Fast Incremental Indexing with Effective and Efficient Searching in XML Element Retrieval

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

In this paper, we propose methods for fast incremental indexing with effective and efficient query processing in XML element retrieval. The effectiveness of a search system becomes lower if document updates are not handled when these occur frequently on the Web. The search accuracy is also reduced if drastic changes in document statistics are not managed. Though it will be important to enable fast updates of indices, preliminary experiments have shown that a simple incremental update approach has two problems: some kinds of statistics are inaccurate, and it takes a long time to update indices. We therefore propose two methods: one to approximate term weights accurately with a small number of documents, even for dynamically changing statistics; and the other to eliminate unnecessary update targets. Experimental results show that our proposed system can update indices up to 32% faster than the simple incremental updates while while the search accuracy improved by 4% compared with the simple approach. The proposed methods can also be fast and accurate in query processing, even if document statistics change drastically.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2013 ACM 978-1-4503-1306-3/12/12 ...$15.00.

An information unit in XML element retrieval is not a document but an element of XML documents. XML element retrieval systems present elements that contain descriptions satisfying the information needs of users who thus do not have to spend time seeking relevant descriptions in each document. Although existing studies of XML element retrieval have attained both effectiveness and efficiency in query processing [20], [22], [12], [5], these studies have not considered document updates.

Web documents are frequently updated; i.e. inserted, deleted, or modified. In particular, Wikipedia articles are updated 4000 to 8000 times per 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 it ranks documents based on obsolete information, which reduces the effectiveness. Thus, we add a function for handling document updates to the existing techniques for XML element retrieval.

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 that this is the first study focused on fast incremental updates of indices in effective and efficient XML element retrieval systems. Although Google supports fast incremental updates with effective and efficient query processing, its approach differs from ours. Google analyses the link information of Web pages to find important pages, whereas our study utilizes text information. We can apply our approach to other structured documents apart from the Web, even if these 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 calculated with various kinds of statistics, including *global weights* that are aggregate statistics derived from all documents in a document set. Thus, global weights are difficult to calculate immediately. These statistics were not contained in the indices of past studies, because those studies did not target index updates. We incorporate these statistics into the proposed indices to enable fast updates. Two problems arise with incremental updates:

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

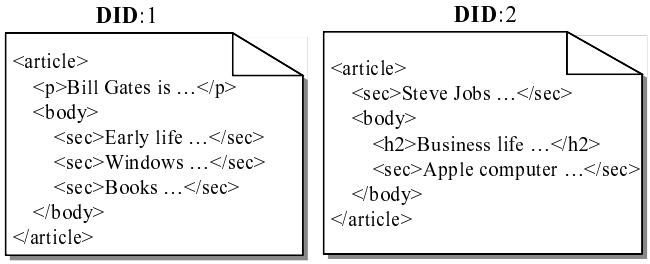

Figure 1: XML document

- the 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, global weights cannot be calculated accurately without a sufficient number of documents. We need a method to approximate global weights accurately with an insufficient number of documents. Such a method would be especially useful as new topics are added and document statistics change drastically. The statistics related to the new topics vary as the documents are updated and must be recalculated rapidly.

The second problem implies that incremental updates prolong the indexing time. Moreover, in XML element retrieval, a much larger number of search targets must be handled in comparison to the number of documents[2]. As a result, the cost to index all data is high. To enable fast updates, we target only the important parts of a document (elements) with an *element filter*, and we target important terms for query processing with a *term filter*.

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

In the remainder of this paper, Sections 2 and 3 describe the basic concept of XML element retrieval and the related studies, respectively. Section 4 implements and evaluates a simple system of extended XML element retrieval for incremental updates of indices. Section 5 discusses the proposed methods that the simple system applies to solve the problems, and Section 6 discusses the experimental evaluations. Section 7 concludes this paper.

2. XML ELEMENT RETRIEVAL

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

2.1 XML Element

We give specific examples in Figures 1–3 to define XML elements. Figure 1 illustrates XML documents. Each document is assigned a document identifier (DID). Figure 2 depicts trees abstracted from Figure 1. An XML document can be presented by a tree structure, which helps to understand the structure of the document. Each element is assigned an element identifier (EID), which is assigned in document order. We can identify an element using its DID and EID.

A pair of start and end tags represents an XML element node within an XML tree, and the nested structure of XML elements represents ancestor–descendant relationships. Each element in Figure 3 is the text that comprises a set of text nodes within the XML tree in Figure 2. We describe the path expression (PE) of each element.

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

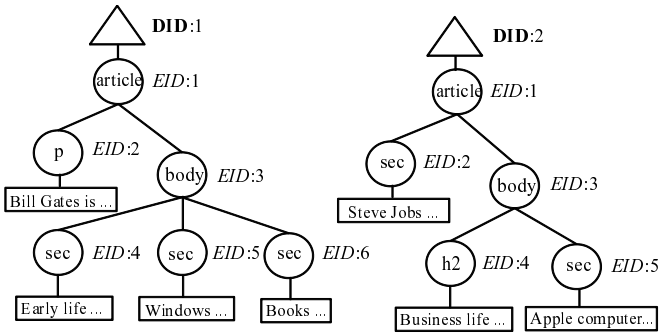

Figure 2: XML tree

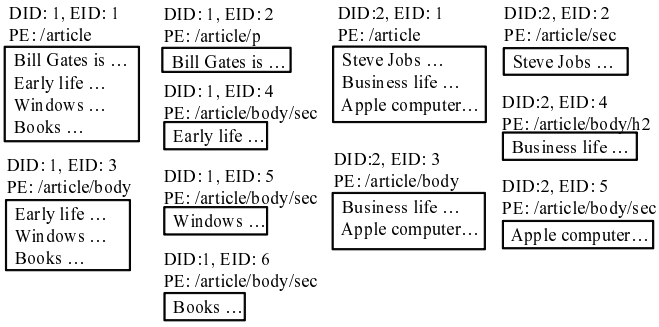

Figure 3: XML element

Authors add structures to a document: e.g. chapters, sections, and paragraphs. We utilize these structures to identify the best material for satisfying the information needs for users. Some structures are meaningless, so elements defined by those structures are inappropriate as search results, and some existing studies [4], [3] include attempts to eliminate these. Suppose that a user seeks information from Document 1 about "Early life . . . ", "Windows . . . ", and "Books · · · ". XML element retrieval systems try to present the user Element 3 of Document 1, because that element contains all of the information that the user needs and no extra information.

2.2 Queries for XML Documents

There are two ways for expressing an information need in a query: through keywords and through document structure. A query entirely composed of query keywords is called a content-only (CO) query, whereas a query composed of pairs of query keywords and a constraint on the document structure is called a content-and-structure (CAS) query.

CO queries are used just as in traditional information retrieval for text documents. Users can submit CO queries even if they do not know the structures of the documents that are retrieved. In contrast, CAS queries utilize one of the most significant features of structured documents: i.e. document structure. With a CAS query, a user can obtain specific results with regard to granularity and content.

We give a specific example of a CO query and a CAS query. These are expressed in the narrow extended XPath I (NEXI) [23] query language. A CO query `//*[about(., "Windows")]` means that the candidate search results are elements containing "Windows". Elements 1, 3, and 5 of Document 1 can be search

results in Figure 3.

A CAS query: `//article[about(., "Steve")]//sec[about(., "Apple")]` is more complex. Let us focus on the first half of the query, `//article[about(., "Steve")]`, which means that candidates for this part are elements that contain "Steve" and whose path expressions end with an `article` tag. The second half of the query, `//sec[about(., "Apple")]`, means that candidate search results are elements that contain "Apple" and whose path expressions end with a `sec` tag. The search results are elements that satisfy the latter constraint and whose ancestor elements satisfy the former constraint. The only element satisfying the query constraints is Element 5 of Document 1 in Figure 3.

3. RELATED STUDIES

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

3.1 Effective and Efficient XML Searches

3.1.1 Effective XML Searches

The most important goal of XML element retrieval is highly accurate searches. The mainstream approach to extracting relevant elements is as follows: first, calculate a term weight for each element by using a term-weighting scheme; next, compute a score for each element using these term weights.

Term-weighting schemes for XML element retrieval are often derived from studies on document retrieval. Both of these are composed of three types of factors: local weights that are statistics derived from each document (element); global weights that are statistics derived from all document in a document set; and constant values (coefficients and parameters). 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 these.

The most significant difference between document retrieval and XML element retrieval is the method for computing global weights. 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. However, 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 classify elements by path expression. In Figure 3, since Elements 4, 5, and 6 of Document 1, and Element 4 of Document 2 all have the same path expression `/article/body/sec`, the global weights are calculated using these elements.

Alternatively, elements with the same tag can be placed in the same class. Because Elements 4, 5, and 6 of Document 1 and Elements 2 and 4 of Document 2 all have the `sec` tag, the global weights are calculated using these elements, as depicted in Figure 3. We use classification based on path expression in our system, because this is reportedly more accurate [18].

There are several kinds of term-weighting schemes for XML element retrieval; e.g. TF-IPF [10], BM25E [11], and the query likelihood model for XML element retrieval [17] (QLMEL). BM25E is regarded as a more effective term-weighting scheme than TF-IPF. Actually, most of the top-ranked search systems at INEX use BM25E [2]. However, no exhaustive comparison between BM25E and QLMEL has been explored. We therefore examine the potentials of these term-weighting schemes in this article.

BM25E [11] is a probabilistic model. In a term calculation of the classic term-weighting scheme TF-IPF, statistics on the occurrence frequencies of terms are utilized. Conversely, BM25E utilizes not only the statistics but also element length (the number of terms in an element). The term weight $w_{bm25e}(p,e,t)$ of term t in element e with path expression p is calculated as follows:

$$w_{bm25e}(p,e,t) = \frac{(k_1+1)tf_{e,t}}{k_1((1-b)+b\frac{el_e}{avel_p})+tf_{e,t}} \cdot \log\frac{N_p - pf_{p,t}+0.5}{pf_{p,t}+0.5} \quad (1)$$

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 elements with p, el_e is the length of element e, and $avel_p$ is the average length of the elements with p. The parameters k_1 and b are set as commonly used values 1.2 and 0.75, respectively. Moreover, $s_{bm25e}(e)$ is the score of e and is calculated as follows:

$$s_{bm25e}(e) = \sum_{t_i \in T} w_{bm25e}(p,e,t_i) \quad (2)$$

where T is a set 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 [13] is well studied and achieves significant results. This model has been adapted to XML element retrieval [17]. In the term-weighting scheme, the score of each element is the product of the occupancy probabilities of the query keywords as shown in Eq. (6). This means that non-zero values are computed only for the elements containing all the query keywords. To avoid this, smoothing techniques are often used. Smoothing values are computed not with a document (element) model but with a background language model, which is applied for an entire document set.

Let $w_{qlm}(p,e,t)$ be a probability that term t is generated in element e (i.e. a term weight), and let $s_{qlm}(p,e,t)$ be a score of element e. These are calculated as follows:

$$w_{qlm}(p,e,t) = \omega\hat{P}_{mle}(t|M_e) + (1-\omega)\hat{P}_{mle}(t|M_p) \quad (3)$$

$$\hat{P}_{mle}(t|M_e) = \frac{tf_{e,t}}{N_p} \quad (4)$$

$$\hat{P}_{mle}(t|M_p) = \frac{\sum_{e\in p} tf_{e,t}}{\sum_{e\in p} el_e} \quad (5)$$

where ω is a given parameter, M_e is an element model for e, M_p is a background language model for p.

Moreover, $s_{qlm}(e)$ is the score of e and is calculated as follows:

$$s_{qlm}(e) = \hat{P}(T|M_e) = \prod_{t\in T} w_{qlm}(p,e,t) \quad (6)$$

3.1.2 Efficient XML Searches

Although the most important requirement of XML element retrieval is enabling effective searches, fast query processing is also required by system users.

To attain efficient XML element retrieval, various approaches have been taken, such as 1) compressing and reducing data to suppress the index size to minimize the amount of data scanned in query processing, and 2) applying top-k algorithms to return search results quickly. Many top-k searches have been proposed [6]. There are two conditions for efficient query processing: 1) term weights are calculated before query processing begins, and 2) terms are sorted in descending order of weight. This means that we only need to scan highly ranked terms in query processing. Note that

some query processing methods also utilize an index for a random scan, which is used to refer to the weight of an arbitrary term in any element.

Some studies [20], [22] have used term-weighting schemes [11] for effective searches. Theobald et al. proposed two types of indices and a top-k algorithm for efficient searches [20]. One type is for scoring an element in query processing, and the other type is for checking a structural constraint on a query. They also proposed cost-based query processing, which identifies an effective moment to check the structural constraints and determines which query keyword is reasonable to process.

Trotman et al. proposed a low-cost method of data compression and selection [22].

In these studies, the aim was to retrieve elements satisfying the information need of a user by retrieving elements from a fixed document set; i.e. document updates were not considered.

3.2 Handling Document Updates on the Web

The handling of document updates is especially important in Web search systems because documents are constantly inserted, deleted, and modified. When documents are updated, useful search systems should treat them as search targets immediately. If systems present search results based on a past snapshot of the Web, the content of the Web documents may since have 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] tackled this challenge in the field of information extraction. They reported that a long processing time is required to apply information extraction techniques to document collections when document updates occur. As a result, a delay occurs before information extracted from the updated documents is available. To shorten the delay, they proposed a method for recycling the intermediate results of past snapshots. Neumann et al. [16] also effectively utilized the information of past snapshots, but with Resource Description Framework (RDF) data.

Ren et al. [19] preserved not only the latest graph data but also past snapshots to trace the transition of the graph. Our study is different from theirs to the extent that we present the information along with the latest state of the Web.

The aforementioned studies utilized the intermediate results of past snapshots. Hence, we also utilize those 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 query processing. Therefore, this is the first study to tackle the problem. Although some researches have focused on incremental updates of an inverted index [21], [9], [14], they proposed index data structures of indices and physical storage methods. Our study differ from their studies because we introduce a function for incremental updates of indices for several purposes in XML element retrieval by proposing an efficient method of data management.

4. SIMPLE EXTENSION FOR INCREMENTAL UPDATES

We propose an XML element retrieval system that can return appropriate search results even when document updates occur. We call this system a simple extension system because a function for incremental updates of indices is extended to general XML element retrieval systems such as [20] and [22].

We show the architecture of the simple extension system in Figure 4. General XML element retrieval systems have functions for index construction and query processing. They do not assume that document updates occur once indices are constructed. In contrast, the simple extension system has capabilities for document insertion, deletion, and modification. In addition, functions for constructing indices and query processing in the simple extension system are the same as the functions of the general systems, though the simple extension system stores global statistics that the general systems do not. We describe proposed index structures, query processing, and document update processing in Sections 4.1, 4.2, and 4.3, respectively.

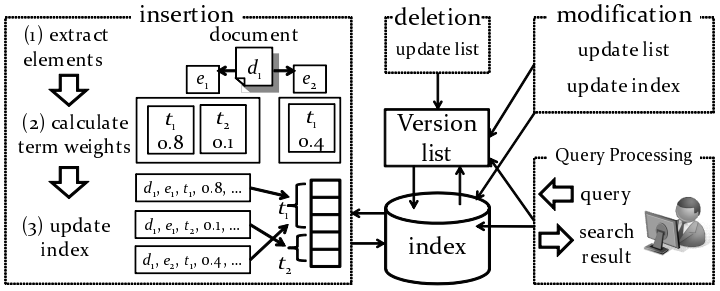

Figure 4: Architecture of the simple approach

- `Term` (DID, EID, term, term weight, Path ID, element length)
- `Tag-term` (DID, EID, tag, term, term weight, Path ID, element length)
- `RS` (DID, EID, term, term wight)
- `Path` (Path ID, path expression)
- `GW-Path-term` (Path ID, term, frequency [, values of background language model for QLMEL])
- `GW-Path` (Path ID, frequency, total length)
- `Term-filter` (tag, term, threshold value)

Figure 5: Structure of the indices

4.1 Structures of Indices

We show the structures of the proposed indices in Figure 5. In many existing studies, the term weights stored in the indices are calculated beforehand, and structural constraints can be checked with these. The proposed indices inherit these capabilities but also contain global weights to calculate a term weight immediately, which is essential for fast incremental updates.

As in the related studies [20], [22], the structures of the indices are defined in an RDB format. Primary keys are underlined. The `Term`, the `Tag-term`, the `RS`, and the `Path` indices are used for efficient and effective query processing as in other studies.

In the `GW-Path-term` and the `GW-Path` indices, the global weights are indexed. To calculate a term weight, we utilize local weights, global weights, and constant values. In Eq. (1), (4), and (5), $tf_{e,t}$ and el_e are local weights that are easy to calculate because they are derived from each element. Conversely, $pf_{p,t}$, N_p, $avel_p$, and $\hat{P}_{mle}(t|M_p)$ are global weights that are difficult to calculate immediately because the entire document set must be scanned to compute them. Therefore, we store global weights in these indices for fast term calculation.

The `Term-filter` index is used with the term filter, which is one of the proposed methods. Because this simple approach does not need the `Term-filter` index, we discuss it below in Section 5.2.2.

4.2 Top-k Searches

The simple extension system has a function for top-k searches [6]

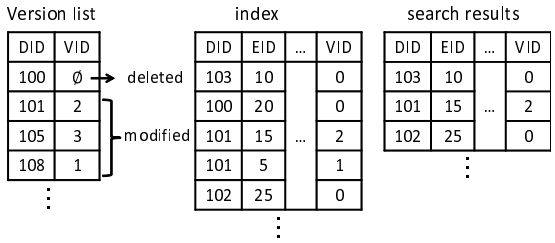

Figure 6: the version list and query processing

to enhance its usability in fast query processing. To return search results, only the top k tuples are retrieved for each term (a pair of tag and term for CAS queries). The term weights in the tuples are summed for each element to calculate scores. Furthermore, a weight in an arbitrary term in any element can be gained with a random scan when we need to calculate exact scores for search results. We can attain not only efficient query processing but also effective query processing with the random scan.

A CO query retrieves the `Term` index whereas a CAS query retrieves the `Tag-term` sequentially to extract candidate search results in query processing. Accurate scores are calculated for elements by a random scan with the `RS` index. Note that tuples in the `Term` index are grouped by term in descending order of term weight, whereas tuples in the `Tag-term` index are grouped by pair of tag and term. When a CAS query contains two or more structural constraints, the path expressions of elements must be checked to determine whether these satisfy the query constraints.

4.3 Handling Document Updates

4.3.1 Document Insertion

When a document is inserted, the updating process is conducted as follows:

(1) extracting elements from the inserted document,

(2) calculating term weights for the elements,

(3) updating indices.

Figure 4 describes each process in detail.

First, the document is parsed and elements are extracted. As a result, elements e_1 and e_2 are extracted.

Second, the term weights of t_1 and t_2 in e_1, and t_1 in e_2 need to be calculated. Term weights are calculated immediately with the `GW-Path-term` and the `GW-Path` indices.

We only need to store all kinds of global weights in the indices when using another term-weighting scheme requiring other statistics.

Finally, the `Tag-term`, the `Term`, and the `RS` indices are updated incrementally after the term weights are calculated.

Note that an entire set of documents can be updated at once to reduce the I/O cost.

4.3.2 Document Deletion

When a document is deleted, there is a high cost to find and delete all tuples related to the document because the tuples are spread across the indices. We therefore take another approach to reduce the cost of the deletion. We manage the DIDs of deleted documents instead of deleting tuples in the indices. Then, we simply ignore the tuples of the DIDs in query processing. With this approach, we can reflect the deletion immediately.

We prepare a version list to manage the deleted documents. The list contains pairs. Each pair contains the DID of the deleted document and the version identifier (VID) with its value marked as $\emptyset$. We overwrite the VID as $\emptyset$ when the DID of the deleted document is contained. Specifically, Document 100 in Figure 6 has been deleted because the VID of Document 100 is $\emptyset$.

The tuples of documents deleted in the indices are eliminated when the load average is low. After eliminating the tuples, the DIDs of the documents deleted in the version list are also eliminated.

4.3.3 Document Modification

The modification process is achieved through the deletion and insertion processes. In more detail, we delete all tuples related to the modified document and insert the latest version of the document to handle the modification. We also utilize the version list to manage the version of the document, because there is a high cost to delete the tuples of a modified document immediately. To enable fast modification, we only target the tuples of the latest version in query processing.

Note that the granularity of modification is document granularity, because some problems arise with element granularity. One of the problems is the size of the version list. The overhead in query processing become greater when we manage not documents but elements. Another problem is the difficulty in mapping old structure to new structure when the document structure changes. These are the reasons that we adopt the document as the granule of modification.

The modification process is conducted as follows: first, when a modification occurs, the version list is scanned to determine whether the DID of the modified document is contained. If the DID exists in the version list, 1 is added to the VID; otherwise, the DID of the modified document and its VID value of 1 are inserted. For example, Document 101 in Figure 6 has gone through modification twice because its VID is 2.

Second, the `Term`, the `Tag-term`, and the `RS` indices are updated in the same manner as for document insertion. As shown in Figure 5, each tuple contains a VID whose value is the same as that written in the version list. Note that the VID of the first document inserted is 0.

Finally, each tuple is checked to determine whether the tuple is valid in query processing based on the VID. The tuple is the latest when the VID of the tuple is the same as that of the modified document in the version list. Moreover, the tuple is also the latest when the DID of the modified document is not contained in the version list. In contrast, the tuple is invalid when the VID of the tuple is smaller than that of the modified document in the version list. We give a specific example of the validation check in Figure 6. The DID of the first tuple in the index is 103, and the version list does not contain that DID. Thus, the first line is valid. The document of the second tuple has been deleted, because the DID of this tuple is contained in the version list and its VID is $\emptyset$. The third tuple is the latest, because the VID of this tuple is the same as the VID corresponding in the version list to the Document 101. Similarly, the fourth tuple is not the latest, because the VID of this tuple is less than that of the VID corresponding to the Document 101.

Old versions of tuples in the indices are removed when the load average is low. In this regard, the VID of the latest version of a tuple is rewritten as 0, and the DID of the deleted document is removed from the version list.

4.4 Preliminary Experiments on the Simple Extension System

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

Table 1: Accuracies of term weighting schemes

	BM25E	QLMEL (ω)			
		0.6	0.7	0.8	0.9
iP[.01]	.540	.500	.502	.514	.490

4.4.1 Test Collection and Implementation Settings

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. Approximately 660,000 articles are in this corpus. The INEX topics include 68 queries, of which 32 are CO queries and 36 are CAS queries. We used all of these in the experiments. The INEX relevance assessments are the evaluations of 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, the interpolated precision at the recall level of 1% (iP[.01]) is used as a formal measure of accuracy. The evaluation tool also measures the mean average interpolated precision (MAiP) as the average precision at 101 recall levels.

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

We conducted a preliminary experiment to choose a term-weighting scheme used in later experiments. We examined the effectiveness of BM25E and QLMEL to ascertain which term-weighting scheme is more accurate one.

Table 1 indicates that BM25E is more effective term-weighting scheme. Hence, we used this in the later experiments.

4.4.2 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 the update documents are the same. For this purpose, we randomly sampled documents in order to distinguish between them. In Section 6.4, we consider a more complex case in which the statistics of the documents change dynamically.

All documents are processed through the stop-word 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; then, the update documents are obtained for updating indices incrementally. All data in the `GW-Path-term` and the `GW-Path` indices are scanned in the main memory during updates. Then, the update documents are parsed and the `Term`, the `Tag-term`, and the `RS` indices are updated incrementally.

4.4.3 Effects of Incremental Updates

We investigated search accuracies, update efficiency per document, and total time of index construction by changing the percentage of initial documents within the document set, as indicated in Table 2. 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 2 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 incremental update practical, we need to solve the problem of inaccurate global weights.

The average time for incremental updates is 101.6 ms per document when the ratio of initial documents is 50%, whereas the time required to construct indices from scratch (*no-update*) is 56.0 ms per document. This suggests that the update efficiency decreases as the ratio of initial documents increase. As a result, indexing may take a long time when we update a number of documents.

5. ACCURATE TERM WEIGHTING AND FAST UPDATING OF INDICES

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

- the system retains search accuracy, and
- the system attains fast updating of indices.

Concerning the first requirement, global weights cannot be calculated accurately without sufficient documents because they originally reflect the entire document set. To solve this problem, the simplest approach is to update the global weights as new documents are inserted. Although the idea of updating global weights is reasonable, we cannot always have enough newly inserted documents to calculate accurate global weights. We therefore consider how to calculate accurate global weights even with a limited number of documents, and we propose a solution in the next section.

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

5.1 Accurate Approximation of Global Weights

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

To accomplish this, we utilize the method proposed in our previous study [7] for integrating path expressions. This integration method calculates an accurate global weight for a path expression of few frequencies. The current case is similar to that in the previous study. In both cases, the global weights of elements with rare path expressions are not calculated accurately. Therefore, the integration method should improve the 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 appearance frequencies of tags. As a result of the integration, we eliminate classes that do not contain enough elements to calculate accurate global weights.

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

Table 2: The results of the simple approach

ratio of the initial documents (%)	iP[.01]	MAiP	update time (ms/doc)	total time of index construction (h)
10	.411	.130	60.8	11.1
30	.471	.139	76.5	11.9
50	.480	.135	101.6	12.3
70	.497	.140	107.2	13.1
90	.508	.144	205.4	13.3
no-update	.540	.143	(56.0)	10.3

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

Figure 7: Examples of path expressions

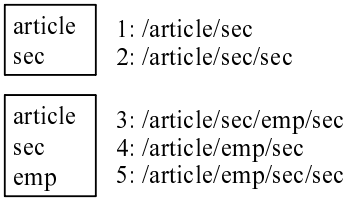

Figure 8: An example of classification in ST

In addition, the cost to adopt these methods is small, because these approaches simply calculate a frequencies and check the order of tags in a path expression. We can ignore the harmful effects on update efficiency. We now explain three 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 separated into two groups. One represents structural classifications such as `article` and `sec` tags. The other indicates semantics, ideas, attributes, and specific contents such as `person`, `emp`, and `table` tags. These two groups of tags are supposedly 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 the appearance order and frequencies of tags in path expressions to integrate similar path expressions.

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

Classification of the path expressions in Figure 7 is shown in Figure 8. The first two path expressions are in the same class because they are both composed of `article` and `sec` tags, while the other three path expressions are in the same class because they are all composed of `article`, `sec`, and `emp` tags. The global weights of the elements with the first two path expressions are calculated together, and the global weights of the elements with the other path expressions are calculated together.

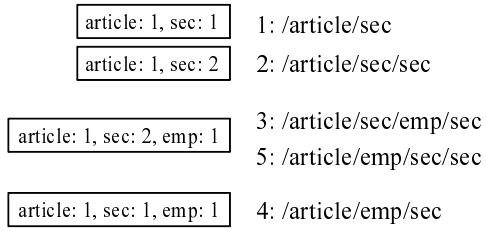

Figure 9: An example of classification in BT

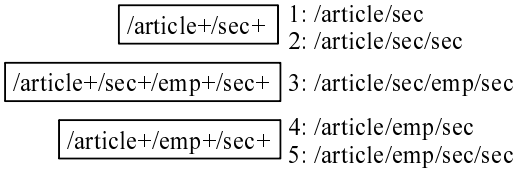

Figure 10: An example of classification in OT

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 perspective of the bag-of-words concept.

Classification of the path expressions in Figure 7 is shown in Figure 9. We first enumerate the names and frequencies of tags in each path expression to integrate the path expressions classified as members of the same class. As a result, we integrate the third and fifth path expressions 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 the appearance frequencies 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 a tag appearing consecutively are different, we suppose that the features of a path expression are not much different because the semantics of each tag are fixed. Therefore, if consecutive tags are the same, such tags can be aggregated.

Classification of the path expressions in Figure 7 is shown in Figure 10. Note that `sec` tags appear consecutively in the second and fifth path expressions. The first and second path expressions are integrated, because these have have one or more `article` tags followed by one or more `sec` tags. The fourth and fifth path expressions are also integrated, these 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

We propose two kinds of filters for selecting important elements and terms to index.

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

We propose an *element filter* to remove unnecessary elements. We previously proposed a method to remove elements that cannot be the search results of any query [3] and a method to identify the most appropriate granularity for search results [8]. Those studies led to the fact that moderate granules are the most appropriate as search results, because extremely large granules (e.g. whole documents) tend to contain irrelevant descriptions and extremely small granules cannot satisfy the information need by themselves. Hereinafter, we attempt to remove extremely small elements, since identifying these is easier.

It is essential to define what extremely small elements are. Many of the Web documents include table-of-contents or reference information, which basically consists not of sentences but of keywords. These descriptions cannot satisfy an information need directly, although they can serve as navigational information. Since one requirement for text summarization is that "information should be self-contained" [13], we remove any element that cannot be understood by itself.

Based on the discussion above, we define three conditions of extremely small elements as follows:

(1) elements containing few terms (threshold τ_{el}),

(2) elements with deep path expressions (threshold τ_{depth}), and

(3) elements with rare path expressions (threshold τ_{Zipf}).

Regarding the first condition, the terms in the information other than the body text including table-of-contents and reference information, contain few terms also in the elements. Actually, study [3] reports that search accuracy improves when short elements are removed.

In the second condition, elements with deep path expressions are eliminated. Tables or lists in HTML have a tendency to be nested deeply. The value of each cell is nonsense without further information, which is the reason that we regard these elements as irrelevant.

Regarding the third condition, path expressions that rarely appear in the document set cannot be calculated accurately, as discussed above in Section 5.1. We therefore use Zipf's law [15] to obtain the threshold of median frequency f, which is computed as follows:

$$f = \frac{\sqrt{8F_1 + 1} - 1}{2} \tag{7}$$

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

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

Figure 11 illustrates the behavior of the element filter. Suppose that four elements, e_1, e_2, e_3, and e_4, are extracted from inserted documents. Elements e_1, e_2, and e_4 are eliminated by the element filter because e_1 is too short, the path expression of e_2 is too deep, and the path expression of e_4 rarely appears. As a result, only e_3 is chosen as a target.

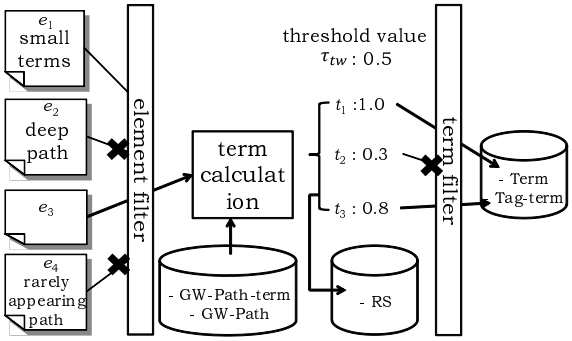

Figure 11: The element filter and the term filter

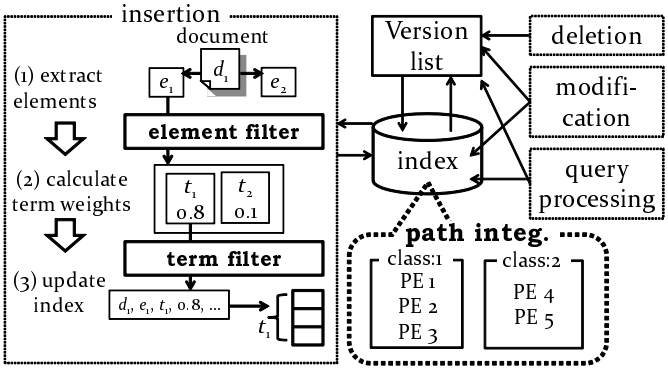

Figure 12: Architecture of the proposed system

5.2.2 Term Filter

Although there are many candidate search results, only a few elements are presented as search results. Therefore, we suppose that search accuracy is not significantly affected if indices do not contain terms with low weights.

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

Note that we apply the term filter only to the `Term` and `Tag-term` indices to enable accurate calculation of the score for elements with the `RS` index. In addition, we do not apply the filter when the number of tuples of the pair of tag and term is less than n.

Figure 11 shows an example of how the term filter works. Suppose that τ_{tw} is 0.5 and there are three terms to insert into the `Tag-term`, the `Term`, and the `RS` indices. We use the single value of τ_{tw} for simplicity although τ_{tw} differs for each pair of tag and term. Terms t_1 ($1.0 > \tau_{tw}$) and t_3 ($0.8 > \tau_{tw}$) are successfully indexed with the `Term`, the Tag-term, and the `RS` indices because they are greater than τ_{tw}. In contrast, term t_2 ($0.3 < \tau_{tw}$) is only indexed with the `RS` index because it is less than τ_{tw}.

5.3 Architecture of the Proposed System

Figure 12 shows the architecture of the proposed system. The main differences between the simple extension system and the proposed system are that latter integrates path expressions to calculate accurate global weights and utilizes two filters (i.e. the element filter and the term filter) to reduce the update cost. The `Term-filter` index contains the thresholds for the term filter.

The query processing part is the same as in the simple approach.

Table 3: Accuracies with changing τ_{el}

τ_{el}	25	30	35	40	45	50	55
iP[.01]	.526	.530	.540	.527	.526	.532	.527

Table 4: 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

When documents are inserted, the global weights in the `GW-Path-term` and the `GW-Path` indices are updated after these are re-calculated using integrated path expressions.

When updating documents, the proposed system treats only the elements and the terms selected by the two filters, unlike the simple extension system.

6. EXPERIMENTAL EVALUATIONS

6.1 Experimental Design

With the simple extension system as the baseline, we investigate whether integrating path expressions and the applying two filters is effective for searching accurately and efficient for updating the indices.

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

Our proposed approaches admit some variations. There are four ways of calculating global weights: the default method, which is classification based on path expression; 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: the element length threshold τ_{el}, the path depth 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 the average update time per document, the index size, and the search accuracy for each variation of the proposed methods. We used the document set with static statistics and chose 50% as the ratio of initial documents. Finally, we confirmed the effectiveness of the proposed methods by using the document set with dynamic statistics.

6.2 Preliminary Experiments for the Element Filter and Term Filter

The element filter eliminates the elements that have extremely short elements, extremely deep path expressions, and rarely appearing path expressions. In this section, we describe some experiments that we conducted to decide the thresholds.

According to the results listed in Table 3, we set τ_{el} to 35; namely, in terms of search accuracy, the best value for the element-length threshold is 35.

Table 4 shows the proportion of elements whose depth of path expressions is less than or equal to τ_{depth}. We measured the proportion for all elements in the test collection and for only those highly ranked elements obtained in our previous study [8]. There is a difference between the result for all elements and that for highly ranked elements. This indicates that we can extract purely useful elements if the depth threshold is set to extract as many highly ranked elements as possible and to discard useless elements. We set τ_{depth} to 6, which ignores any element whose depth is six or more.

Table 5: Effects of the term filter with changing n

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

Table 6: Effects of the proposed approaches

run ID	update time (ms/doc)	disk size(GB)	iP[.01]	MAiP
no-update	(56.0)	158	.540	.143
baseline	101.6	158	.480	.123
ST	102.3	160	.525	.120
BT	100.4	158	.489	.115
OT	100.4	159	.492	.115
τ_{el}	80.1	130	.518	.122
τ_{depth}	95.1	147	.490	.132
τ_{Zipf}	92.6	156	.490	.133
elem filter	75.9	124	.506	.123
term filter	84.9	142	.491	.127
two filters	69.1	114	.490	.126
ST_filters	69.5	116	.499	.125

We also investigated the threshold of Zipf's law. We computed median frequency of the path expressions by using Eq. 7. We set τ_{Zipf} to 166, which ignores any element whose path expression appears 166 or fewer times in the initial index.

In analogy with the element filter, the term filter eliminates terms whose weights are below the threshold. We conducted an experiment to decide the threshold for the term filter, as shown in Table 5. We set n to 10000 or τ_{tw}, which ignores the terms whose weights are less than the 10,000th largest weight of each pair of tag and term.

6.3 Evaluations of the Document Set with Static Statistics

We measured the average update time per document, the size of indices, and the search accuracy with each variation of the proposed methods, as indicated in Table 6. Note that in the case of *no-update*, or constructing a new index from scratch, the average update time replaces the construction time of the initial indices. Note that elements whose length is less than τ_{el} are removed from all results, even those of *no-update*.

Compared with the iP[.01] of the simple baseline system, those of ST, BT, and OT are improved. In particular, ST is the most effective method for calculating accurate global weights and is 9% more accurate than the baseline. In addition, the update efficiencies of these methods are almost equal.

All components of the element filter (i.e. τ_{el}, τ_{depth}, and τ_{Zipf}) save update cost without reducing search accuracy. The combination of τ_{el}, τ_{depth}, and τ_{Zipf} is the most effective of all possible combinations and yields 25% faster updates than the baseline approach. We used this setting for the element filter in the subsequent experiments. The term filter also reduces the update cost by 16% without sacrificing search accuracy compared with the baseline approach.

Next, we evaluated the combination of the two filters. This approach performs better than either of single filters in terms of both update efficiency and search accuracy. The update efficiency is improved by 32%.

Table 7: 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

The former experiments showed that the search accuracy improved with the path expression integrating method and the update efficiency improved with two filters. Then, we combined ST and the two filters as ST_filters. The search accuracy improved by 4% compared with the baseline, while the update efficiency improved by 32%.

In terms of query efficiency, each method takes 1.5 s to 2.0 s per query. This should be acceptable for users. Finally, we can attain fast incremental updates of indices with an effective and efficient search.

6.4 Evaluations of the Document Set with Dynamic Statistics

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

In this set, the initial documents do not include a certain topic but the updated documents do include the topic. We outline the steps to evaluate as follows: (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 whether a document belongs to a certain topic. Wikipedia has many categories of various sizes: twelve major categories are listed in Table 7. We separated 68 queries into the twelve categories. Each query contains from one to five query keywords, and we obtained a keyword set for each category. Since the categories "Technology and applied sciences" (*technology* for short) and "Culture and the arts" (*culture* for short) include relatively large numbers of queries (category queries, or CQs) and query keywords (category keywords, or CW), we used these categories in the evaluation. We assigned a document to a certain category if the document contains the category keywords. Note that these category keywords 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 focus on the effects of term distributions with dynamically changing statistics. In this situation, we assumed that users expect an effective search to be available as soon as new topics are added to the collection.

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

Table 8 lists the iP[.01] of each category for the baseline and ST_filters. For both categories, the proposed methods attained better search accuracies than the baseline. In particular, ST_filters increased the search accuracies rapidly even when the number of update documents was small.

7. CONCLUSION

In this paper, we proposed methods for fast incremental updates of 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 reduce update time by 32% while the search accuracy improved by 4% compared with the simple extension system for static statistics.

8. ACKNOWLEDGMENT

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] Shlomo Geva, Jaap Kamps, and Andrew Trotman. *Advances in Focused Retrieval*. Springer Berlin, 2009.

[3] Kenji Hatano, Hiroko Kinutani, Toshiyuki Amagasa, Yasuhiro Mori, Masatoshi Yoshikawa, and Shunsuke Uemura. Analyzing the Properties of XML Fragments Decomposed from the INEX Document Collection . In *Advances in XML Information Retrieval*, volume 3493 of *LNCS*, pages 168–182. Springer Berlin, 2005.

[4] Fang Huang, Stuart Watt, David Harper, and Malcolm Clark. Compact Representations in XML Retrieval. In *Formal Proc. of INEX 2006 Workshop*, volume 5631 of *LNCS*, 2007.

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

[6] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys (CSUR)*, 40:1–58, 2008.

Table 8: Effects on emerging a new topic

# of indexed doc. ($\times 10^4$ doc.)	*technology* (iP[.01])	
	baseline	ST_filters
37 (25% updated)	.356	.365
47 (50% updated)	.363	.392
56 (75% updated)	.338	.409
66 (100% updated)	.345	.443

# of indexed doc. ($\times 10^4$ doc.)	*culture* (iP[.01])	
	baseline	ST_filters
31 (25% updated)	.456	.517
43 (50% updated)	.506	.560
54 (75% updated)	.501	.585
66 (100% updated)	.496	.587

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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