XQuery Full Text Implementation in BaseX

Christian Grün, Sebastian Gath, Alexander Holupirek, and Marc H. Scholl

`<firstname>.<lastname>@uni-konstanz.de`
Department of Computer & Information Science
Box D 188, 78457 Konstanz, Germany
University of Konstanz

Abstract. LONG PAPER. BaseX is an early adopter of the upcoming XQuery Full Text Recommendation. This paper presents some of the enhancements made to the XML database to fully support the language extensions. The system's data and index structures are described, and implementation details are given on the XQuery compiler, which supports sequential scanning, index-based, and hybrid processing of full-text queries. Experimental analysis and an insight into visual result presentation of query results conclude the presentation.

1 Introduction

XML has been widely adopted as an exchange and storage format for textual data in both research and industry. The existence of more than fifty XQuery processors clearly underlines the large interest in querying XML documents and collections. While many of the database-driven implementations offer their own extensions to support full-text requests, the upcoming XPath and XQuery Full Text 1.0 Recommendation [1] will satisfy the need for a unified language extension and will most probably attract more developers and users from the Information Retrieval community. The recommendation offers a wide range of content-based query operations, classical retrieval tools such as Stemming and Thesaurus support, and an implementation-defined scoring model that allows developers to adapt their database to a large variety of use-cases and scenarios.

In this paper, we present aspects of the implementation of XQuery Full Text in the database system BaseX [14, 15, 17]. GalaTex [7] and Quark [4] were two systems that supported early versions of the proposal, and BaseX is, to the best of our knowledge, the first implementation to fully support all features of the specification. More implementations are expected to follow in the near future as soon as the recommendation has reached its final state.

A simple full-text test looks nearly the same as a General Comparison in XQuery [5]. An `ftcontains` expression can get pretty large, however, if the right-hand side is extended by match options, positional filters or logical connectives:

```
/library/book[content ftcontains ("biogenetics" ftor
("biology" ftand "genetics" ordered distance at most 5 words))
language 'en' with stemming with thesaurus default]
```

Due to the complexity of the language extension, this paper will focus on its core features. Special attention will be given to the discussion of different execution plans. As full-text requests heavily depend on index structures, the query compiler will try to use a full-text index whenever possible. If this strategy fails, a sequential approach is chosen. A third, hybrid variant takes advantage of the index, but processes all XML nodes sequentially.

While iterative query processing (streaming) adds some overhead to simple database operations, it clearly wins when large intermediate and small final result sets are to be expected. As all XQuery expressions in BaseX are implemented in an iterative manner, the iterative approach was not only maintained for all full-text operators, but even pushed down to the index methods and structures. This way, execution times for small results will not suffer from bulky index results.

Many full-text queries produce large result sets with long textual contents. Since, from the beginning, BaseX supported visual access to data and query results, the graphical frontend was extended to meet the demand of visualizing large text bodies and results in a compact way.

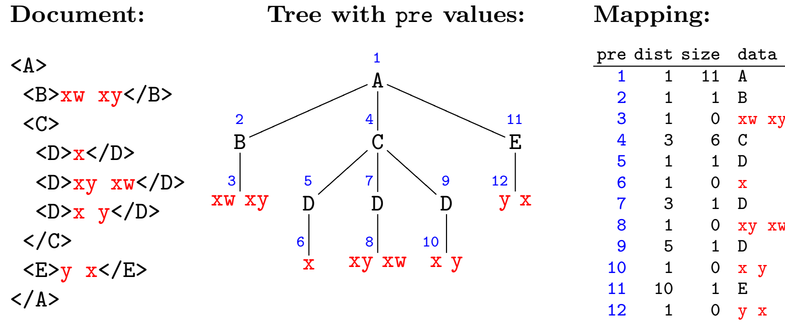
The paper is organized as follows: Section 2 presents the storage and index structures that allow for efficient query evaluation. The sequential, index-based and hybrid execution strategies are discussed in Section 3, and details on iterative query evaluation are given in Section 4. Some performance results in Section 5 analyze execution times of the evaluation variants. Section 6 gives insight into the visual presentation of full-text results; it is concluded by the summary in Section 7.

2 Database Architecture

2.1 Document Storage

While many different XML storage models have been discussed over the last ten years—and none of them has superseded the others—the Pre/Post encoding and its variants have proven to generally yield good performance. It was introduced by Grust [16] and successfully applied by the MonetDB/XQuery implementation [6]. Several variations of this encoding can be used to faithfully represent the XML structure. In MonetDB, for example, XML nodes are mapped to a `pre/size/level` triple. The attributes represent a node identifier, the number of descendant nodes and the depth of a node inside the document tree.

As shown in Figure 1, BaseX stores a `pre/dist/size` combination for each node. The `size` attribute is mainly used to speed up child and descendant traversals, whereas `dist` contains the distance to the parent node, allowing access to the parents and ancestors of a node in constant time. As we will see later, index-based queries benefit greatly from fast access to ancestor nodes. A relative parent encoding (the distance) was favored over an absolute reference as it has shown to be update-invariant, i.e., sub-trees keep their original distance values if they are moved to another place or inserted in a new document.



pre	dist	size	data
1	1	11	A
2	1	1	B
3	1	0	xw xy
4	3	6	C
5	1	1	D
6	1	0	x
7	3	1	D
8	1	0	xy xw
9	5	1	D
10	1	0	x y
11	10	1	E
12	1	0	y x

Fig. 1. Document Encoding in BaseX

The main advantage of a flat storage of XML documents is that documents can be sequentially parsed—a property that is particularly useful if many subsequent nodes have to be accessed, which is the case, e.g., for traversals of the `descendant` step. Next to that, the final table contains no variable-sized entries. As tags and attribute names are indexed and texts and attribute values are separately stored, tuples can be stored with a fixed size, and the memory/disk offset of XML nodes can be calculated easily and accessed in constant time [14].

A closer look at the table attributes reveals some specific properties for each node kind (element, attribute, text, etc.):

- the `size` value of text and attribute nodes will always be `0`
- the number of distinct tag and attribute names is much smaller than the number of document nodes
- as elements have a limited number of attributes, the `dist` value of attribute nodes is small
- attributes, however, consist of two values (attribute name and value)

Based on these and some other observations, the storage of XML node tuples can be compacted. This compression procedure further speeds up node access by minimizing the tuple sizes.

The presented storage was simplified for the sake of clarity. The actual storage model includes some other data structures, such as a directory to reference the first `pre` values of the disk-based table blocks [14]. This extension is needed to support update operations on the storage. The general access time, however, is not affected by the extension. To get even better performance, the database table can be completely kept in main-memory—a feature which is obviously limited by the amount of available memory.

2.2 Index Structures

The presented storage is extended by a number of index structures. **Name indexes** convert variable-sized tag and attribute names as well as namespaces to fixed-size numeric references. An additional **path summary** maintains information on all distinct location paths in an XML document [3, 12]. Both indexes are

enriched by statistical data (number of occurrences, minimum and maximum values of attached text nodes/attribute values), which are interpreted by the query optimizer, as shown in Section 3. **Value indexes** reference all text nodes and attribute values of a document. They are used to speed up content-based queries. A classical example for the application of a value index is the combination of a location path filtered by an equality predicate: `/A/C[D = "x"]`. Query evaluation can be skipped at an early stage, if a value index indicates that a query will yield zero hits. Among others, the attribute index is beneficial to evaluate the XQuery *fn:id()* and *fn:idref()* functions.

To capture the challenges of XQuery Full Text, all text nodes are tokenized, normalized and stored in an additional **full-text index**. The tokenization process is further specified in Section 4.1 of the language specification [1]. Normalization includes the removal of diacritics, a case insensitive representation, optional stemming, etc. A Compressed Trie [2, 10] was implemented that, apart from simple token requests, supports flexible operations such as range, wildcard and fuzzy queries. Figure 2 shows a trie structure (simplified) for the document from Figure 1. Each node contains characters of the indexed `token`, and the `pre,pos` value pairs ($\texttt{pre}_0$,$\texttt{pos}_0$|...|$\texttt{pre}_n$,$\texttt{pos}_n$) identify all occurrences of the token. The `pre` value references the text nodes stored in the database table; the position within the text node is remembered as `pos` value. As the index is built in document order, all stored `pre,pos` values are automatically sorted—a property which comes in handy, as we will see in Section 4.

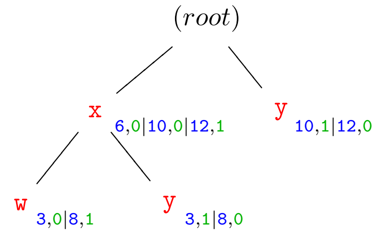


Fig. 2. Compressed Trie: characters with `pre,pos` value pairs

Whereas many tries are designed to work in main memory, the presented index exclusively operates on flattened and compressed array structures. This way, it can be directly stored to disk, and access time and memory consumption is minimized. As some index requests—such as a *count()* on the number of results—will only access meta data, structural and reference data are stored in separate containers. The structural container contains the indexed token characters, references to child nodes, the number of results, and offsets to the reference container which contains all `pre,pos` pairs. More implementation details can be found in [11].

3 Full-text Evaluation Strategies

BaseX employs three different evaluation strategies for full-text queries: sequential scanning, index-based processing with path inversion and a hybrid approach. All of them are presented here, along with a decision framework to select the best mode. The following queries are used to illustrate the query evaluation strategies:

Q1: `/A/C[D/text() ftcontains "x"]`
Q2: `//D[text() ftcontains "x"]`
Q3: `//*[text() ftcontains ftnot "x"]`

3.1 Sequential Scanning

Query Q1 consists of child steps and a predicate with an `ftcontains` expression. The corresponding sequential query plan (simplified) is depicted in Figure 3. The evaluation requires a sequential scan of the document. The *LocationPath* expression starts from the root node and traverses all child nodes. Each `A` element is passed on to the next child step, and the resulting `C` elements are filtered by the *FTContains* expression. The left-hand *LocationPath* yields all `text()` nodes of `D` elements, which are checked for the token `"x"`.

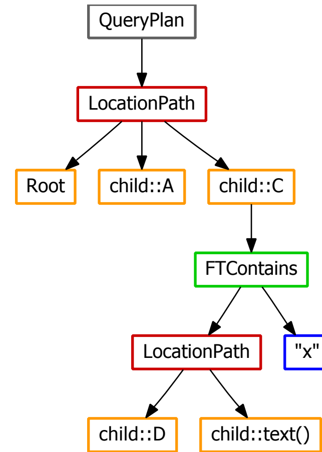


Fig. 3. Query Plan: sequential processing of Query Q1

Obviously, with increasing document size, the sequential scan becomes a bottleneck as all nodes addressed by the query have to be touched at least once.

3.2 Index-based Processing with Path Inversion

In XML databases, a large variety of index types exists. Content (or value) indexes facilitate direct access to text nodes in a document, and different variants are found in practice:

- Some databases reference results on the **document level**. This is often done if XML is stored in relational database columns. Queries on many small documents can be accelerated by this approach, while there is no benefit for single and large documents.
- Certain **location paths** can be pre-selected for being indexed. While this seems promising at first glance, it often fails when queries are nested or getting more complex. Moreover, users need explicit knowledge about the existing index structure.
- Implementation-defined **XQuery functions** allow for a direct index access. Knowledge on the database internals is needed, and, next to that, a query compiler will not benefit from the indexes, as the user alone decides whether the index is to be used.

To support arbitrary full-text expressions, we chose to index all text nodes by default, regardless of their position in the document structure. As demonstrated in the following, the query optimizer will rewrite and invert location paths and predicates whenever an index access is possible.

In Figure 4, the index-based execution plan of Query Q1 is depicted. In contrast to the sequential scanning mode, which evaluates queries from the document root down to leaf nodes, a bottom-up approach is pursued by first accessing

the full-text index and secondly traversing the path back from the leaf nodes to the document root.

First of all, the *FTIndex* operator returns the references of all text nodes containing the token `"x"`. Next, parent elements `D` and `C` are selected. Finally, the ancestor path of the remaining nodes (including the document node) is checked to dismiss results which do not comply with the original query path. Path inversion is possible due to the symmetries of certain XPath axes. Forward-looking, top-down variants have been discussed in detail in [20], and some of them are shown in Table 1. By extending them to multiple location steps, they serve well to dynamically rewrite a large number of location paths.

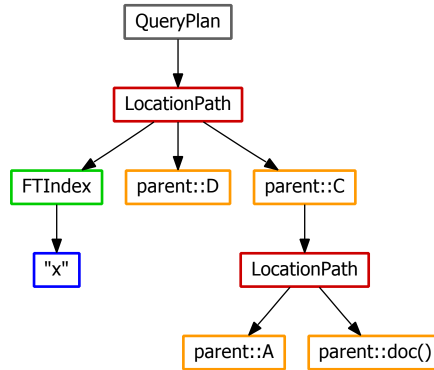


Fig. 4. Query Plan: index-based processing of Query Q1

Path	Equivalent Path
`/descendant-or-self::m/child::n`	`/descendant::n[parent::m]`
`/descendant-or-self::m/descendant::n`	`/descendant::n[ancestor::m]`
`p[ancestor::m]/self::n`	`p/self::n[ancestor::m]`
`p/following::m/descendant::n`	`p/following::n[ancestor::m]`

Table 1. Location paths and their equivalents

The second Query Q2 (`//D[text() ftcontains "x"]`) introduces a descendant-or-self and child step, which can be merged, in this case, to a single `descendant::D` step. Queries with descendant steps will be executed more slowly by some query engines, as virtually all document nodes have to be touched and checked for its node kind and tag name. The optimized, index-based execution plan in Figure 5, however, is very compact: as the descendant step in the original query selects all `D` elements in the document, regardless of their path to the root node, the ancestor and document test can be completely skipped. As the additional ancestor traversal, which has to be evaluated for each single node, takes additional time, this query will be executed even faster than Q1.

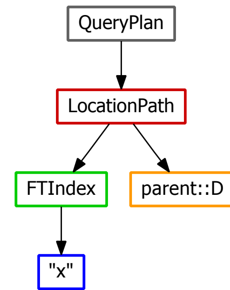


Fig. 5. Query Plan: index-based processing of Query Q2

Value indexes can be used to find out, where a text is found in a document, but not to find places of its absence. If a full-text query contains an `ftnot`

expression, the option to use an index access with consecutive path inversion turns out to be useless.

However, the index can still be of value in a sequential traversal, as the tokenization and normalization of all touched text nodes can take much longer than a simple reference test in a modified `FTNot` operator implementation.

3.3 Hybrid Processing: Sequential Evaluation with Index Usage

Figure 6 shows the resulting query plan for Query Q3 (`//*[text() ftcontains ftnot "x"]`). It resembles the sequential execution plan—except for the full-text expressions, which are all index-aware. If the *FTIndex* operator is called for the first time, the index is accessed once. *FTIndexNot* checks for each node if it is not part of the index result, and *FTIndexContains* works similar to the conventional *FTContains* operator, but basically avoids tokenizing the current node. If the incoming nodes are guaranteed to be sorted, *FTIndexNot* will operate even faster. As all index references are sorted as well (see Section 2.2), it can completely run in an iterative manner.

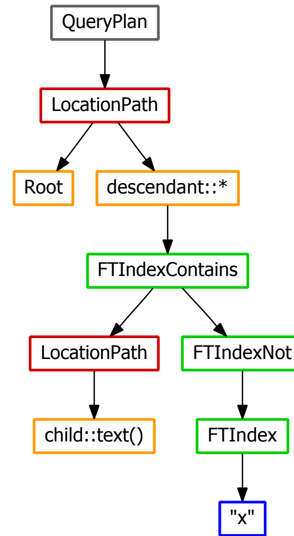


Fig. 6. Query Plan: hybrid processing of Query Q3

3.4 Choosing the Proper Processing Strategy

A two-step model is used by the query compiler for choosing the proper processing strategy. In the first step, it is decided whether it is possible and efficient to use the index, while the second step rewrites the affected operators in a positive first case. The tag/attribute index and path summary are used to perform some basic cost estimations, which influence the decision for or against index access. The number of expected text nodes, their average text length (which influences the time for tokenizing text nodes) and their position in the path summary are considered as well as the number of index results, which can be requested from the full-text index. If a query potentially allows performing several index requests, it can be cheaper to only access the index once and process the other predicates sequentially. Query execution can be completely skipped if the index indicates that a term will yield no results at all.

For the sake of simplicity and to present but the core functionality, we have limited the discussion to the optimization of basic location paths. A slightly more complex query is shown in Figure 7. It contains a *FLWOR* expression, a general comparison and an `ftcontains` expression with an additional `ftand` connective. The query plan illustrates that the available indexes can be applied here as well.

4 Iterative Evaluation of XQuery Full Text

4.1 Sequential Evaluation

Iterative/pipelined query evaluation is a general database concept [13] which is applied in a number of other XQuery implementations [8, 9, 18]. In contrast to a conventional, set-based approach, items are processed one-by-one, which guarantees constant memory consumption. The pipeline is only broken by so-called blocking operators that need their complete input, which is the case for sorting, for instance. Iterative evaluation can add some minimal overhead, but it yields particularly good performance when the creation of large intermediate results can be avoided, that are later reduced to a small, final result set.

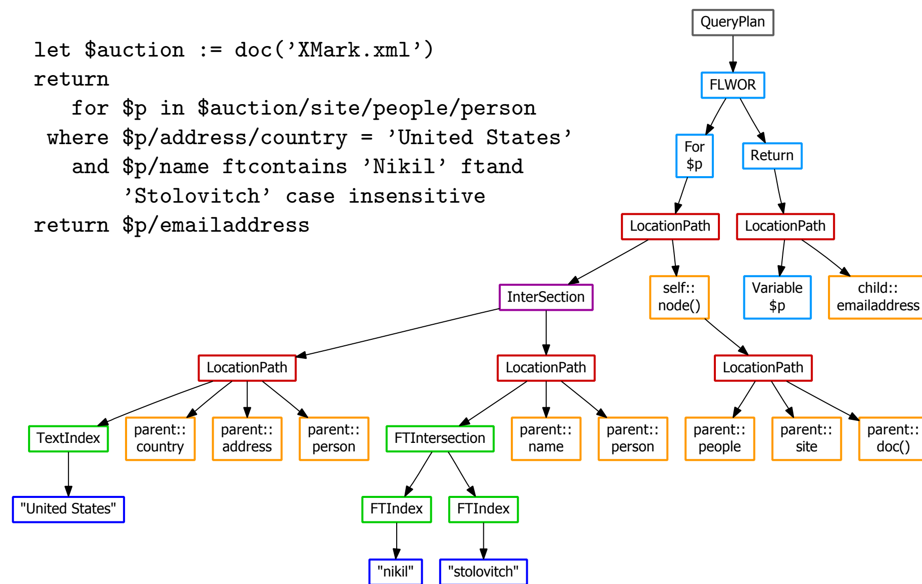


Fig. 7. XQuery with *FLWOR* expression

Although the internal XQuery Full Text data model is complex, as scoring values are calculated and word positions are passed on to evaluate so-called positional filters (such as word order or distances, see [1] for details), all expressions can be evaluated in an iterative manner. The *FTAnd*, *FTOr* and *FTUnaryNot* expressions are implemented similarly to their XQuery counterparts; both processing modes handle one node per iterator step. Consider, e.g., the *FTOr* iterator that merges nodes with equal `pre` values and returns the node with the smallest `pre` value and its corresponding `pos` values. The full-text references of the remaining operands have to be temporarily cached as iterators return data only once. Additionally, the *FTMildNot* operator, which has no XQuery equivalent, has to check whether one occurrence of the first operand is not followed by any other occurrence of the remaining operands.

4.2 Index-Based Full-Text Iterator

As described in Section 2.2, the full-text index references `pre` and `pos` values for each index term. Querying the index means that all references are fetched from disk and returned via an iterator. But in many cases, the entire full-text data is not needed to successfully evaluate a query. Therefore, the iterator concept was pushed down to the index structures. The iterative implementation of the `FTIndex` operator works as follows: After initializing the iterator with the structural data, all data for the first node reference ($\texttt{pre}_0$) is read, i.e., all `pre,pos` value pairs from $\texttt{pre}_0$,$\texttt{pos}_0$ to $\texttt{pre}_0$, $\texttt{pos}_n$ are processed and returned. In the next iteration, the data stored for the reference $\texttt{pre}_1$ is read and returned, and so on. This process continues as long as more index results are requested, or all references have been returned.

4.3 Iterator Trees: Processing Non-Trivial Index Requests

Iterative index processing is simple and straightforward, as long as single index terms are requested. If the index, returns results for wildcard queries, for instance, the references of several index terms have to be merged and returned. As all index references (i.e., their `pre` value) are sorted by document order, the iterative approach can easily be extended to an arbitrary number of index iterators and a union expression on top of them. Each single index access is managed by an index iterator. It keeps the offset and number of `pre,pos` value pairs stored for an index token.

The following wildcard example is based on the introductory XML document and full-text index shown in Figures 1 and 2. For each index hit, which is recursively matched by the trie algorithm, an index iterator is created. The resulting index tree is evaluated every time an index result is requested. The `pre,pos` value pairs with the smallest `pre` value are merged and returned.

The following example illustrates the presented approach. The full-text query `//*[text() ftcontains "x.*" with wildcards]` yields all elements with a text node that contains a token starting with the character `"x"`. In our example, three tokens (`x`, `xw`, `xy`) match the wildcard expression. The wildcard algorithm creates an index iterator tree, which is depicted in Figure 8.

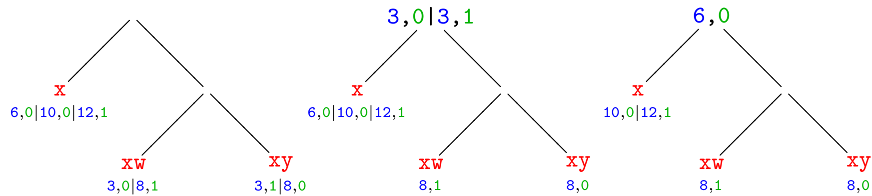


Fig. 8. Index iterator for `//*[text() ftcontains "x.*" with wildcards]`

Each iterator, which represents one single token, returns results in the known format $\texttt{pre}_0$,$\texttt{pos}_0$|...|$\texttt{pre}_n$,$\texttt{pos}_n$. At the first step, the smallest `pre` values have to be obtained. Therefore, each node of the iterator tree returns its smallest

pre value and the corresponding pos values. Next, the pos references of equal pre values are merged. As shown in the figure, the root node now contains the minimum pre value 3 and the merged pos values 0 and 1. The next step will move pre value 6 to the top. After that, the second and third iterator will return their values for pre value 8, and the index tree will be reduced to a single iterator, which will return pre values 10 and 12.

5 Experimental Analysis

The following tests demonstrate the performance gains by applying indexes to full-text querying. All tests were performed with BaseX 5.6[1]. We used a 2.3 GHz Intel Xeon CPU with 32 GB RAM as hardware and Suse Linux 10.2 and Java 1.5.0.16 as software. Four XMark instances (sized 11 MB, 111 MB, 1 GB and 11 GB) were generated and used as query input.

#	Query
Q1	`doc('xmark')//keyword[text() ftcontains 'barrel']`

Q2	`for $mail in doc('xmark')/site/regions/*/item/mailbox/mail`
	`where $mail//text/text() ftcontains 'seeking.*' with wildcards`
	`return $mail/from`

Q3	`for $item in doc('xmark')/site/regions/*/item`
	`where $item//listitem/text/text() ftcontains ftnot 'preventions'`
	`return <result>{ $item/location/text() }</result>`

Table 2. Tested queries

The three queries in Table 2 are supposed to summarize the discussed query rewritings. Query Q1 contains a simple descendant step and an ftcontains expression. Query Q2 uses a number of child steps to address the relevant text nodes, and the full-text expression is extended by a wildcard option. An ftnot operator is used in the third query Q3.

All performance results are listed in Table 3 and illustrated in Figure 9. The times represent the average over several runs (5-100 runs, depending on the document size); they include the time for parsing, compiling and evaluating the query as well as printing the result. The boxes show the result sizes in kilobytes.

Query	11 MB	111 MB	1 GB	11 GB
Q1: Size	0,2	0,7	6	60
Q2: Size	1,5	14	118	1190
Q3: Size	16	165	1656	16602
Q1: Sequential	0.116	1.109	11.03	109.8
Q1: Index	0.001	0.003	0.017	0.128
Q2: Sequential	0.302	2.964	29.39	292.3
Q2: Index	0.006	0.041	0.396	3.831
Q3: Sequential	0.138	1.383	13.43	132.3
Q3: Hybrid	0.074	0.721	7.355	75.15

Table 3. Result size in KB, execution times in seconds

As expected, all index-based queries yield better results than their sequential equivalents. The index-based version of Q1 is evaluated fastest, as the resulting query plan (which is similar to Figure 5)

[1] Open-source, available at http://www.basex.org.

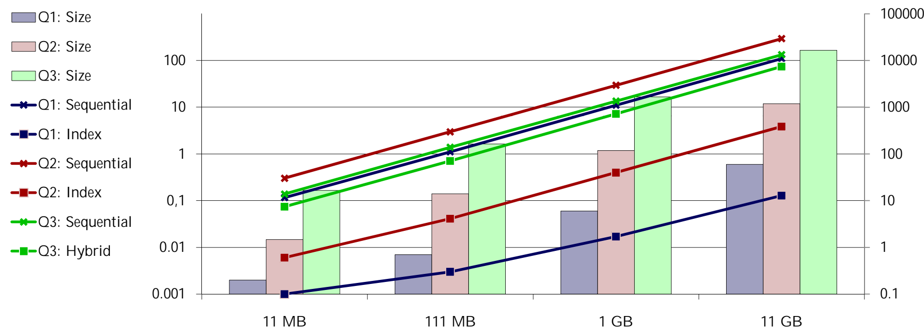


Fig. 9. Performance results. Boxes/right axis: result size in KB, lines/left axis: execution time in seconds

only contains the index access and a parent step. The scalability is sub-linear, as the index version is about 1000 times faster than the sequential version with the 11 GB input, compared to a factor of 100 for the 11 MB input. Q2 adds some overhead with the wildcard operator, and the larger result size amounts to a virtually linear execution time for both the sequential and the index-based approach. Query Q3 demonstrates the potential of the hybrid query evaluation. As tokenization of text nodes can be avoided, index-supported querying is about twice as fast as pure sequential processing. In spite of the large result size, the hybrid approach is still faster than the pure sequential solution for Query Q1. Documents with larger text nodes (such as, e.g., the Wikipedia XML instances[2]) will yield even better results if text tokenization can be avoided.

As the performance results indicate, there is a clear relationship between the execution times and the data size. As larger XML instances yield larger result sets, it is worth adding that the sequential and hybrid execution is mainly dependent on the size of the input document, whereas the index-based variant exclusively depends on the size of the query result.

6 Visualization of XML and Full-Text Results

Since the first release, BaseX offers a graphical frontend to visually explore content and structure of stored XML data [15, 17]. Figure 10 (background) shows a Wikipedia fragment using the Treemap visualization [21]. Each element is drawn as a rectangle and the element tag is printed in the upper left area of this rectangle. The inherent structure of the instance is clearly recognizable: A starting `siteinfo` element containing some meta data, which is followed by several `page` elements each corresponding to a Wikipedia article. The structure of the `page` elements is good to grasp as well: `page` elements contain a `title`, `id` and `revision` element, which again contains elements, for instance the `text` element storing the full-text article. The space-filling treemap often allows the viewer to comprehend the complete structure of a document at a glance. By

[2] Available at `http://download.wikimedia.org`

interacting with the treemap, e.g., zooming into a subarea, a higher level of detail can be achieved. As such, an explorative browsing approach may be used to obtain further details about the data instance. Rectangles corresponding to result nodes of a query are highlighted using a contrasting color code.

Fig. 10. Treemap visualizations of XML data.

It is in the nature of full-text queries to often produce large result sets with long textual contents. Our standard text visualizations have shown to be insufficient in terms of compact result presentation and general overview over content and structure. We chose to enhance the treemap visualization by a dynamic abstraction layer using token/sentence thumbnails in combination with full-text tooltips to overcome these deficiencies.

As previously discussed, full-text operators report the `pre` value and the token position `pos` for each search term in a full-text query. Leveraging such information, a visualization can provide a more compact and space-preserving treemap layout by using thumbnail representations for tokens or, at a higher level, sentences. The approach is straightforward. Whenever there is enough space to place the original text into a rectangle, it is displayed as usual. If this is not the case, tokens are replaced by thumbnails, following an approach by Kau-

gars [19]. The length of a thumbnail correlates with the size of the represented text token. Line breaks between tokens are preserved.

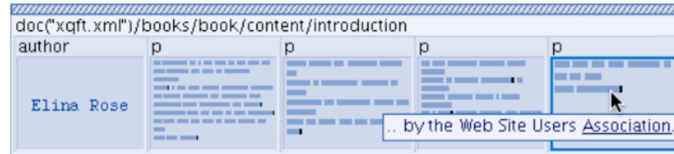


Fig. 11. Full-text thumbnail and tooltip representation.

Figure 11 illustrates the thumbnail representation. As the textual node of the `author` element fits into the corresponding rectangle, it is displayed in its readable format. The thumbnail representation is used for the text nodes of the `p` elements. The black thumbnail entities denote periods or other sentence terminators. As mentioned, the length of a thumbnail is relative to the length of the represented token, as such the structure of the sentence is preserved. Once the mouse cursor is moved over a thumbnail, the original text is displayed in a tooltip.

Figure 10 (foreground) displays an area of 35 elements in a Wikipedia instance. All occurrences of the term *"the"* are highlighted. The figure demonstrates another abstraction layer (representing a whole sentence by a thumbnail) of the visualization procedure. In one of the treemap rectangles, there is enough space to fit in the textual content (*"redirect alexander the great r from camelcase"*). However, it is yet too narrow to display the tokens *"alexander"* and *"camelcase"* completely, so they are truncated to *"alexan.."* and *"camelc.."*. In the `comment` elements, the token thumbnail representation is chosen. Once more, black rectangles indicate sentence delimiter. For the `text` elements, the sentence-based thumbnail abstraction is chosen. Hereby we can observe two characteristics: For text passages of median length the original sentences are still good to be recognized. The longer text passages are, the darker they appear due to the increasing number of delimiters. The structure of the text, however, is preserved in all abstraction levels.

Using dynamic thumbnail representation for full-text bodies allows space-saving visual representations of large text bodies in a small display area. Combined with tooltips, which additionally display preceding and following text blocks of the selected token, it is possible to sequentially read and browse through the compacted, thumbnailed text, as illustrated in Figure 12.

7 Summary

We presented aspects of the architecture of the XQuery Full Text Recommendation in BaseX, an open-source DBMS developed at U Konstanz. As one of, if not *the*, first complete implementation of all language features, our system provides

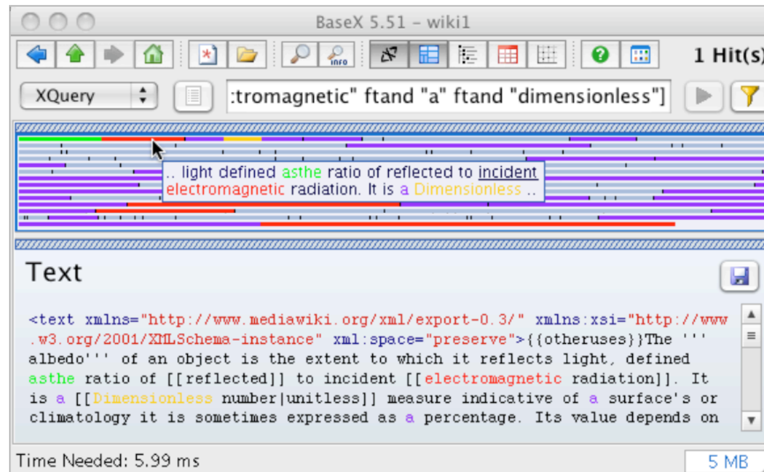


Fig. 12. Split visual result presentation of a full-text query. Above: a sentence based thumbnail representation with highlighted full-text tokens. Below: the textual representation in the original document.

simple sequential query processing algorithms that allow for pipelined processing of operator sequences as well as (full-text) indexes to speed-up search. In addition, a hybrid query execution strategy is employed whenever pure index-based or sequential processing seems to promise only second-best performance. Substantial query rewrite optimizations have already been incorporated, even though BaseX does not yet involve a full-blown cost-based query optimizer trying to always find the best possible plan.

Our initial performance evaluation proves perfect scalability of both, sequential and index-based execution plans. Actually, we were even able to take advantage of indexes for the evaluation of queries with negated full-text predicates (`Not` expressions). Finally, BaseX's visual querying interface and result display has also been extended for full-text applications, such that matches w.r.t. full-text predicates can be highlighted in query results. Several XML visualizations are available in BaseX, e.g., the treemap that clearly show the document structure together with varying content detail, depending on document or result set size. Using highlights and tooltips or split views, the system gives visual feedback to the user as to where matching part of the XML document have been found.

Future work will include more subtle query optimization and index evaluation strategies as well as additional functionality to cover language-specific full-text features. Also, we plan to extend our visual querying interface and result display with a variety of zoomable representations.

References

1. Sihem Amer-Yahia et al. XQuery and XPath Full Text 1.0. W3C Candidate Recommendation. http://www.w3.org/TR/xpath-full-text-10, May 2008.
2. Jun-Ichi Aoe et al. An Efficient Implementation of Trie Structures. *Software – Practice and Experience*, 22(9):695–721, 1992.
3. Attila Barta et al. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *Proc. of the 31st VLDB Conference*, pages 133–144, Trondheim, Norway, 2005.
4. Anand Bhaskar et al. Quark: an efficient XQuery full-text implementation. In *Proc. of the ACM SIGMOD Conference, Demo Tracks*, pages 781–783, Chicago, Illinois, USA, 2006.
5. Scott Boag et al. XQuery 1.0: An XML Query Language. W3C Recommendation. http://www.w3.org/TR/xquery, January 2007.
6. Peter A. Boncz et al. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. of the ACM SIGMOD Conference*, pages 479–490, Chicago, Illinois, USA, 2006.
7. Emiran Curtmola et al. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. In *Proc. of the 2nd XIME Workshop*, Baltimore, Maryland, USA, 2005.
8. Peter Fischer et al. MXQuery – a low-footprint, extensible XQuery Engine. http://www.mxquery.org, 2009.
9. Daniela Florescu et al. The BEA/XQRL Streaming XQuery Processor. In *Proc. of the 29th VLDB Conference*, pages 997–1008, Berlin, Germany, 2003.
10. Edward Fredkin. Trie Memory. *J-CACM*, 3(9):490–499, September 1960.
11. Sebastian Gath. Processing and Visualizing XML Full-Text Data. Master's thesis, University of Konstanz, Germany, 2009.
12. Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, 1997.
13. Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
14. Christian Grün et al. Pushing XPath Accelerator to its Limits. In *Proc. of the 1st ExpDB Workshop*, Chicago, Illinois, USA, 2006.
15. Christian Grün et al. Visually Exploring and Querying XML with BaseX. In *Proc. of the 12th BTW Conference, Demo Tracks*, pages 629–632, Aachen, Germany, 2007.
16. Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the ACM SIGMOD Conference*, pages 109–120, Madison, Wisconsin, USA, 2002.
17. Alexander Holupirek et al. BaseX & DeepFS: Joint Storage for Filesystem and Database. In *Proc. of the 12th EDBT Conference*, pages 1108–1111, 2009.
18. Wolfgang Hoschek. Nux – an Open-Source Java toolkit for XML Processing. http://acs.lbl.gov/nux, 2006.
19. Karlis Jekabs Kaugars. *A Hierarchical Approach to Detail + Context Views.* PhD thesis, New Mexico State University, Las Cruces, NM, USA, 1998.
20. Dan Olteanu et al. XPath: Looking Forward. In *Proc. of the XMLDM Workshop*, pages 109–127. Springer Verlag, 2002.
21. Ben Shneiderman. Tree Visualization with Tree-Maps: 2-d Space-Filling Approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.