Fast and Incremental Indexing in Effective and Efficient XML Element Retrieval Systems

Atsushi Keyaki, Jun Miyazaki
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma
Nara 630-0192, Japan
{atsushi-ke, miyazaki}@is.naist.jp

Kenji Hatano
Faculty of Culture and Information Science
Doshisha University
1-3 Tatara-Miyakodani
Kyotanabe
Kyoto 610-0394, Japan
khatano@mail.doshisha.ac.jp

Goshiro Yamamoto, Takafumi Taketomi, Hirokazu Kato
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma
Nara 630-0192, Japan
{goshiro, takafumi-t, kato}@is.naist.jp

ABSTRACT

A method for fast and incremental indexing, with both effective and efficient query processing, is proposed for XML element retrieval. When frequent document updates occur on the Web, they must be handled to maintain the effectiveness of the search system. When new topics are added and document statistics change drastically, search accuracy is also reduced. We therefore consider a method not only for updating indices efficiently but also for processing queries effectively and efficiently. We construct indices for fast updating and propose a method for computing accurate term weights even under dynamically changing statistics. Experimental results show that our proposed system can handle document updates at low cost and search documents accurately even when their statistics change.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]; H.3.3 [**Information Storage and Retrieval**]

General Terms

EXPERIMENTATION

Keywords

XML information retrieval, incremental updates of indices, accurate global weights, reduction of update cost

1. INTRODUCTION

An information unit of XML element retrieval is not a document but an element in XML documents. XML element retrieval systems present elements to satisfy users' information needs. Users do not have to spend time seeking out relevant descriptions in each document. Although existing studies of XML element retrieval have already attained both effectiveness and efficiency in query processing [16], [18], [8], [2], these studies do not consider document updates.

Web documents are frequently updated, i.e., inserted, deleted, and modified. Notably, Wikipedia articles are updated 4,000 to 8,000 times an hour[1]. Information retrieval systems are expected to present search results based on the latest content on the Web, especially as new topics are added to documents. Without handling updates, a search system cannot find newly inserted documents and rank documents based on old information, which reduces effectiveness. Thus, we add a function for updating documents to the existing XML element retrieval techniques.

The mainstream approach for updating an index is to construct a new index periodically *from scratch* while discarding the existing one. It may take a long time to retrieve updated documents if constructing a new index is costly. Incremental updates are required to shorten this delay.

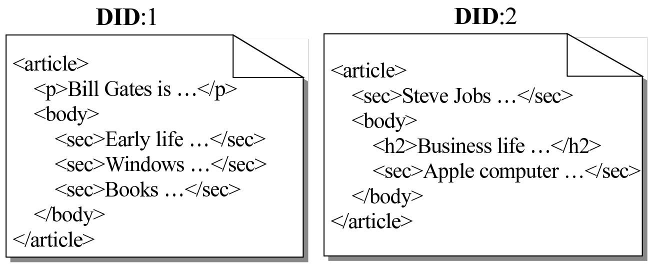
We believe this is the first study focused on fast and incremental updates of indices in effective and efficient XML element retrieval systems. Although Google supports fast and incremental updates with both effective and efficient query processing, Google's approach differs from ours. Google analyzes link information in Web pages to find important pages, while our study utilizes text information. We can apply our approach to other structured documents apart from the Web even if they do not have link information.

We therefore present a function for incremental updates with XML element retrieval. Term weights must be calculated during incremental updates. A term weight is derived from statistics, including aggregate statistics that are difficult to calculate immediately. These statistics are not contained in the indices of the past studies because these studies do not target updating indices. We put these statistics into the proposed indices for handling the updates. Two problems arise in incremental updates:

- search accuracy is affected by inaccurate global weights, and
- it takes a long time to index all terms in all elements.

Concerning the first problem, these statistics are global weights, which are derived from all data of the same class in the document set. A global weight cannot be calculated accurately with insufficient data in the class. We need an effective way to define the

[1]`http://www.wikichecker.com/editrate/`


DID:1

```
<article>
   <p>Bill Gates is ...</p>
   <body>
      <sec>Early life ...</sec>
      <sec>Windows ...</sec>
      <sec>Books ...</sec>
   </body>
</article>
```

DID:2

```
<article>
   <sec>Steve Jobs ...</sec>
   <body>
      <h2>Business life ...</h2>
      <sec>Apple computer ...</sec>
   </body>
</article>
```

Figure 1: XML document

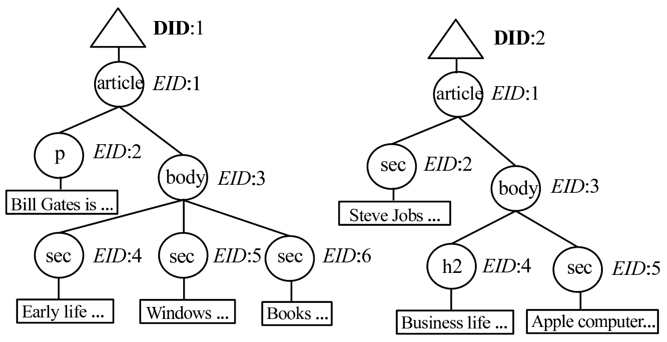


Figure 2: XML tree

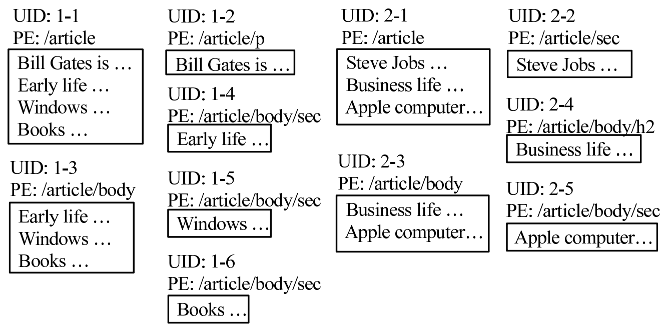


Figure 3: XML element

class to handle this. Such a method would be especially useful as new topics are added and document statistics change. The statistics relating to the new topics vary as the documents are updated, and they must be recalculated rapidly.

The second problem means that incremental updates prolong indexing time. Moreover, in XML element retrieval, a much larger number of search targets must be handled compared to the number of XML documents[2]. Therefore, the cost to index all data is high. To enable fast updates, we target only "important" parts of a document with an *element filter*, and target significant terms for query processing with a *term filter*. For search accuracy, we must identify these parts and terms carefully.

Using these techniques, we propose a method for fast incremental updates on indices with effective and efficient querying in XML element retrieval. We evaluated the effectiveness and efficiency of our approaches through experiments with two scenarios: the static statistics case where topics rarely change, and the dynamic case in which new topics are added frequently.

2. XML ELEMENT RETRIEVAL

In this section, we describe the abstracts of XML elements and queries in XML element retrieval.

2.1 XML Element

We show concrete examples in Figures 1, 2, and 3 to explain the definition of XML elements. Figure 1 illustrates an example of XML documents. Each document is assigned a Document ID (DID). Figure 2 depicts trees that are translated from Figure 1. An XML document can be expressed as a tree structure, which helps to understand the structure of the document. Each element is assigned an Element ID (EID). EIDs are assigned in document order.

[2]For a given document set, the index size of a document retrieval system is 13GB, while that of an XML element retrieval system is 89GB. XML elements are discussed in Section 2.

We can identify an element by its DID and EID. We call this identifier a Unique ID (UID). A pair of start and end tags represents an XML element node in an XML tree, and the nested structure of XML elements represents parent-child relationships. Each element in Figure 3 is the text that is composed of a set of text nodes in the XML tree in Figure 2. We describe the path expression (PE) of each element.

The author of a document makes structures, e.g., chapters, sections, paragraphs. We utilize these structures to identify the best description for satisfying users' information needs. Some structures are meaningless and elements defined by these structures are inappropriate as search results, and some existing studies try to eliminate them. For example, suppose a user seeks information about "Early life ...", "Windows ..." and "Books ···" of DID 1. XML element retrieval systems will try to present an element whose UID is 1-3 to the user because the element contains all of the information that the user needs and no further information.

2.2 Queries of XML Documents

There are two ways for expressing a user's information need in a query: keyword and document structure. A content-only (CO) query is composed of query keywords. Users can use a CO query even if they do not know the structures of documents they query. In contrast, a content-and-structure (CAS) query is composed of one or more pairs of a query keyword and a constraint on the document structure. With a CAS query, users can get specific results with regard to granularity and content. CO queries are used just as they are used in traditional text document information retrieval, whereas CAS queries utilize one of the most significant features, i.e., document structures.

We show concrete examples of a CO query and a CAS query, respectively. They are expressed as XPath I (NEXI) [19]. A CO query: `//*[about(., "Windows")]` means that the candidates of search results are elements containing "Windows". The elements of UIDs 1-1, 1-3, and 1-5 can be search results in Figure 3.

A CAS query: `//article[about(., "Steve")]//sec [about(., "Apple")]` is a more complex one. Let us focus on the first half of the query: `//article[about(., "Steve")]` means that candidates for this part are elements containing "Steve" whose path expressions end with an `article` tag. The second half of the query, `//sec[about(., "Apple")]`, means that candidates for search results are elements containing "Apple" whose path expressions end with a `sec` tag. Search results are elements satisfying the latter constraint whose ancestor elements satisfy the former constraint. The only element satisfying the query constraints is UID 2-5 in Figure 3.

3. RELATED STUDIES

Here we explain effective and efficient XML retrieval. We also discuss studies of updates in search systems.

3.1 Effective and Efficient XML Search

3.1.1 Effective XML Search

The most important goal of XML element retrieval is highly accurate search. The mainstream approach of extracting relevant elements is as follows. First, calculate a term weight in each element using a term-weighting scheme. Next, compute an element's score using these term weights.

Term-weighting schemes for XML element retrieval are often derived from studies on document searches. The most significant difference between document retrieval and XML element retrieval is the method of computing global weights, which are derived from all data of the same class in a document set. Term-weighting schemes in document retrieval assume that every document has the same attribute and belongs to the same class. Thus, global weights are calculated using all documents. On the other hand, in XML element retrieval, elements are assigned to classes. Global weights are calculated for elements of the same class.

There are different ways to classify elements. One approach is to place into the same class all elements having the same path expression. In Figure 3, since every element of UID 1-4, 1-5, 1-6, and 2-4 has the same path expression `/article/body/sec`, the global weights are calculated using these elements.

Alternatively, elements with the same tag are placed in the same class. Because all elements of UID 1-4, 1-5, 1-6, 2-2, and 2-4 have the same `sec` tag, the global weights are calculated using these elements, as depicted in Figure 3. We use path expression-based classification in our system because it has been reported [14] to be more accurate.

TF-IPF [6] is a classic term-weighting scheme. TF is a local weight and calculated by the term frequency of a term in each element. IPF is a global weight and it is calculated by an inverse path frequency of each term in all elements with the same path expression. The weight of TF-IPF is derived from the product of TF and IPF, while an element's score is the sum of term weights of query keywords.

Another popular approach is the BM25E [7] probabilistic model, which considers the statistics of not only term frequency but also element length (the number of terms in an element). The score of an element is also the sum of term weights of query keywords.

Language model techniques have been developed in the fields of speech recognition and machine translation. Recently, these techniques have been introduced into the field of information retrieval. In particular, the query likelihood model [9] is well studied and achieves significant results. This model has been adapted to XML element retrieval [13]. In the term-weighting scheme, the score of each element is the product of the probability of each query keyword. This means that only the elements containing all query keywords are computed as a nonzero value. To avoid this, smoothing techniques are often used. Smoothing values are computed not by a document model but by a collection model, which is computed for an entire document set.

Each of the term-weighting schemes is composed of three types of factors: local weights, global weights, and constant values (coefficient values and parameters). When a new term weight is calculated, local weights and constant values are easy to calculate and refer to because local weights are computed for a newly inserted element. However, it is difficult to calculate global weights on demand because the entire document set must be scanned to compute them. Therefore, to calculate a new term weight immediately, the statistics of global weights must be stored in an index for quick access.

3.1.2 Efficient XML Search

Fast query processing is also required by system users although the most important requirement of XML element retrieval is enabling effective search.

To attain efficient XML element retrieval, various approaches have been taken, such as 1) compressing and reducing data to suppress index size to minimize the amount of data scanned in querying, and 2) applying Top-*k* algorithms to return search results quickly. In particular, some studies [16], [18] have used term-weighting schemes [7].

Many Top-*k* searches have been proposed. There are some conditions for efficient querying: 1) term weights have already been calculated before querying, and 2) terms are arranged in descending order of their weights. This means that we have only to scan highly ranked terms in querying.

Theobald et al. proposed two types of indices and a Top-*k* algorithm for efficient search [16]. One is for scoring an element in querying, and the other is for checking a structural constraint of a query. They also proposed cost-based query processing, which identifies an effective moment to check the structural constraints and identifies which query keyword is reasonable to be processed.

Trotman et al. proposed a low-cost method of data compression and selection [18].

In these studies, the aim is to retrieve elements satisfying a user's information need from a fixed document set, i.e., they do not consider document updates.

3.2 Handling Document Updates on the Web

The handling of document updates is especially important in Web search systems because new documents are constantly inserted, deleted, and modified. When documents are updated, useful search systems should treat them as search targets. Moreover, if search systems present search results based on a past snapshot of the Web, the original content of the Web documents may have been changed. Search systems should reflect the current state of the Web and handle dynamically changing Web documents.

Recently, some techniques for handling document updates have been proposed. Chen et al. [1] tackle this challenge in the field of information extraction. They report the long processing time required to apply information extraction techniques to document collections when document updates occur. As a result, a delay occurs before information extraction is available on the updated documents. To shorten the delay, they propose a method to recycle intermediate results of past snapshots. Neumann et al. [12] also utilize the information of past snapshots effectively with RDF data.

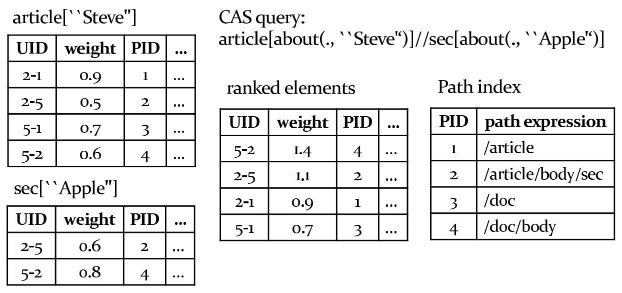
Ren et al. [15] preserve not only the latest graph data but also past snapshots to trace the transition of the graph. This study is different from ours to the extent that we present the information along with the latest state of the Web.

These studies utilize the intermediate results of past snapshots. Hence, we also utilize them or existing indices. We incrementally update existing indices when new documents are inserted. In addition, Web search systems are expected to maintain high performance with a low update cost. In the case of text retrieval, high search accuracy should also be maintained.

There has been no adequate study focused on incremental updates in XML element retrieval with effective and efficient querying. Therefore, this is the first study tackling this problem. Although some studies have focused on incremental updates of an inverted index [17], [5], [10], they propose data structures of indices and physical storage methods. These studies differ from ours

– `Term` (Unique ID, term, term weight, Path ID, element length)
– `Tag-term` (Unique ID, tag, term, term weight, Path ID, element length)
– `Path` (Path ID, path expression)
– `GW-Path-term` (Path ID, term, frequency [, smoothing values of the query likelihood model])
– `GW-Path` (Path ID, frequency, total length)
– `Term-filter` (tag, term, threshold value)

Figure 4: Structure of the indices



article[``Steve"]

UID	weight	PID	...
2-1	0.9	1	...
2-5	0.5	2	...
5-1	0.7	3	...
5-2	0.6	4	...

sec[``Apple"]

UID	weight	PID	...
2-5	0.6	2	...
5-2	0.8	4	...

CAS query:
article[about(., ``Steve")]//sec[about(., ``Apple")]

ranked elements

UID	weight	PID	...
5-2	1.4	4	...
2-5	1.1	2	...
2-1	0.9	1	...
5-1	0.7	3	...

Path index

PID	path expression
1	/article
2	/article/body/sec
3	/doc
4	/doc/body

Figure 5: Query processing

because we introduce a function for incremental updates on the indices for several purposes in XML element retrieval by proposing a method of efficient data management.

4. A SIMPLE EXTENSION FOR INCREMENTAL UPDATES

We present a function for incremental updates in XML element retrieval systems based on the former discussions. A general constitutional XML element retrieval system has functions for managing indices and query processing. Such a system can be easily extended by calculating term weights immediately. We call a system with incremental updates a simple extension system. We discuss query processing followed by index management.

4.1 Query Processing with the Indices

We define a structure of the proposed indices. In many existing studies, term weights are first calculated before being stored in the indices, and structural constraints can be checked with them. The proposed indices inherit these capabilities, but also contain global weights to calculate a term weight immediately, which is essential for incremental updates.

As in the related studies, we define the structures of the indices in an RDB format as shown in Figure 4. Primary keys are underlined. `Term`, `Tag-term`, and `Path` indices are used for efficient and effective queries as in other studies. The other indices, `GW-Path-term`, `GW-Path`, and `Term-filter` indices, are proposed for fast and incremental updates.

To process a CO query, the `Term` index is used. The data to be scanned are specified by a term. For efficient queries, the data are grouped by the term. The score of an element is computed with the weights of each term.

A CAS query needs the `Tag-term` and `Path` indices to be processed. The data in the `Tag-term` index to be scanned are specified by a tag-term pair. For efficient queries, the data are grouped by tag-term pair. The score of an element is also computed with each term weight. When the CAS query contains two or more structural constraints, the path expressions of elements must be checked to see whether they satisfy the query constraints.

We show a concrete example of query processing in Figure 5. Suppose that a user submitted a CAS query: `//article[about(., "Steve")]//sec[about(., "Apple")]`, which is explained in Section 2. First, we calculate each element's score. We use the `Tag-term` index because the query to be processed is a CAS query. We scan the data of two tag-term pairs: one is `article` and "Steve" while the other is `sec` and "Apple". With two lists of tag-term pairs, each element's score is calculated by summing the term weights. Next, we determine search results. The search results of CO queries and single-structural-constraint CAS queries are presented in descending order of element score. In contrast, multi-structural-constraint CAS queries (like this example) should check the structural constraints. The first element in ranked elements is eliminated because its Path ID (PID) does not satisfy the structural constraint. As the result of checking structural constraints of ranked elements, the only element satisfying the structural constraints is UID 2-5, and the element is returned as a search result. Although query optimization is not the main focus of this study, we need only sort data with term weights to perform a Top-k search.

Before a datum is stored in the `Term` and `Tag-term` indices, a term weight should be calculated. To compute the weight of a term t in an element whose path expression is p, many term-weighting schemes require the statistics of the number of elements with p, the number of elements with p containing t, and the total length of the elements with p. These statistics are contained in the `GW-Path-term` and `GW-Term`. We need only store all kinds of the global weights in indices when using another term-weighting scheme requiring other statistics. In addition, we need to store smoothing values of the query likelihood model in the `GW-Path-term` index when using it as a term-weighting scheme.

Because this simple approach does not need the `Term-filter` index, we discuss it in Section 5.2.2.

4.2 Managing Indices

In this section, we explain how to create and update indices.

As a concrete example, we use the BM25E [7] term-weighting scheme. Let $w(e,t)$ be a term weight of a term t in an element e. Suppose that the path expression of e is p.

$$w(e,t) = \frac{(k_1+1)tf_{e,t}}{k_1((1-b)+b\frac{el_e}{avel_p})+tf_{e,t}} \cdot \log\frac{N_p - pf_{p,t}+0.5}{pf_{p,t}+0.5} \tag{1}$$

where $tf_{e,t}$ is the term frequency of term t in element e; $pf_{p,t}$ is the element frequency of term t in the elements with p; N_p is the number of the elements with p; el_e is the length of element e; and $avel_p$ is the average length of the elements with p. Variables k_1 and b are given parameters, which are set to 2.5 and 0.85, respectively, based on our past experiments.

These statistics are classified into three groups: local weights, global weights, and constant values. The statistics, $tf_{e,t}$ and el_e, are calculated for a single element, and so we classify these statistics as local weights. The statistics, $pf_{p,t}$, N_p, and $avel_p$ cannot be determined from a single element, and so they are classified as global weights. The others, including parameters k_1 and b, are constant values. Thus, we need local weights and global weights to calculate a term weight.

We parse the document to extract elements and construct indices. First, we calculate global weights by parsing the entire document set as depicted in Figure 6(1). We also construct the `Path` index simultaneously. Second, we parse the entire document set again to calculate term weights using the global weights contained in the `GW-Path-term` and `GW-Path` indices and local weights derived from each element as shown in Figure 6(2). Normally, general con-

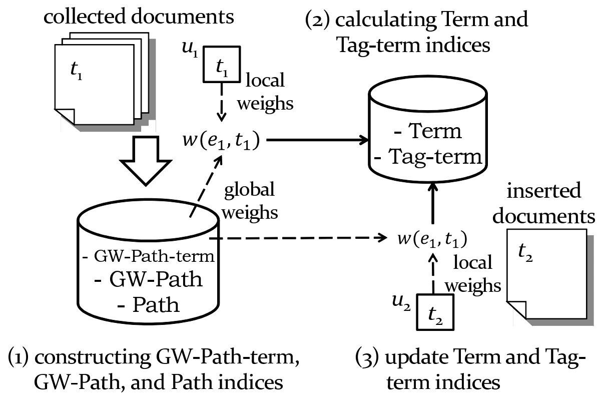


Figure 6: Constructing and updating indices

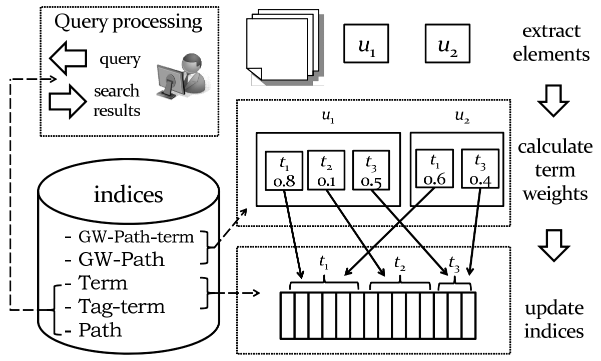


Figure 7: Architecture of the simple approach

stitutional XML element retrieval systems discard global weights when term weights are calculated. However, the simple extension system must retain the global weights for processing document insertions.

The `Tag-term` and `Term` indices are updated incrementally when a document is inserted into the system. We also parse the document to extract elements and calculate a term weight for each element using local weights derived from the element and the global weights contained in the `GW-Path-term` and `GW-Path` indices, as shown in Figure 6(3). Using the `GW-Path-term` and `GW-Path` indices, we can attain fast updates of the indices. The granularity of update can be extended to a set of documents although the basic granularity of update is a single document.

We show the architecture of the simple extension system in Figure 7. It shows that elements u_1 and u_2 are extracted by document insertions. Element u_1 contains t_1, t_2, and t_3, while element u_2 contains t_1 and t_3. After each of their term weights is calculated, the `Term` and `Tag-term` indices are updated. Terms t_1, t_2, and t_3 are inserted into the corresponding positions.

When document modifications occur, we delete the related data and insert new data. A modification-oriented indexing approach remains one of our future works.

4.3 Evaluations and Problems of the Simple Extension System

We examine the effectiveness and efficiency of incremental updates of the indices with the simple approach.

4.3.1 Test Collection and Implementation Settings

Table 1: The results of the simple approach

ratio of the initial documents (%)	iP[.01]	MAiP
10	.411	.130
30	.471	.139
50	.480	.135
70	.497	.140
90	.508	.144
100 (*no-update*)	.504	.143

In the experiments, we used the INEX 2008 test collection provided by the INEX project[3]. This test collection consists of three components: 1) the INEX document collection, 2) the INEX topics, and 3) the INEX relevance assessments. The INEX document collection is an XML Wikipedia corpus based on a snapshot of the English version of Wikipedia. Approximate 660,000 articles are in the corpus. The INEX topics include 68 queries, of which 32 are CO queries and 36 are CAS queries. We use all of them in the experiments. The INEX relevance assessments are the evaluations for the queries to measure the effectiveness of XML element retrieval systems. In this test collection, at most 1,500 elements are presented as search results for each query.

In the INEX project, interpolated precision at recall level 1% (iP[.01]) is used as a formal measure of accuracy. The evaluation tool also measures mean average interpolated precision (MAiP) as average precision at 101 recall levels.

The PC that we used for the experiments has four Intel Xeon X7560 CPUs (2.3GHz), 512GB of memory, and a 4.5TB disk array, and runs Oracle Enterprise Linux 5.5. The indices are implemented using BerkeleyDB in GNU C++.

4.3.2 Experimental Procedure

We define an index before incremental updates take place as an *initial index*. We distinguish between documents used to construct initial indices (*initial documents*) and documents used to update indices (*update documents*). Here, we assume that the statistics of the documents are static, i.e., the statistics of the initial documents and update documents are the same. For this purpose, we randomly sampled documents in order to distinguish them. In Section 6.4, we consider a more complex case in which the statistics of the documents dynamically change.

All documents are processed through the stopword and stemming steps before the construction of the initial indices begins. The procedure is as follows: first, the initial documents are parsed to calculate term weights and the initial indices are constructed, and then, the update documents are obtained for updating indices incrementally. All data in the `GW-Path-term` and `GW-Path` indices are scanned in the main memory during updates, and then, the update document are parsed and the `Term` and `Tag-term` indices are updated incrementally.

4.3.3 Effects of Incremental Updates

We investigated search accuracies by changing the percentage of initial documents within the document set, as indicated in Table 1. For example, when the ratio is 30%, the initial indices are constructed using 30% of the documents in the set, and the indices are updated using the remaining 70% of the documents. When the ratio of initial documents is 100%, updates of the indices do not take place (*no-update*).

Table 1 shows that incremental updates reduce search accuracy, which demonstrates that global weights cannot be computed accurately using only a subset of the documents. To make the incre-

[3]`https://inex.mmci.uni-saarland.de/`

mental update practical, we need to solve the problem of inaccurate global weights.

Suppose the initial indices are constructed using 50% of the documents. We measured the construction time in two ways: 1) rebuilding the indices from scratch, and 2) updating the indices incrementally. It takes 5.5 h, 2.8 h, and 3.4 h to rebuild from scratch, to construct the initial indices, and to update the indices, respectively. Although the total construction time to complete the indices using the incremental update approach takes 0.7 h longer compared with the *from scratch* approach, an incremental update finishes 2.1 h faster than rebuilding the indices from scratch.

The average time for incremental updates is 37.0 ms per document when the ratio of initial documents is 50%, whereas the time required to construct indices from scratch is 29.8 ms per document. It may take a long time to index when we update a number of documents. A breakdown of the time needed to update the indices shows that execution, user, and system times are 3.4, 1.6, and 0.3 h, respectively. This indicates that I/O is a bottleneck and suggests that a method for eliminating update targets to reduce I/O will be effective for shortening the update time.

We also mention that the time to scan the global weights in the `GW-Path-term` and `GW-Path` indices (about 300 s) is less than that of calculating the global weights using all documents (about 700 s). Accessing the global weights does not incur high costs. Note that the disk space occupied by the global weights in the indices is approximately 1.3GB, while that of all indices is approximately 89GB.

5. ACCURATE TERM WEIGHTING AND FAST UPDATE OF THE INDICES

Our previous experiments showed that the simple extension system has some problems: 1) search accuracy is reduced by inaccurate global weights, and 2) it takes longer for indexing. Hence, we should handle the following requirements.

- The system retains search accuracy.
- The system attains fast update of indices.

Concerning the first requirement, global weights cannot accurately be calculated with insufficient documents because they must reflect the entire document set. To solve this problem, the simplest approach is to update the global weights as new documents are inserted. It is not always true that we can gain enough new inserted documents to calculate accurate global weights although the idea of updating global weights is reasonable. We therefore consider how to calculate accurate global weights even with a limited number of documents, and propose a solution in the next section.

Regarding the second requirement, experiments have shown that incremental updates lengthened the indexing time. Even if update time is not increased, a large number of update targets are handled in XML element retrieval, although only a few of them appear in search results. Therefore, it is effective to select update targets to reduce indexing time. We propose the *element filter* and *term filter* to select update targeted elements and terms, respectively, so that we can identify important parts, or elements, and significant terms in a document. We discuss these filters in Section 5.2.

5.1 Approximate Accurate Global Weights

We try to calculate accurate global weights using a limited number of documents. Since they are calculated within elements having the same path expression, we cannot get appropriate statistics for a path expression appearing rarely in the document set. We therefore consider a more effective approach. Concretely, we integrate path expressions with a similar property to expand the elements in the same class.

1: /article/sec
2: /article/sec/sec
3: /article/sec/emp/sec
4: /article/emp/sec
5: /article/emp/sec/sec

Figure 8: Examples of path expressions

article sec	1: /article/sec 2: /article/sec/sec
article sec emp	3: /article/sec/emp/sec 4: /article/emp/sec 5: /article/emp/sec/sec

Figure 9: An example of classification in ST

To accomplish this, we utilize the method of integrating path expressions [3] proposed in our previous study. This integration method calculates an accurate global weight for a path expression of few frequencies. The current case is similar to that of the previous study; in both cases, the global weights of elements with rare path expressions are not calculated accurately. Therefore, the integration method should show positive results.

To integrate path expressions, we regard a path expression as an array of tags and identify the path expressions that are similar to each other in terms of the appearance order and/or the appearance frequency of tags. As a result of the integration, we eliminate classes that do not contain enough elements to calculate accurate global weights.

In addition, the cost to adapt these methods is small because these approaches simply count a frequency and check an order of tags in a path expression. We can ignore the harmful effects on update efficiency. We now explain three types of integration methods:

1) set-of-tags method (ST),
2) bag-of-tags method (BT), and
3) order-of-tags method (OT)

5.1.1 Set of Tags Method (ST)

Tags in structured documents are categorized into two groups. One represents structural classifications like `article` and `sec` tags. The other indicates ideas, attributes, and specific contents like `person`, `emp`, and `table` tags. These two types of tags are supposed to be independent in their appearance. This suggests that a combination of tags can generate two or more path expressions. It is not always appropriate that these path expressions are placed into different classes. This is why we focus on relaxing appearance order and frequency of tags in path expressions to integrate similar path expressions.

The set-of-tags (ST) method relaxes both appearance order and frequency of tags in path expressions. Accordingly, we consider only the names of the tags. We classify the path expressions composed of the same tag names as the same class.

Classification of the path expressions in Figure 8 is shown in Figure 9. Path expressions 1 and 2 are in the same class because they are both composed of `article` and `sec` tags, while path expressions 3, 4, and 5 are in the same class because they are composed

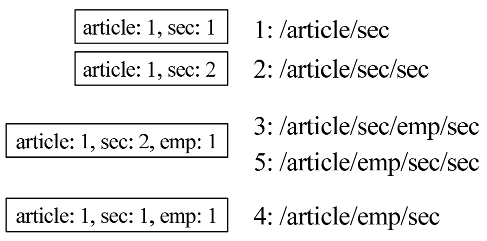


Figure 10: An example of classification in BT

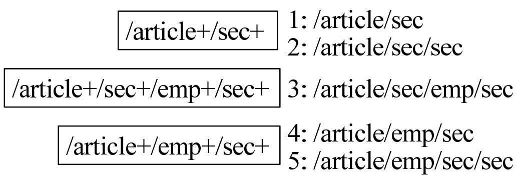


Figure 11: An example of classification in OT

of `article`, `sec`, and `emp` tags. The global weights of the elements with path expressions 1 and 2 are calculated together, while the global weights of the elements with path expressions 3, 4, and 5 are calculated together.

5.1.2 Bag-of-Tags Method (BT)

The bag-of-tags (BT) method relaxes only the appearance order of tags in path expressions. We do not consider the order of tags from the viewpoint of the bag-of-words concept.

Classification of the path expressions in Figure 8 is shown in Figure 10. We first enumerate the names and frequencies of tags in each path expression to integrate the path expressions classified as the same class. As a result, we integrate path expressions 3 and 5 because both have one `article`, two `sec`, and one `emp` tags.

5.1.3 Order-of-Tags Method (OT)

The order-of-tags (OT) method relaxes only appearance frequency of sequential tags in a path expression. In some path expressions, a tag appears consecutively two or more times, for example, `col` tags in `table` of HTML. In this case, even if the frequencies of the same tag appearing consecutively are different, we suppose that the features of a path expression are not so different because the semantics of each tag are fixed. Therefore, if consecutive tags are the same, such tags can be aggregated into one.

Classification of the path expressions in Figure 8 is shown in Figure 11. Since `sec` tags appear consecutively in path expressions 2 and 5, path expressions 1 and 2 are integrated. Path expressions 4 and 5 are also integrated because path expressions 1 and 2 have one or more `article` tags followed by one or more `sec` tags, whereas expressions 4 and 5 have one or more `article` tags followed by one or more `emp` tags, and one or more `sec` tags.

5.2 Filters for Reducing Update Cost

It costs much to calculate all term weights and store them in the indices. We therefore identify and exclude unnecessary elements and terms that should not be indexed.

We select the elements to be calculated as targets to decrease the calculation cost. More precisely, we eliminate elements that might not be appropriate for search results.

It takes a long time and much disk space if all terms are indexed. To avoid this, we identify the terms regarded as effective ones before they are indexed.

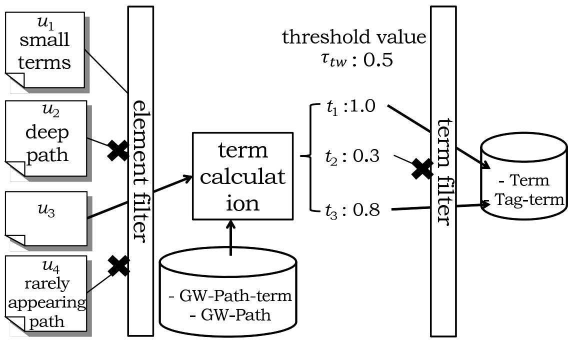


Figure 12: The element filter and the term filter

For this purpose, we propose the *element filter* and *term filter*, to attain fast indexing.

It is obvious that we can reduce update cost with these filters. However, search accuracy will be reduced if we remove elements and terms relevant to any query. This would violate the first requirement. To avoid a decrease in search accuracy, we should decide carefully which elements and terms can be removed.

5.2.1 Element Filter

It is meaningless to calculate the term weights for elements that are not presented in search results. We remove these irrelevant elements to reduce the update cost.

We consider characteristics of irrelevant elements and observe that information outside of the main content is not informative. Irrelevant elements include:

1) elements containing few terms;
2) elements with "deep" path expressions; and
3) elements with rare path expressions.

For example, additional information like a table of contents and references contain few terms, and data content like a `table` tag in HTML has a deep document structure.

Moreover, if a path expression appears rarely in the document set, the path expression does not represent an important structure. To identify informative path expressions, we use Zipf's law [11] to get the threshold number of middle frequency f, which is computed as follows:

$$f = \frac{\sqrt{8F_1 + 1} - 1}{2} \quad (2)$$

where F_1 is the number of the path expressions appearing only once in the document set.

To retain search accuracy, we seek appropriate threshold values to remove only irrelevant elements. Preliminary experiments for the element filter are described in Section 6.2.

We illustrate the behavior of the element filter in Figure 12. Suppose that four elements, u_1, u_2, u_3, and u_4, are extracted from inserted documents. Elements u_1, u_2, and u_4 are eliminated by the element filter because u_1 is too short, the path expression of u_2 is too deep, and the path expression of u_4 rarely appears. As a result, only u_3 is chosen as a target.

5.2.2 Term Filter

Although there are many potential candidates for search results, only a few elements are presented as search results. Therefore, we suppose that it does not affect search accuracy significantly if indices do not contain terms with low weights.

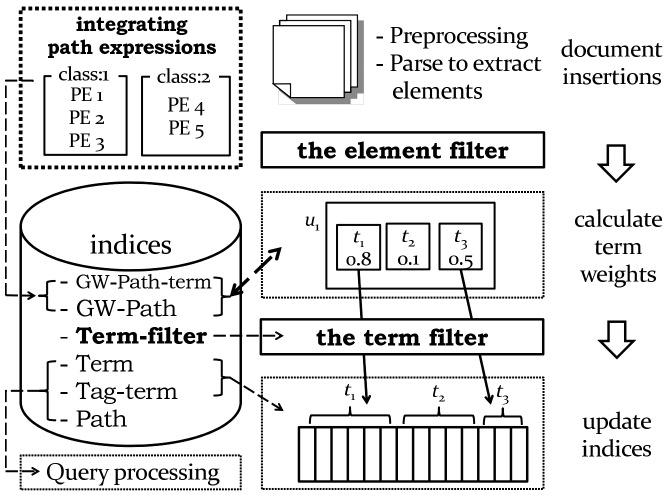


Figure 13: Architecture of the proposed system

Based on this idea, we remove these unimportant terms with the term filter. Threshold values τ_{tw} are defined as the term weights of the k-th largest term of each tag-term pair contained in indices. These values are stored in the `Term-filter` index so that they can be looked up quickly.

However, even terms with low term weights may affect the ranking of search results and search accuracy. We should treat the value k carefully to avoid decreasing search accuracy. Preliminary experiments for the term filter are described in Section 6.2.

Figure 12 shows an example of how the term filter works. Suppose that τ_{tw} is 0.5 and there are three terms to be inserted into the `Tag-term` index and the `Term` index. We use the unitary value of τ_{tw} for simplicity although τ_{tw} differs for each tag-term pair. Terms t_1 ($1.0 > \tau_{tw}$) and t_3 ($0.8 > \tau_{tw}$) are indexed successfully because they are greater than τ_{tw}. In contrast, term t_2 ($0.3 < \tau_{tw}$) is not indexed because it is less than τ_{tw}.

5.3 Architecture of the Proposed System

Figure 13 shows the architecture of the proposed system. The main differences between the simple extension system and the proposed system are that the proposed system integrates path expressions for calculating accurate global weighs, and utilizes two filters, i.e., the element filter and the term filter, to reduce update cost. The `Term-filter` index contains the threshold values for the term filter.

The query processing part is the same as in the simple approach. The global weights in the `GW-Path-term` and `GW-Path` indices are updated after re-calculating them using integrated path expressions when documents are inserted.

When updating documents, the proposed system treats only the elements and the terms selected by the two filters, unlike the simple extension system. First, a term weight is calculated which goes through the element filter. Figure 13 shows that element u_1 gets through the element filter, and terms in the element are calculated. Next, the terms that get through the term filter are indexed. Terms t_1 and t_3 are indexed because they get through the term filter, as indicated in Figure 13.

6. EXPERIMENTAL EVALUATIONS

6.1 Experimental Design

We investigate whether integrating path expressions and the two filters are effective for search accuracy and efficient for updating the indices, compared with the simple extension system as a baseline.

The experimental environment is the same as the one used for the simple extension system. The proposed methods are evaluated using two document sets, one with static statistics in which topics rarely change, and the other with dynamic statistics in which new topics are added drastically.

Our proposed approaches have some variations. There are four ways of calculating global weights: path expression-based classification as a default option, the set-of-tags method (ST), the bag-of-tags method (BT), and the order-of-tags method (OT). There are three parameters in the element filter: element length threshold τ_{el}, path length threshold τ_{depth}, and Zipf's threshold τ_{Zipf}. By examining the effectiveness of each approach, we can choose the best setting.

In our experimental procedure, we first ran some preliminary experiments to tune the parameters of the element filter and term filter. Next, with these tuned parameters, we measured average update time per document, index size, and search accuracy using each variation of the proposed methods. We used the document set with static statistics and chose 50% for the ratio of initial documents. Finally, we confirmed the effectiveness of the proposed methods using dynamic statistics.

We mainly used the BM25E [7] term-weighting scheme because it is one of the most effective ones for XML element retrieval, but we also used the TF-IPF and the query likelihood model to show that the proposed methods can be applied to other term-weighting schemes.

Table 2: Accuracies with changing τ_{el}

τ_{el}	25	30	35	40	45	50	55
iP[.01]	.526	.530	.541	.527	.526	.532	.527

Table 3: Depth of PEs and the ratio of elements

τ_{depth}	3	4	5	6	7
All	.19	.44	.69	.88	.96
Top	.69	.89	.97	.99	1.00

6.2 Effects of the Element Filter and Term Filter

The element filter eliminates elements having extremely small element length, extremely deep path expressions, and rarely appearing path expressions. In this section, we describe some experiments we conducted to decide their threshold values.

According to the results shown in Table 2, we set τ_{el} to 35, because the best parameter for the element length, in terms of search accuracy, is 35.

Table 3 shows the ratio of the elements whose depth of path expressions is less than or equal to τ_{depth} for all elements in the test collection. We also measured the ratio of the same statistics appearing only in highly ranked results, obtained in our previous study [4]. There is a difference between the result of all elements and that of highly ranked elements. This indicates that we can extract useful elements and only those, if the threshold depth is set to extract as many highly ranked elements as possible and to discard useless elements. We set τ_{depth} to 6, which ignores elements whose depth is 6 or more.

We also investigated the threshold values of Zipf's law. We computed the value of the middle frequency of path expressions using Equation 2. We set τ_{Zipf} to 166, which ignores the elements whose

Table 4: Effects of the term filter with changing *k*

k	no filter	1500	5000	10000	15000	30000
iP[.01]	.480	.477	.483	.490	.462	.484

Table 5: Effects of the proposed approaches

run ID	update time (ms/doc)	disk size(GB)	iP[.01]	MAiP
no-update	29.8	89	.504	.143
baseline	37.0	88	.480	.135
ST	37.2	88	.513	.135
BT	38.3	88	.501	.134
OT	37.7	88	.506	.142
τ_{el}	27.3	71	.486	.140
τ_{depth}	34.4	77	.484	.139
τ_{Zipf}	31.4	79	.487	.140
elem filter	27.3	73	.487	.140
term filter	37.6	80	.490	.132
two filters	29.0	67	.491	.133
ST_filters	35.3	82	.505	.135

path expressions appear 166 times or less in the initial index.

In the same manner as the element filter, the term filter eliminates terms whose weights are less than the threshold value. We conducted an experiment to decide the threshold values for the term filter as shown in Table 4. We set *k* of τ_{tw} to 10000, which ignores the terms whose weights are less than the 10,000th largest weight of each tag-term pair.

6.3 Evaluations of the Document Set Using Static Statistics

We measured average update time per document, the size of indices, and search accuracy with each variation of the proposed methods, as indicated in Table 5. Note that in the case of *no-update*, or the *from scratch* approach, the average update time replaces the construction time of initial indices.

Compared with the simple baseline system, the iP[.01]'s of ST, BT, and OT are improved. ST and BT are even more accurate than *no-update*. As a result, ST is the most effective for calculating accurate global weights. In contrast, the update efficiencies of these methods are slightly reduced.

As for the element filter, all components of the element filter τ_{el}, τ_{depth}, and τ_{Zipf} save update cost without reducing search accuracy. The combination of τ_{depth} and τ_{Zipf} is the most effective and shows 26% faster updates than the baseline approach. We used this setting for the element filter in the following experiments.

Although the term filter reduces the amount of disk space utilized and does not decrease search accuracy, updates take longer than in the baseline approach. This is because only a small number of terms are omitted for maintaining search accuracy. Therefore, the term filter would show better results when used in a Top-*k* search.

Next we evaluated the combination of the two filters. This approach required more time to update indices than the element filter only, although it retains search accuracies and decreases index size.

We combined ST and the element filter as ST_filters. The search accuracy improved to the extent of *no-update*. However, this method does not show significant results for reducing update time.

We conclude that the element filter approach is a well-balanced one, whereas the ST_filters approach is the most accurate.

In terms of query efficiency, each method took 1.5 to 2.0 s per query, which would be acceptable for users.

Table 6: Category and Query

Category name	CQ	CW
Technology and applied sciences	18	54
Culture and the arts	20	51
Natural and physical sciences	9	24
Society and social sciences	4	13
History and events	4	11
Philosophy and thinking	3	8
General reference	3	7
Health and fitness	2	7
People and self	3	6
Geography and places	2	5
Mathematics and logic	0	0
Religion and belief systems	0	0

We omit the experimental results of the TF-IPF and query likelihood tests because they are about the same as the BM25E results.

6.4 Evaluations of the Document Set Using Dynamic Statistics

In the previous evaluations, we assumed that term distribution and term statistics are static. However, on the Web new topics can emerge suddenly and it may change the term distribution drastically. Here we artificially assemble a document set using dynamic statistics to investigate the effectiveness of the proposed methods.

In this set, the initial documents do not contain a certain topic but the update documents do include the topic. We outline the steps to evaluate: 1) identify documents on a certain topic; 2) construct the initial index using the other documents; and 3) update the indices incrementally using the documents related to the topic.

We utilized the categories in Wikipedia to judge if a document belongs to a certain topic. In Wikipedia there are many kinds of categories of various sizes; twelve major categories are shown in Table 6. We classified 68 queries into the twelve categories. Each query contains one to five query keywords, and we obtained a keyword set for each category. Since the categories of "Technology and applied sciences" (*technology* for short) and "Culture and the arts" (*culture* for short) include relatively large numbers of queries (Category Query, CQ for short) and query keywords (Category key-Words, CW for short), we used these categories in the evaluation. We assigned a document to a certain category if the document contains the category words. Note that these category words are stemmed.

CW of *technology* : *aircraft, applied, automobil, aviat, bay, bletchlei, break, car, code, colossu, compani, comput, databas, detect, engin, expert, file, filter, format, graphic, imag, inform, instal, intrus, invent, java, languag, linux, manag, mechan, metadata, mine, motor, museum, network, nikola, open, oper, park, patent, program, raid, record, retriev, rotari, secur, social, sourc, storag, system, tata, tesla, virtual, wireless*

CW of *culture* : *acquisit, africa, al, basketbal, berber, bilingu, childbirth, children, classic, countri, cultur, danc, dish, europ, european, fiction, film, food, franc, game, guitar, hors, instrument, japanes, keyboard, languag, mahler, museum, nba, north, person, picasso, player, portugues, produc, region, rule, scienc, scrabbl, song, spanish, style, symphoni, tap, tast, terracotta, tradit, typic, vegetarian, vodka, wine*

We used the category queries only to examine the effectiveness of the proposed methods, because we know the effects under dy-

Table 7: Effects on emerging a new topic

# of indexed doc. ($\times 10^4$ doc.)	*technology (iP[.01])*			
	baseline	ST	elem	ST_elem
37 (25% updated)	.332	.378	.246	.376
47 (50% updated)	.339	.415	.341	.414
56 (75% updated)	.372	.420	.338	.419
66 (100% updated)	.389	.448	.424	.446

# of indexed doc. ($\times 10^4$ doc.)	*culture (iP[.01])*			
	baseline	ST	elem	ST_elem
31 (25% updated)	.321	.358	.319	.358
43 (50% updated)	.311	.376	.341	.376
54 (75% updated)	.334	.397	.365	.397
66 (100% updated)	.338	.394	.363	.398

namically changing statistics of term distributions. In this situation, we assumed that users expect that effective search is available as soon as new topics are added to the collection.

The number of the documents in the initial indices of *technology* and *culture* are 280,000 and 200,000, respectively. We evaluated the effects of changing statistics at four points during the updates. After all updates, the number of indexed documents reached 660,000 for both categories.

Table 7 shows the iP[.01]'s of the approaches of the baseline, ST, the element filter, the combination of ST and the element filter (ST_elem), in both categories. The proposed methods attained better search accuracies than those of the baseline in both categories. In particular, ST and ST_elem increased the search accuracies rapidly even when the number of update documents is small.

In addition, the fact that even the element filter approach overwhelms the baseline indicates that updating the global weights is useful when the statistics of the documents dynamically change, or new topics are added to the collection.

7. CONCLUSION

In this paper, we proposed the methods of fast incremental updates on indices for XML element retrieval to attain both effectiveness and efficiency in the query processing. The simple solution for incremental updates has two problems: 1) decreased search accuracy, and 2) increased update time. We solved these problems by integrating path expressions and utilizing two filters for excluding unnecessary data.

The experimental evaluations showed that our proposed approaches are effective and efficient for both static statistics and dynamic statistics. In particular, a variation of the proposed approaches can save 26% of update time while maintaining search accuracy compared to the simple extension system and static statistics.

With a Top-*k* search, the term filter can be used more effectively. Implementing a Top-*k* search remains part of our future work. Proposing a method of incremental updates specialized for document modifications is also one of our future projects.

8. ACKNOWLEDGEMENTS

This work was partly supported by Grant-in-Aid for JSPS Fellows and JSPS KAKENHI Grant #22240005, #23500121, and #22700248.

9. REFERENCES

[1] Fei Chen, Xixuan Feng, Christopher Ré, and Min Wang. Optimizing Statistical Information Extraction Programs Over Evolving Text. In *Proc. of the 28th IEEE ICDE*, 2012.

[2] Yu Huang, Ziyang Liu, and Yi Chen. Query Biased Snippet Generation in XML Search. In *Proc. of ACM SIGMOD*, pages 315–326. ACM, 2008.

[3] Atsushi Keyaki, Kenji Hatano, and Jun Miyazaki. Relaxed Global Term Weights for XML Element Search. In *Formal Proc. of INEX 2010 Workshop*, volume 6932 of *LNCS*, 2011.

[4] Atsushi Keyaki, Kenji Hatano, and Jun Miyazaki. Result Reconstruction Approach for More Effective XML Element Search'. *International Journal of Web Information Systems (IJWIS)*, 7(4):360–380, 2011.

[5] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *Proc. of the 27th Australasian conference on Computer Science*, 2004.

[6] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective Keyword search in Relational Databases. In *Proc. of ACM SIGMOD*, 2006.

[7] Wei Liu, Stephen Robertson, and Andrew Macfarlane. Field-Weighted XML Retrieval Based on BM25. In *Formal Proc. of INEX 2005 Workshop*, volume 3977 of *LNCS*, 2006.

[8] Ziyang Liu and Yi Chen. Identifying Meaningful Return Information for XML Keyword Search. In *Proc. of ACM SIGMOD*, pages 329–340. ACM, 2007.

[9] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval*, pages 157–159. Cambridge University Press, 2008.

[10] Giorgos Margaritis and Stergios V. Anastasiadis. Low-cost Management of Inverted Files for Online Full-Text Search. In *Proc. of 18th ACM CIKM*, 2009.

[11] M. E. Maron. Automatic Indexing: An Experimental Inquiry. *Journal of the ACM*, 8:404–417, 1961.

[12] Thomas Neumann and Gerhard Weikum. xRDF3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. In *Proc. of 36th VLDB*, pages 256–263, 2010.

[13] Paul Ogilvie and Jamie Callan. Parameter Estimation for a Simple Hierarchical Generative Model for XML Retrieval. In *Formal Proc. of INEX 2005 Workshop*, volume 3977 of *LNCS*, 2006.

[14] Benjamin Piwowarski and Patrick Gallinari. A Bayesian Framework for XML Information Retrieval: Searching and Learning with the INEX Collection. *Journal of Information Retrieval*, 8(4):655–681, 2005.

[15] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On Querying Historical Evolving Graph Sequences. In *Proc. of the 37th VLDB*, 2011.

[16] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. *The VLDB Journal*, 17(1):81–115, 2008.

[17] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *Proc. of ACM SIGMOD*, 1994.

[18] Andrew Trotman, Xiang-Fei Jia, and Shlomo Geva. Fast and Effective Focused Retrieval. In *Formal Proc. of INEX 2009 Workshop*, volume 6203 of *LNCS*, 2010.

[19] Andrew Trotman and Börkur Sigurbjörnsson. Narrowed Extended XPath I (NEXI). In *Formal Proc. of INEX 2004 Workshop*, volume 3493 of *LNCS*, 2005.