluation of numerous large XML instances up to 38 GB, we noted that the maximum values for token references showed out to be constantly smaller than 32 bit, and as the node kind requires only 1 bit, the two values are merged into one integer. The remaining space will be used for additional node information in future. The attribute name and value references are merged as well, sharing 10 and 22 bits, respectively. The original values can be efficiently accessed via CPU-supported bit shifting operators.

The second data structure, the hash index, is basically an implementation of a linked list hash structure. Though, some optimization was done to minimize memory usage and to allow very quick access to the stored tokens. Similar to the table structure, the hash index was flattened to work on simple integer arrays. All arrays are sized by a power of two, and the bitwise AND operator (&) cuts down the calculated hash value to the array size. This approach works faster than conventional modulo (%) operations as the common CPU operators are directly addressed. Moreover the array size does not rely on the calculation of prime numbers, and the index structure can be quickly resized and rehashed during index generation, leading to a quick document shredding process.

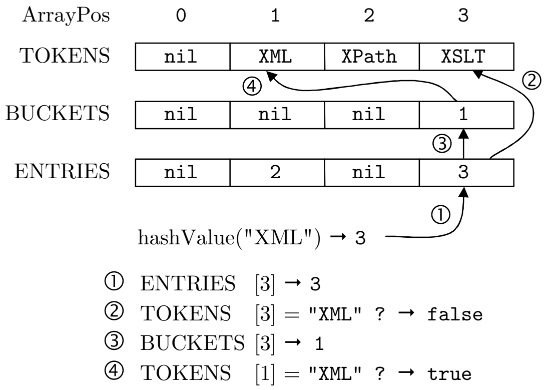


Figure 3: Hash index structure: finding "XML"

Three arrays suffice to organize all hash information (see Figure 3). The TOKENS array references the indexed tokens. The ENTRIES array references the positions of the first tokens, and the BUCKET array maps the linked list to an offset lookup. After a hash value has been calculated for the input token and trimmed to the array size, the ENTRIES array returns the pointer to the TOKENS array. If the pointer is `nil`, no token is stored; otherwise the token is compared to the input. If the comparison fails, the BUCKET array points to the next TOKENS offset or yields `nil` if no more tokens with the same hash value exist.

Querying. The integrated XPath parser evaluates basic XPath queries, including all XPath axes, node tests and basic predicates (with textual, numeric and positional matches). Similar to MonetDB, BaseX is built on a step-based path execution. It has been pointed out that intermediate context sets can get pretty large whereas the result set is often small [5]. This observation is particularly obvious when commonly used and highly selective predicates occur in the query, such as those pointing, e.g., to single attribute ids. To speedup predicate queries, we thus chose to introduce a general value index [17] for all text nodes and attribute values. The index is applied when exact string comparisons are found in the query. Moreover, it is also very effective when nested loops with predicate joins are encountered, reducing the quadratic to a linear execution time without the need of additional algorithms. More detailed information can be found in the evaluation section 3.4.

The value indexes are implemented pretty straightforward. The existing text and attribute value indexes are extended by references to the table's `pre` values, resulting in an inverted list. The correct use of the index is a little bit trickier: as the index returns a node set for the specified predicate, the XPath query has to be inverted and rewritten. Descendant steps are converted into ancestor steps and vice versa, and predicates are moved. In the following query `doc("XMark.xml")/site//item[@id = "item0"]`, `item0` is matched against the attribute value index. The resulting context set is matched against the `item` parent, the `site` ancestor and `doc("XMark.xml")` as the initial context node. The internal query reformulation thus yields `index::node() [@id = "item0"]/parent::item[ancestor::site/parent: :doc()]`. A detailed analysis on rewriting reverse to forward axes can be found in [19].

The Staircase Join [12] offers three basic concepts to speed up path traversal: Pruning, Partitioning and Skipping. The basic ideas behind all three concepts could be modified and applied on the `pre/parent` encoding, thus accelerating step traversals by orders of magnitudes[2]. Depending on the current node set, further speedups are applied. One of them is described in more detail: the Pruning algorithm, which is part of the Staircase Join, removes nodes from a context set that would otherwise be parsed several times and yield duplicate result nodes. For example, if a context node has descendant context nodes, these can be pruned before a descendant step is traversed. But in fact pruning is unnecessary in many cases; prunable descendant nodes never occur when only child location steps are evaluated. Referring to the queries in our performance evaluation, none of the queries actually needs a Pruning of the context set. This is why we added a flag, stating if Pruning is necessary or not.

Thanks to the optimizations, the main-memory representation of an XML file occupies from 0.6 to 2.5 the size of the original document on disk (find examples in Table 1). The included value index just represents 12 to 18% of the main-memory data structure, and XML instances up to 13 GB have been successfully mapped into memory, still allowing efficient querying of the data.

Document	File Size	MM Size	Factor
Treebank	82 MB	182 MB	2.21
XMark	111 MB	142 MB	1.28
DBLP	283 MB	526 MB	1.86
Swissprot	1.43 GB	2.41 GB	1.69
Wikipedia	4.3 GB	5.75 GB	1.34
XMark	11 GB	10.65 GB	0.97

Table 1: Main-memory consumption of BaseX

2.2 Idefix – Native block storage for XPath Accelerator

The trigger for evolving a native block storage for XPath Accelerator was twofold. First, the current reference implementation MonetDB does not scale linearly beyond the main-memory barrier. As soon as the XML data exceeds the available RAM, the performance either degrades exponentially due to extensive swapping, or the database enters an unpredictable state. Second, the latest update functionality introduced with [4] essentially runs in linear time. Linear overhead for `pre` value relabeling is avoided only for page-local modifications. As soon as a whole page must be added or removed, the page index must be updated – an operation which runs in $\mathcal{O}(n)$ time.

Idefix aims at efficiently querying and updating large-scale persistent XML data, e.g., as it would be required

[2] The Staircase Join demands the storage of an additional `post` or `size` value. This is why no exact equivalence can be derived for the exclusive `pre/parent` representation

to map file system metadata to XML. We present a set of index, tuple, and block structures that allow to update XPath Accelerator encodings in $\mathcal{O}(log\ n)$ time while pushing the amount of available XML data beyond current main-memory limits. The trade-off is both the logarithmic cost to lookup a `pre` value and a potential loss of performance due to disk-based I/O.

The prototype demonstrating the feasibility of our ideas is written in Java. A rudimentary storage manager provides access to a block-oriented *64-bit* storage. Idefix currently supports an in-memory block storage for testing purposes and a random-access file-based block storage for benchmarking. To bypass the file system cache of the operating system and gain access to vast amounts of block storage, an iSCSI-based block storage is in the works. The file system cache can not exploit the tree-knowledge found in XPath Accelerator, still occupies memory and blurs potential scalability measurements because smaller XML data sets might be fully cached whereas larger XML data sets might not.

Block allocation is handled similar to XFS [22]. Two B+ Trees [10] support the dynamic allocation of single blocks or extents (multiple contiguous blocks) close to a requested address. The storage manager currently implements a simple LRU block buffer. Recent caching algorithms such as temporal-spatial caches [14, 9] could be plugged-in if required.

Index Structures. Idefix employs two well-known block-based index structures to map *64-bit* keys or positions to tuple addresses consisting of a *48-bit* block address and a *16-bit* block offset. Keys are immutable, unique, in dense ascending order, and generated by a persistent sequence as it is commonly found in database systems. Positions are volatile in the sense that they might reference different tuples over time due to updates. Note that there currently are no fulltext, path, or value index structures.

The *Positional B+ Tree [21]* is a slight modification of a B+ Tree. B+ Trees store the key range contained in each child node. In contrast, positional B+ Trees store the number of leaf values contained in the whole subtree of each child. This allows to access any element of the index by position and in logarithmic time. Updates potentially trigger expensive rebalancing operations.

The *Trie [6]* is a dense distribution of unique keys as they are frequently occurring in Idefix. This specific key distribution allows for an index structure that does not require rebalancing. A set of hierarchical arrays can efficiently be queried and updated in logarithmic time because the array (i.e., block) offset of each level can be precomputed.

A third index structure appearing in Idefix, the *Hash Map*, is only held in main memory to speed up certain operations and can be reconstructed from a trie-based index structure at any time.

Tuple Structures. Figure 4 shows the core tuple and index structures of Idefix. XPath Accelerator is persistently stored in the node list. Each XML node is stored as a node tuple (see Table 2) at the node-list-position equal to the `pre` value. Names and values are offloaded from the node list and separately stored as name tuples (Figure 3) in the name map and value tuples (Figure 4) in the value map respectively.

The offloading of strings has four advantages. First, a very tight packaging of the frequently accessed node list results in fewer I/Os. Second, the name map can be kept in memory due to its small size even for very large XML data [18], leading to constant-time name-to-reference resolution. Third, filtering of node tuples according to a name can be reduced to a fast reference (integer) comparison. Fourth, the reference is usually much smaller than the string. A disadvantage is the additional cost of retrieving a value due to the additional mapping and potentially distant block address of the value tuple.

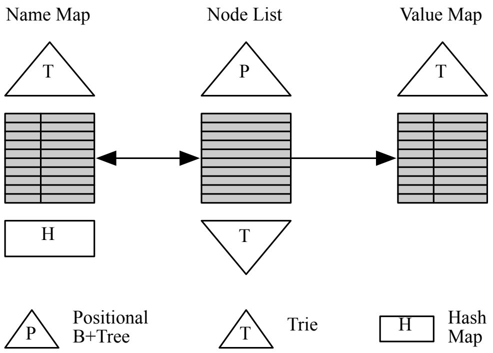


Figure 4: Core tuple & index structures of Idefix

The ancestor axis is supported by an immediate reference to the parent element. The Staircase Join will therefore have to mix keyed and positional access. Since the absolute position is lost after a keyed access, the Staircase Join must always work with relative positions[3]. The attribute count is stored to quickly skip attributes if they are not required for evaluation. Nevertheless, attribute nodes are kept close to the corresponding element nodes to streamline attribute-related evaluations.

Field	Bytes	D	E	A	T
kind	1	x	x	x	x
key	1..9	x	x	-	-
parentKey	1..9	x	x	-	-
size	8	x	x	0	0
level	1	x	x	x	x
attributeCount	1	-	x	-	-
nameReference	1..5	x	x	x	-
valueReference	8	-	-	x	x

Table 2: Node tuple stored in node list. The following kinds are currently stored (denoted with 'x') for an XML node: (D)ocument, (E)lement, (A)ttribute, and (T)ext. Variable-length encodings are denoted with '..'. '0' is a constant unstored zero. '-' means not stored.

Node Tuple (Table 2). A node tuple can be accessed both by position and by key. Positional access is provided by a positional B+ Tree. Note that a positional B+ Tree does not suffer from the linear-time relabeling of `pre` values required after an update and hence offers logarithmic update behavior. Positional access is required for the Staircase Join that basically operates on `pre` values, i.e., positions. Keyed access is provided by a trie. It is required to support future

[3] A reverse access path to find the absolute position will be investigated.

Field	Bytes	Description
count	8	# of occurrences
name	..	UTF-8-encoded String

Table 3: Name tuple stored in name map

Field	Bytes	Description
value	..	UTF-8-encoded String

Table 4: Value tuple stored in value map

index structures (such as a fulltext index) that reference specific node tuples and must not lose their context after an update.

Name Tuple (Table 3). A name tuple is accessed both by name and by a key stored with the node tuple. The (reverse) mapping between name and key is achieved by a hash map. This access path is required to maintain the counter (i.e., the number of occurrences of the name in the stored XML data) assigned to each name and to efficiently filter node tuples by their name. The mapping between key and name tuple is done by a trie and required whenever the name of a node tuple must be resolved.

Value Tuple (Table 4). A value tuple is accessed by a key stored with the node tuple. The mapping between key and value tuple is done by a trie and required whenever the value of a node tuple must be resolved.

Block Structures. Figure 5 shows the node, name, and value block layouts. The first block shows a name or value block containing two name or value tuples. The next three blocks show an empty node block that is updated by adding two new node tuples and finally has the first node tuple removed.

Node Block. Node tuples must be locatable through two different access paths. A keyed access will always yield a node block address with an immediate node block offset where the node tuple is stored. A positional access, in contrast, will return a node block address with a position relative to the block node. This position is locally resolved to an immediate node block offset with the help of an intra-node-block directory. This directory is always kept ordered by the `pre` axis and grows bottom-up according to the number of node tuples in this node block. New node tuples are appended to the last allocated node block if it still has enough contiguous space between the directory and the last inserted node tuple. If it is full, a new node block is allocated. The insertion or deletion of one node tuple affects both the positional B+ Tree as well as the trie. Due to updates, a node block containing node tuples might be split or merged with another node block. This operation has an upper bound of $\mathcal{O}(2m \log n)$, which yields $\mathcal{O}(\log n)$; m is the number of affected node tuples. Defragmentation of node blocks from which node tuples are deleted is postponed to the next merge or split operation.

Name Block. Name tuples are written to disk like to a content-addressable storage, i.e., each name is only stored once. If a name is added, the hash map is looked up for a key associated with it. If it is found, the name tuple is located and the counter increased. The trie therefore maps the key to the name block address and an immediate name block offset where the name tuple is stored. If the name has not been stored before, the name tuple is appended to the last allocated name block if it fits. As soon as the last name block is full, a new name block is allocated. A modified name is treated like a new name. Deleting a name results in a decrease of the counter. Name tuples with zero occurrence remain accessible to support versioning. A garbage collector could free unreferenced names from time to time if required.

Value Block. Value tuples are written to disk in a log-structured fashion. If a new value tuple fits into the last allocated value block, it is appended there. If the last value block is full, a new value block is allocated. The trie maps the new key to this value block address and the immediate value block offset where the value tuple was written. The XML shredder assures that a value tuple is not bigger than a single value block by splitting larger text nodes into multiple smaller text nodes. This internal fragmentation of text nodes is not visible to the upper layers in order not to break the XPath or XQuery Data Model. A modified value is treated like a new value. Deleted values again remain accessible to support versioning.

API. IDEFIX currently offers a low-level API based on a cursor pointing to a node tuple. The cursor can be moved to an absolute or relative position as well as to a given key. The cursor can retrieve all fields after locating the appropriate node tuple. Name or value strings are lazily fetched from the name and value map. New node tuples can be appended to the end of the node list.

On a higher level, a rudimentary API allows to execute a Staircase Join for a given context set on the descendant axis including simple predicate evaluation. The resulting context set consisting of ordered `pre` values can either be passed to the next Staircase Join to implement a simple XPath evaluation or materialized by fetching all fields of each context node with the cursor.

3. PERFORMANCE EVALUATION

3.1 Perfidix – Evaluation framework

We used our own Java-based benchmarking framework PERFIDIX to guarantee and facilitate a consistent evaluation of all tested systems. The framework was initially inspired by the unit testing tool JUnit [8]; it allows to repeatedly measure execution times and other events of interest (e.g., cache hits). The results are aggregated, and average, minimum, maximum, and confidence intervals are collected for each run of the benchmark.

A sample output for the IDEFIX 11MB XMark benchmark is shown in Table 5. The whole benchmark was implemented in one Java class and each Query as a Java method. PERFIDIX was configured to run the benchmark 5 times and only

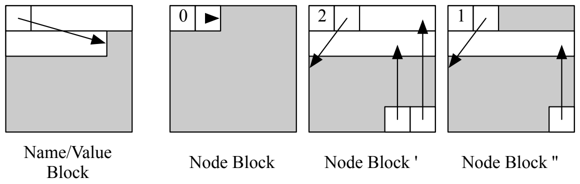


Figure 5: Name/Value & Node block layout

Method	Unit	Sum	Min	Max	Avg	StdDev	Conf95	Runs
Query 1	ms	248	12	172	49.60	61.37	[00.00, 103.40]	5
Query 2	ms	312	49	103	62.40	20.37	[44.54, 80.26]	5
...	...	...	...	...	...	...	...	...
Query 20	ms	215	41	48	...	...	...	...
Summary	ms	17595	3228	4317	...	...	...	...

Table 5: Sample output for Idefix 11MB XMark benchmark run by Perfidix.

measure execution times of each method.

3.2 Test Data Sets

Our tests are based on the scalable XMark Benchmark [20] and the XML version of the DBLP database [16][4]. We formulated six mainly content-oriented XPath queries for the DBLP data, yielding small result sets (see Table 8), and implemented the predefined XMark queries in our prototypes. Detailed information on the queries can be found in [20]. Single queries will be described in more detail whenever it seems helpful to understand the test results.

3.3 Process of measurement

Instead of splitting up query processing into single execution steps, our test results represent the systems' overall query execution times, including the compilation of queries and the result serialization into a file. As all four systems use different strategies for parsing and evaluating queries, a splitting of the execution time would have led to inconsistent results. Table 6 lists the methodology of execution time measurements for each system.

System	Compile	Execute	Serialize
MonetDB	x	x	x
BaseX	hard-coded	x	x
X-Hive	x	x	x
Idefix	hard-coded	x	x

Table 6: Methodology of execution time measurements. 'x' means included in overall execution time.

The hard-coded query execution plans of BaseX and Idefix do not allow to include the time for parsing, compiling and optimizing the queries, but we intend to extend the MonetDB engine to produce query execution plans for BaseX and Idefix. Meanwhile, the hard-coded query execution plans are carefully implemented based on the API of each prototype to follow automated patterns and avoid ”smart” optimizations that can not easily be detected and translated by a query compiler.

To get better insight into the general performance of each system, we run each query several times and evaluated the average execution time. As especially small XML documents are exposed to system- and cache-specific deviations, we used different number of runs for each XML instance. The number of executions is listed in Table 7.

Document	Size	No. Runs
XMark	110 KB	100
XMark	1 MB	50
XMark	11 MB	10
XMark	111 MB	5
XMark	1 GB	5
XMark	11 GB	1
XMark	22 GB	1
DBLP	283 MB	5

Table 7: Query execution times

Queries were excluded from the test when the execution time was expected to take more than 24 hours (1GB XMark Query 8, 9, 11 and 12 for X-Hive) or yielded error messages (11GB XMark Query 11 and 12 for MonetDB). All test runs were independently repeated several times with a cold cache; this was especially important for the 11GB XMark instance which was only tested once at a time.

All tests were performed on a *64-bit* system with a 2.2 GHz Opteron processor, 16 GB RAM and SuSE Linux Professional 10.0 with kernel version 2.6.13-15.8-smp as operating system. Two separate discs were used in our setup (each formatted with ext2). The first contained the system data (OS and the test candidates), the second the XMark and DBLP input data and the internal representations of the shredded documents for each system. The query results were written to the second disc. We used MonetDB 4.10.2 and X-Hive 7.2.2 for testing.

We compare the in-memory BaseX with MonetDB to analyse the impact of the optimizations applied with BaseX. Then we compare the disk-based Idefix with X-Hive to confront natively persistent XPath Accelerator to Persistent DOM. Finally, we compare Idefix with MonetDB to verify our assumptions about the difference of an in-memory and a disk-based system. The comparison of Idefix and BaseX is not explicitly mentioned but can be deduced from the presented figures and analysis.

3.4 Performance Analysis

Figure 6 gives an overview on the scalability of each system and the average execution times of all XMark queries and XML instances. First observations can be derived here: MonetDB and BaseX can both parse XML instances up to 11 GB whereas Idefix and X-Hive could easily read and parse the 22 GB instance. The execution times for the 11GB document increase for Idefix and MonetDB as some queries generate a heavy memory load and fragmentation. The most obvious deviations in query execution time can be noted for the Queries 8 to 12, which all contain nested loops; details on the figures are following.

An aggregation of the 20 XMark queries is shown in Figure 7, summarizing the logarithmic values of all XMark query averages. All systems seem to generally scale linearly on the tested queries. MonetDB consumes an average time of 38 ms to answer a query, mainly because in-

[4] State: 2005/12/12, 283 MB

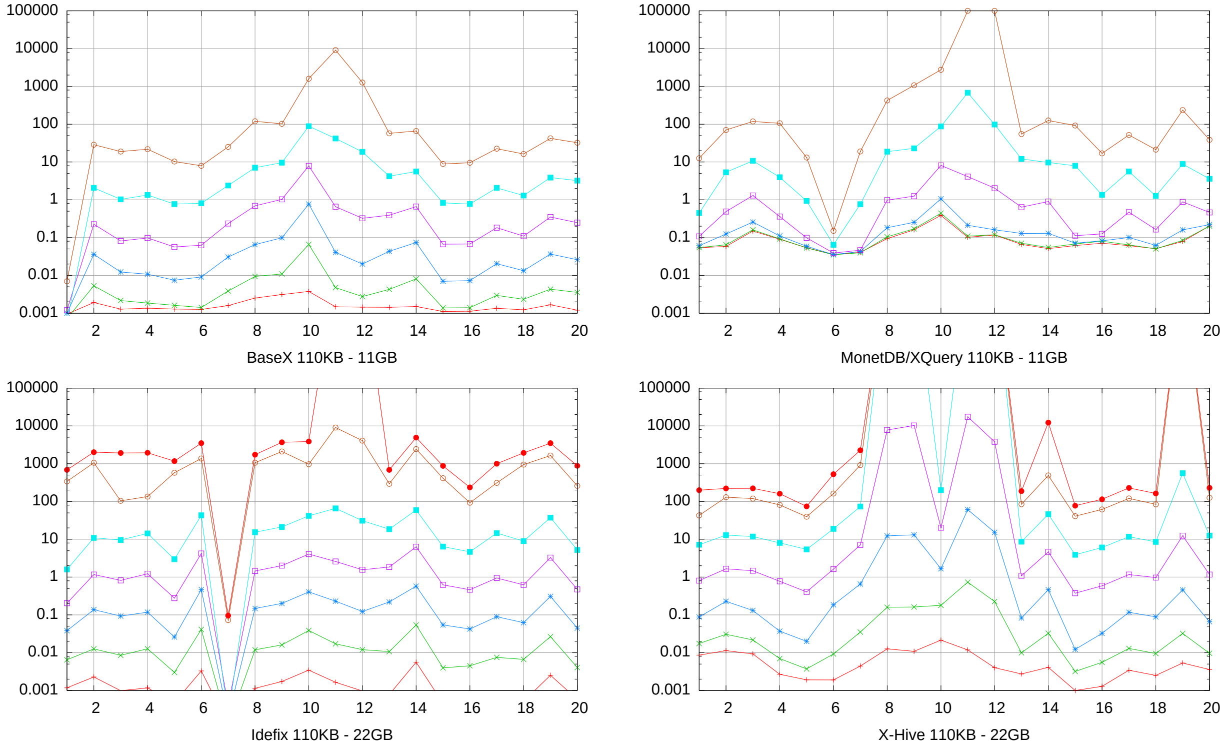


Figure 6: Scalability of the tested systems (x-axis: XMark query number, y-axis: time in sec)

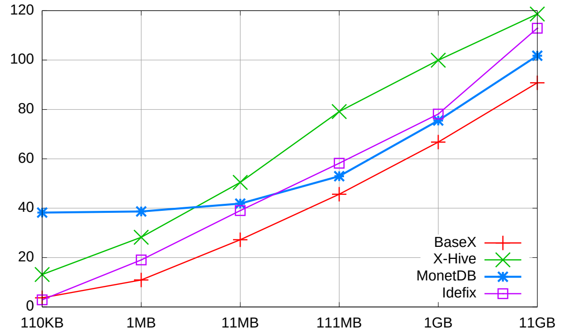


Figure 7: Logarithmic Aggregation of all XMark queries (y-Axis: $\sum_{i=1}^{20} log(avg_i)$ in ms)

put queries are first compiled into an internal representation and then transformed into the MonetDB-specific MIL language. Though, the elaborated compilation process pays off for complex queries and larger document sizes.

BaseX and MonetDB – XMark. Obviously, BaseX yields best results for Query 1, in which an exact attribute match is requested. The general value index guarantees query times less than 10 ms, even for the 11 GB document. MonetDB is especially good at answering Query 6, requesting the number of descendant steps of a specific tag name. MonetDB seems to allow a simple counter lookup, thus avoiding a full step traversal.

The top of Figure 8 relates the query times for MonetDB and BaseX and the XMark query instances 111MB, 1GB, and 11GB. For the 111MB instance, BaseX shows slightly better execution times than MonetDB on average which is partially due to the fast query compilation and serialization. The most distinct deviation can be observed for Query 3 in which the evaluation time of position predicates is focused. Results for Query 8 and 9 are similar for both systems, although the underlying algorithms are completely different. While MonetDB uses loop-lifting to dissolve nested loops [3], BaseX applies the integrated value index, shrinking the predicate join to linear complexity. The respective results for the 1 GB and 11 GB instance underline the efficiency of both strategies.

The Queries 13 to 20 yield quite comparable results for MonetDB and BaseX. This can be explained by the somewhat similar architecture of both systems. The most expensive Queries are 11 and 12, which both contain a nested loop and an arithmetic predicate, comparing double values. To avoid repeated string-to-double conversions, we chose to extract all double values before the predicate is actually processed. This approach might still be rethought and generalized if a complete XQuery parser is to be implemented.

Idefix and X-Hive – XMark. Both Idefix and X-Hive store the XML data on disk and consume only a limited amount of memory for the buffer. They differ however in the handling of query results. Idefix materializes all intermediate query results in main memory and only writes them to disk after successfully executing the query. This results in the extensive alluded memory consumption and fragmentation for XML instances bigger than 1GB. X-Hive immediately writes the results to disk.

We expected the systems to show query execution times in the same order of magnitude as both are essentially disk

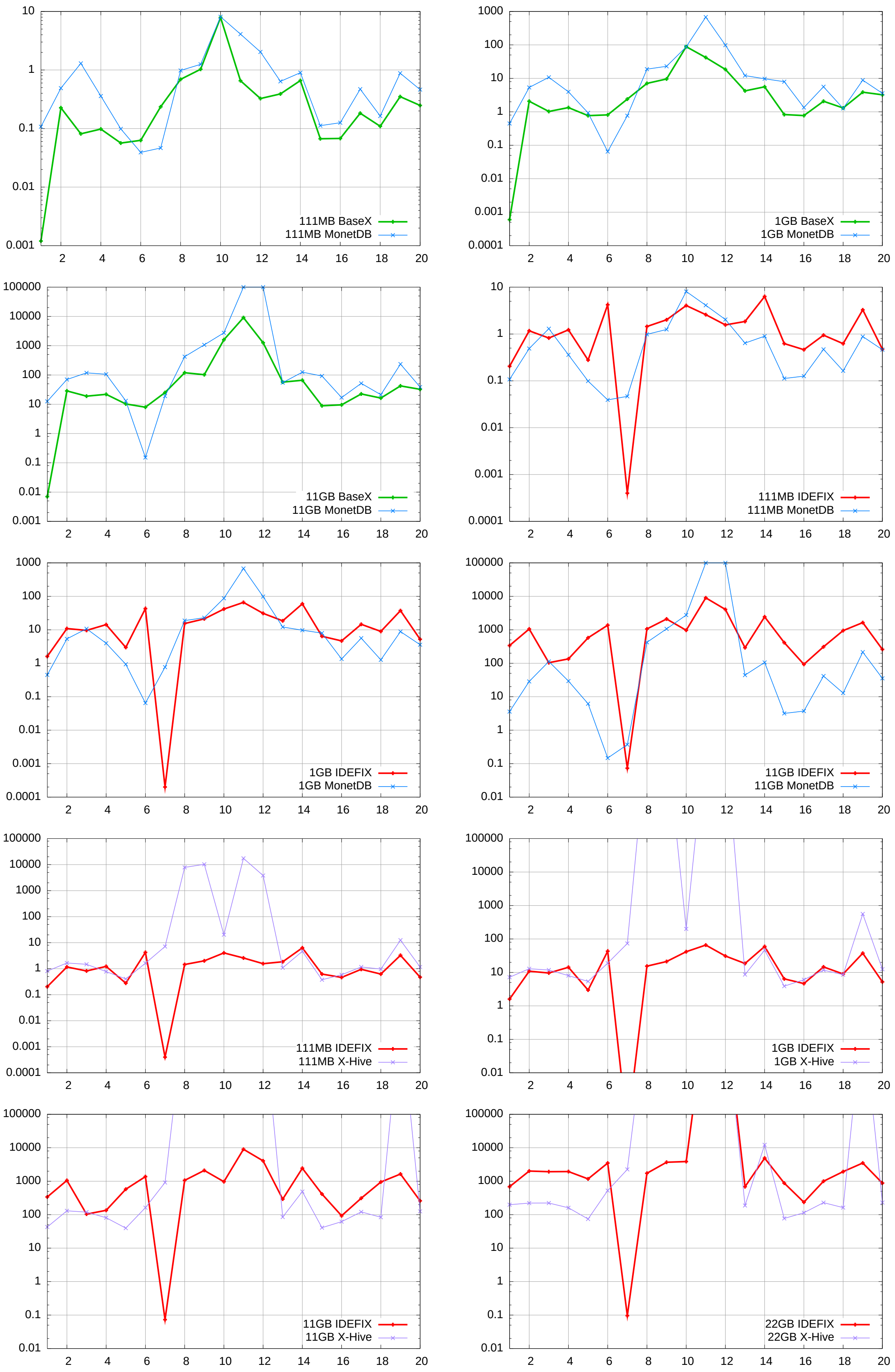


Figure 8: Comparison of the candidates (x-axis: XMark Query number, y-axis: time in sec)

No	Query	Hits
1	/dblp/article[author/text() = ’Alan M. Turing’]	5
2	//inproceedings[author/text() = ’Jim Gray’]/title	52
3	//article[author/text() = ’Donald D. Chamberlin’][contains(title, ’XQuery’)]	1
4	/dblp/*[contains(title, ’XPath’)]	113
5	/dblp/*[year/text() < 1940]/title	55
6	/dblp//inproceedings[contains(@key, ’/edbt/’)][year/text() = 2004]	62

Table 8: DBLP queries

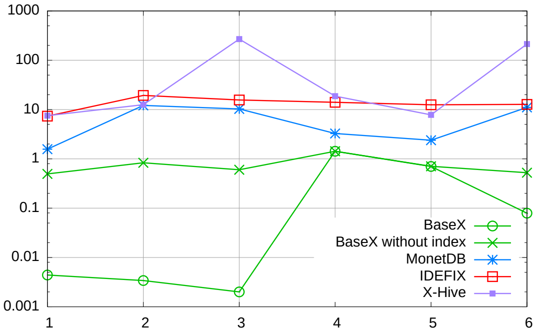


Figure 9: DBLP Execution Times (x-axis: DBLP Query number, y-axis: time in sec)

bound. The effects of block allocation, buffer management, and the maturity of the implementation (optimizations) as well as the different materialization strategy were assumed to have a minor impact. Idefix was expected to have a small constant advantage because the queries are not compiled and a disadvantage because it currently is fully synchronous, i.e., disk and CPU intensive phases do not overlap and hence waste resources. Both systems are equal with respect to value or fulltext indexes: while Idefix does currently not support them, X-Hive was configured not to create them.

In Figure 8 (111MB and 1GB Idefix and X-Hive), our expectations are only partially confirmed. With Query 7, Idefix can take advantage of the name occurrence counter (comparable to a reduced variant of the MonetDB path summary) and quickly calculates the count of a specific element name. Note that this would take longer if the count would be applied on a subtree, starting at a higher level. Queries 8 to 12 also clearly deviate. Here, the execution time of the query is no longer simply disk-bound but dependent on the decision of the query compiler and execution plan. Idefix uses a hash-based join by default whereas X-Hive probably executes a cross-join or another unapt join variant. Note that Idefix does not estimate the sizes of the sets to join but just picks the one first mentioned in the query.

The memory allocation and fragmentation resulting from the full in-memory materialization of intermediate results with Idefix has an unexpected though major impact on the overall performance which is also aggravated by the garbage collection of Java. The linear scalability beyond 1GB is therefore slightly impaired. An immediate lesson learned is to switch to the same serialization strategy as it is employed with X-Hive.

Idefix and MonetDB – XMark. We consider MonetDB to be the reference XMark benchmark in-memory implementation. We expected Idefix to perform an order of magnitude slower because it is disk-bound. The results confirm the expectations and only step out of line for the Queries 7 (see discussion of Idefix and X-Hive) and 10 to 12. In Figure 8 the plots for 111MB, 1GB, and 11GB for Idefix and MonetDB consistently show a surprising result for the queries 10 to 12 where MonetDB performs worse than Idefix. The same argument, i.e., the query execution plan, applies as with X-Hive. The main-memory consumption and fragmentation of MonetDB to materialize intermediate results is a supporting argument.

BaseX – DBLP. The query performance for BaseX was measured twice, with the value index included and excluded. The results for BaseX and the Query 4 and 5 are similar for all tests as the `contains()` function demands a full text browsing for all context nodes. The index version of BaseX wins by orders of magnitude for the remaining queries as the specified predicate text contents and attribute values can directly be accessed by the index. The creation of large intermediate result sets can thus be avoided, and path traversal is reduced to a relatively small set of context nodes.

Idefix – DBLP. Figure 9 summarizes the average execution times for the queries on the DBLP document. The lack of a value and fulltext index forces Idefix to scan large parts of its value map to find the requested XML node. This holds for all six queries and results in near-constant query execution time for all DBLP Queries.

4. CONCLUSION AND FUTURE WORK

In this paper we presented two improvements backed with their corresponding prototypes to tackle identified shortcomings with XPath Accelerator. The performance and scalability of these two prototypes were measured and compared to state-of-the-art implementations with a new benchmarking framework.

The current prototype of BaseX outperforms MonetDB. The tighter in-memory packaging of data as well as the value index structure stand the test. Though, the presented evaluation times are still subject to change as soon as a more elaborated query compilation is performed. Idefix can efficiently evaluate XML instances beyond the main-memory barrier. The cost of making XPath Accelerator persistent is paid-off for larger XML data sets. Idefix introduces logarithmic overhead to locate a node tuple by position or by key. This overhead is negligible compared to the overhead resulting from disk I/O and largely compensated by caches. We regard our paradigm shift away from constant time array lookups, as found in in-memory XPath Accelerator implementations, a good trade-off with disk-based implementations because it allows superior update behavior and scalability.

The Persistent DOM concept as implemented with X-Hive shows a scalability and query performance comparable to XPath Accelerator. This is a surprising result since former versions of X-Hive did not even scale beyond the 1GB XMark document [3].

Future Work. BaseX still lacks features such as the support of several documents, namespaces or the storage of some specific XML data, including comments or processing instructions. Node information of this kind will lead to an extension of the existing data structure. Moreover no schema information is evaluated yet, and some effort has still to be invested in implementing or integrating a complete XPath and XQuery implementation.

However, the code framework was carefully designed to meet the requirements of the XPath and XQuery specifications. A major focus will next be set on further indexing issues, supporting range, partial and approximate searches. Besides, some effort will be put on fulltext-search capabilities, expanding the prototype to support flexible fulltext operations as proposed by the W3C [1].

Idefix. The update functionality will next be fully implemented and benchmarked. In the near future, we intend to add a fulltext index as well as an XQuery-to-algebra compiler based on the engine of MonetDB. Further work will explore pre-fetching, caching, pipelining, and schema-aware techniques to exploit the Staircase Join-inherent knowledge about the XML data to minimize disk touches while maximizing CPU utilization. Another domain will be the distribution of XPath Accelerator across multiple nodes to further increase the performance and scalability.

Perfidix. Initially developed as a simple benchmarking framework to avoid error-prone repetitive manual tasks, Perfidix will be integrated into a development environment as well as enriched with a chart generator. The achieved progress can then constantly be tracked while the code is being optimized or new concepts and ideas are introduced.

Finally, we will investigate how BaseX and Idefix can be conceptionally merged to streamline our research efforts and profit from both improvements.

5. ACKNOWLEDGMENTS

We would like to thank Daniel Butnaru, Xuan Moc, and Alexander Onea contributing many hours of coding to bring the prototypes and Perfidix up and running. Moreover we thank our anonymous reviewers for useful comments and suggestions. Alexander Holupirek is supported by the DFG Research Training Group GK-1042 *Explorative Analysis and Visualization of Large Information Spaces*.

6. REFERENCES

[1] S. Amer-Yahia, C. Botev, et al. XQuery 1.0 and XPath 2.0 Full-Text. Technical report, World Wide Web Consortium, May 2006.

[2] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[3] P. Boncz, T. Grust, et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of ACM SIGMOD/PODS Int'l Conference on Management of Data/Principles of Database Systems*, Chicago, IL, USA, June 2006.

[4] P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proceedings of 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, Baltimore, Maryland, USA, June 2005.

[5] N. Bruno, N. Koudas, et al. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of ACM SIGMOD/PODS Int'l Conference on Management of Data/Principles of Database Systems*, pages 310–321, Madison, Wisconsin, USA, June 2002.

[6] R. de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of Western Joint Computing Conference*, pages 295–298, 1959.

[7] M. Fernández, J. Siméon, et al. Implementing XQuery 1.0: The Galax Experience. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[8] E. Gamma and K. Beck. JUnit – A Regression Testing Framework. http://www.junit.org/.

[9] B. S. Gill and D. S. Modha. WOW: Wise Ordering for Writes - Combining Spatial and Temporal Locality in Non-Volatile Caches. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, Dec. 2005.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[11] T. Grust. Accelerating XPath Location Steps. In *Proc. of ACM SIGMOD/PODS Int'l Conference on Management of Data/Principles of Database Systems*, pages 109–120, Madison, Wisconsin, USA, June 2002.

[12] T. Grust, M. v. Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 524–525, Berlin, Germany, Sept. 2003.

[13] G. Huck, I. Macherius, and P. Fankhauser. PDOM: Lightweight Persistency Support for the Document Object Model. German National Research Center for Information Technology, Integrated Publication and Information Systems Institute, 1999.

[14] S. Jiang, X. Ding, et al. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, Dec. 2005.

[15] M. Kay. SAXON – The XSLT and XQuery Processor. http://saxon.sourceforge.net/.

[16] M. Ley. DBLP – Digital Bibliography & Library Project. http://www.informatik.uni-trier.de/∼ley/db/.

[17] J. McHugh and J. Widom. Query Optimization for XML. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 315–326, Edinburgh, Scotland, UK, Sept. 1999.

[18] M. Nicola and B. v. d. Linden. Native XML Support in DB2 Universal Database. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 1164–1174, Trondheim, Norway, Aug. 2005.

[19] D. Olteanu, H. Meuss, et al. XPath: Looking Forward. In *EDBT Workshops*, pages 109–127, Prague, Czech Republic, Mar. 2002.

[20] A. R. Schmidt, F. Waas, et al. XMark: A Benchmark for XML Data Management. In *Proc. of Int'l Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, Aug. 2002.

[21] S. Tatham. Counted B-Trees. http://www.chiark.greenend.org.uk/∼sgtatham.

[22] R. Y. Wang and T. E. Anderson. xFS: A Wide Area Mass Storage File System. Technical Report UCB/CSD-93-783, EECS Department, University of California, Berkeley, 1993.

[23] X-Hive Corporation. X-Hive DB Version 7.2.2. http://www.xhive.com/.