# Approximative Filtering of XML Documents in a Publish/Subscribe System

**Annika Hinze**[1] **Yann Michel**[2] **Torsten Schlieder**

[1]University of Waikato, New Zealand
[2]Freie Universitaet Berlin, Germany
a.hinze@cs.waikato.ac.nz
ymichel@inf.fu-berlin.de
torsten.schlieder@gmx.net

**Abstract**

Publish/subscribe systems filter published documents and inform their subscribers about documents matching their interests. Recent systems have focussed on documents or messages sent in XML format. Subscribers have to be familiar with the underlying XML format to create meaningful subscriptions. A service might support several providers with slightly differing formats, e.g., several publishers of books. This makes the definition of a successful subscription almost impossible. This paper proposes the use of an approximative language for subscriptions. We introduce the design of our ApproXFilter algorithm for approximative filtering in a publish/subscribe system. We present the results of our performance analysis of a prototypical implementation.

## 1 Introduction

The recent years have seen a new generation of applications based on the principle of *publish/subscribe* (pub/sub): distribution of stock quotes, news articles, or library alerts. A publish/subscribe system is a (distributed) middleware implementing the event-based communication paradigm: A source or *publisher* publishes *event messages* that announce the occurrence of events, i.e., the occurrence of something of interest within the system. Examples are the publication of a new book or CD. *Subscribers* can subscribe to events that are of interest to them; these subscriptions are called *profiles*. The system filters the incoming messages according to the profiles and forwards matched messages to their subscribers.

Publish/susbscribe systems have their origin in alerting services for digital libraries (Salton 1968). In the first generation of alerting systems, event messages contained the full text of documents, such as a newly published scientific paper (e.g., in SIFT (Yan & García-Molina 1995)). A profile would equal a simple Information Retrieval (IR) query using keywords. Note that the concept of filtering documents against a set of profiles has been explored earlier in *information filtering* by the Information Retrieval community. However, the focus there is on information quality, whereas we are looking at efficiency for large scale settings with high numbers of profiles. The focus of publish/subscribe systems lies more on the efficient filtering of structured data sets. Thus, earlier pblish/subscribe systems supported either attribute-value pairs (e.g., in Siena (Carzaniga 1998)) or SQL-like queries (e.g., in CQ (Liu, Pu & Tang 1999)).

Recently, XML-based messages or documents have been used to encode the event messages (e.g., in NiagaraCQ (Chen, DeWitt, Tian & Wang 2000), XFilter (Altinel & Franklin 2000)). Applications are eBusinesses such as online catalogs or digital libraries. Here, a profile is a XML query expressed in XML-QL (NiagaraCQ) or Xpath (XFilter); the definition of which is a rather demanding task for a user who is not familiar with XML query languages. In addition, almost all existing systems assume that the users are well informed about the structure of the event messages and that they are therefore able to create meaningful profiles.

The task of creating a meaningful profile is even more demanding if the system supports different providers of information, e.g., different publishers of music CDs or books, which may use slightly differing catalogue structures. Currently, no system supports filtering over varying structures. In addition, current filter mechanisms detect only documents that contain the exact values a subscription specifies, but it is not possible to detect documents that contain synonymous values.

Typical solutions for this kind of searches in digital libraries are extensions or replacements of search terms with synonyms using a thesaurus or a dictionary (e.g., in the DejaVu system (Gordon & Domeshek 1998)). Other techniques that have been used to explore semantic relationships between terms include user feedback and enriched search interfaces (Rao, Pedersen, Hearst, Mackinlay, Card, Masinter, Halvorsen & Robertson 1995). For substructured data, the problem of approximative results has been extensively addressed for XML search queries (e.g., in (Schlieder 2003, Theobald & Weikum 2002)).

For publish/subscribe systems, the problem of how to extend the profiles and how to efficiently filter using approximations remains open. Note that the issue of how to create thesauri or cost-enriched term lists remains the same problem as for searching. We see this as a separate problem that is not addressed. In this paper, we focus on an efficient filter algorithm for approximate publish/subscribe. We show later that also for algorithms, inspiration may be found in IR solutions, but it is not possible to simply copy these algorithms.

In this paper, we propose an approximative filtering algorithm ApproXFilter to address the problem of approximate filtering. The main challenge for filter algorithms in a publish/subscribe context is efficient filtering of large numbers of profiles. We introduce two forms of an approximative algorithms for filtering XML documents: a time optimized version and a space optimized version. We present a performance analysis of our prototypical implementation and show the usefulness of our approach.

This paper is structured as follows: Section 2 introduces an example scenario that is used to illustrate the concepts throughout the paper and discusses related approaches. In Section 3, we propose the design of an approximative filter and illustrate the design by example. Two implementation variants are introduced in Section 4. We give details about our prototypical implementation in Section 5. Section 6 presents and discusses the results of our analysis of the algorithms and shows the usefulness of our approach. In Section 7, we discuss complementary approaches. The final section summarizes the contributions and indicates future work.

## 2 Motivation

This section introduces an illustrative example scenario. We show that existing approaches in publish/subscribe systems are not sufficient and discuss related approaches from information retrieval. We explain why the principles of approximative IR cannot be simply copied for filtering.

Assume an online warehouse offers a publish/subscribe mechanism for its books. A user may know in advance that author Smith will publish a book in the near future. But unfortunately, nothing about the final title or other information is known other than it deals with XML.

Publish/subscribe systems supporting keyword subscriptions (e.g., SIFT) would notify about all documents that contain at least one of the values "XML" and "Smith". The user cannot specify that she prefers books with the title "XML" over books containing a chapter title "XML". Similarly, the user cannot prefer the author Smith over the editor Smith. Current systems supporting structured XML queries (e.g., XFilter) would result in the contrary: Only exactly matching documents are considered. The XPath query

$$/catalog/book[title = \text{“}XML\text{”} \, and \, author = \text{“}Smith\text{”}]$$

will neither allow for books with a chapter title "XML" nor books of the category "XML" nor books edited by "Smith", nor other media formats than books (e.g., articles or tutorials) with the appropriate information.

Of course, the user can create a subscription that exactly matches the cases mentioned, but she must know that similar results may exist and how they are represented. Since all results of her expanded query are treated equally, the user still cannot express her preferences. It is important to note that different to a search query, a user of a publish/subscribe system cannot simply reformulate their subscription query until it gives the desired results - false negatives will occur and the subscriber misses information without being aware of it.

For search engines, solution have been proposed to cope with the approximative searches. For example, ApproXQL (Schlieder 2003) is an approximative filter language with corresponding search algorithm. While common query languages will only match on exact values that were requested, ApproXQL also supports the matching on similar values or structures. This is achieved by skipping or rewriting parts of the query using synonyms. ApproXQL supports hierarchical, Boolean-connected query patterns. The interpretation of ApproXQL queries is founded on cost-based query transformations: The total cost of a sequence of transformations measures the similarity between a query and the data and is used to rank the results. All results of an ApproXQL query can be computed in polynomial time with respect to the database size.

Here, we will follow the concept of ApproXQL and re-use the syntax of its language for filtering purposes. Similar to the case of searching, we follow the approach of using cost-based query transformation. For publish/subscribe systems, we have to develop a new filter algorithm; it is not possible to use the underlying approximative search algorithm: The concept of filtering is the reverse to the concept of searching. In searching, a set of documents forms the foundation; they are indexed and the incoming search query is compared to the index keys. In filtering, a set of subscription queries exists; they are indexed and the incoming document is compared to the indexed query keys. Similarly, the concept of ranking does not have an exact equivalent in filtering. For filtering, the documents are sent to the user or not.

## 3 The ApproXFilter Algorithm

This section describes the principle of the ApproXFilter algorithm. We start by describing the concept of the algorithm and then move on to discuss each of its steps.

ApproXFilter supports the matching of similar values or structures in addition to direct matches. This is achieved by profile query transformations using skipping, inserting, or renaming parts of the query using synonyms. Whenever a profile query is rewritten for a certain document, each of these transformations may create costs. We introduce the concept of *costs* to judge the quality of a document regarding a given query. A cost of zero means highest quality, i.e., the document exactly matches the profile query as defined by the subscriber. The greater the costs, the lower the matching quality of a document.

Document filtering may be seen as a comparison of the document tree to the set of profile query trees (which are combined in a single directed acyclic graph (DAG)). The more similar a document tree is to a given profile tree, the better the match. That is, the better the match the lower the costs. If all possible transformations are supported for a query, each document will match. The costs describe the amount of transformation necessary to reach that match (similar to relevance in Information Retrieval). If only selected transformations are allowed, not all documents will match a given profile. The costs can be seen as a (reverse) measure for the similarity between the documents and the matched profiles. For profiles that are not matched using transformations, and for profiles that are matched creating high costs, the similarity between the profiles and the document is low.

We now introduce the overall structure of the algorithm. Subsequently, we illustrate the algorithm by using our example scenario.

**Step 1 - Normalization:** After the definition of the subscriptions, transform all ApproXFilter subscriptions into their conjunctive normal form (i.e., Boolean disjunctions combined by conjunctions)

**Step 2 - Profile Extension:** Extend all subscriptions using the allowed predefined transformations (renaming, skipping, insertion)

**Step 3 - Tree-building:** Build a subscription match DAG containing all extended subscriptions

**Step 4 - Filtering:** For each incoming document: Go sequentially through document; concurrently traverse the match DAG depth-first; whenever moving upwards in the match DAG accumulate the costs

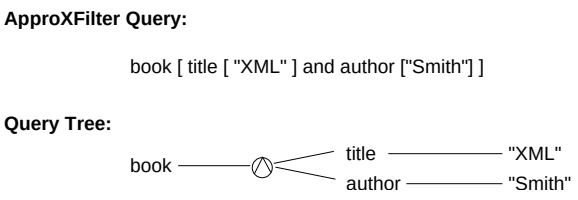


Figure 1: ApproxFilter sample profile query and its query tree (Query 1)

**Step 5 - Notification:** If the accumulated costs for the matching document are less than a predefined threshold, inform the subscriber about the document

We will now illustrate these steps by applying the algorithm to our example scenario introduced in Section 2. We will use two example subscriptions and one incoming XML document to show the principle of the filtering algorithm.

**Normalization** Consider the warehouse's publish/subscribe service from the previous section: Our user is still interested in works about XML by author Smith. Based on her interest, she builds the following subscription written in ApproXFilter:

$$Query\,1 : book[title[''XML''] \\ and\, author[''Smith'']]$$

The subscription query and its query tree representation are shown in Figure 1. Another user is interested in all database books that also consider XML, are published in 2005. He defines the following query:

$$Query\,2 : book[title[''DB''\, and\, ''XML''] \\ and\, year[''2005'']]$$

ApproxFilter's syntax is introduced in detail in Section 4. Note that in this paper, we refer to "XML", "Smith", and "2005" as *values* and to 'book', 'title', 'year' , and 'author' as *structures*; both structures and values are referred to as *terms* in a subscription query. Both subscription queries are already normalized.

**Profile Extension** Using ApproXQL, it is possible to define synonyms or renamings, deletions or skippings, and insertions. For example, the administrators of the warehouse's system may have defined sets of possible transformations for queries regarding print media. In addition, experienced users may define possible transformations. For simplicity, we use very basic transformations as given in Table 1 for profile extensions in our example Query 1.

| Method | Changes | Costs |
|---|---|---|
| Rename | book → article | 4 |
| | title → abstract | 4 |
| | "XML" → "RDF" | 7 |
| Skip | title | 10 |
| | "XML" | 20 |
| Insert | *optional* | 0 |

Table 1: Example profile transformations for Query 1

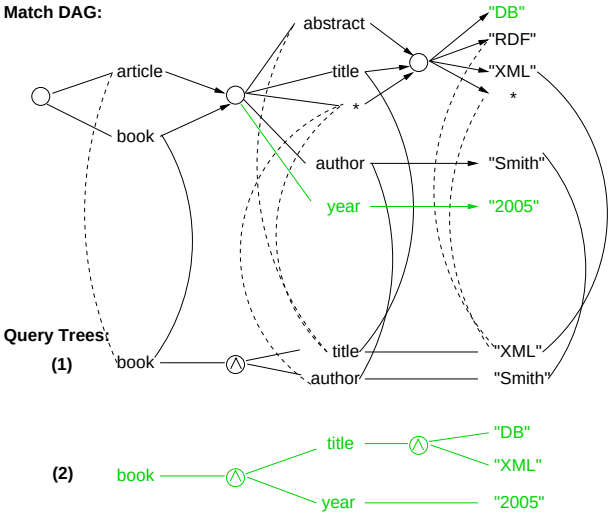


Figure 2: Concept of Match DAG and original query tree. Solid lines for normal edges ($cost = 0$) and dashed lines for additional edges ($cost > 0$)

**Tree-building** The match graph is built as a directed acyclic graph (DAG) for the user profiles as shown in Figure 2. For simplicity, the mapping between the match DAG and the queries is shown only for Query 1; all data regarding Query 2 is shown in a lighter colour. Every term (values and structures) in the extended query is interpreted as a graph vertex.

Figure 2 shows the original profile query (at the bottom) which was extended using the transformations from Table 1: Solid lines between query tree and match DAG represent normal edges, i.e., direct copies from the query tree into the match DAG with no additional costs. Dashed lines represent additional edges, i.e., references created by synonymous structures (e.g., article instead of book) or values ("RDF" instead of "XML") as defined in the transformations table (see Table 1). Additional edges might carry additional costs for the filtering, e.g., as defined as 'Insert' in Table 1. Note that the cost values are chosen arbitrarily. We are aware of the implications of choosing costs, either as a requirement for the user/administrator as well as the challenge of automatic cost assignments. Here, we focus on the performance issues of our approach. In our future work, we plan to address the issues of cost functions and quality.

Note the asterisks in the match DAG in Figure 2: these denote possible skippings of vertices, e.g., the structure 'title' or the value "XML" might be skipped in the filtering. By default, any vertex in the match DAG may be skipped except the root. Skipping vertices may also result in additional costs. The costs for transformations may be defined by system administrators (who should be domain experts) or subscribers.

**Filtering** Event messages passed into the system are assumed to be well-formed XML documents, such as the simple one in Figure 3. Author Smith has named his book "Storing RDF models in Databases", which is a book about XML technology. The word "XML" does not appear in the title. Conventional publish/subscribe systems would not be able to notify about the book. However, ApproXFilter supports approximative matches and can therefore cover this book by using the appropriate synonyms for values and structures.

The filter algorithm parses the document sequentially and traverses the match DAG in depth-first or-

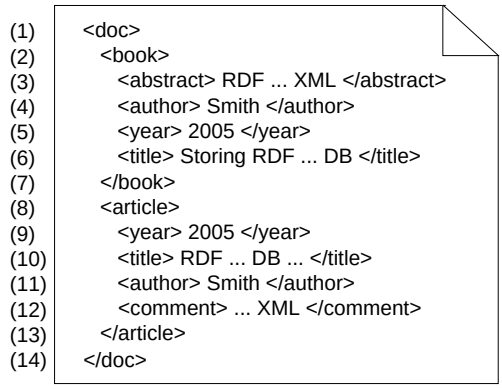


Figure 3: Example document submitted to the publish/subscribe system for filtering

der. Every difference to the original query is scored with additional costs. For simplicity, the costs are not shown in the example DAG but only in Table 1. For each visited node in the match DAG, the corresponding costs are calculated.

The assignment of costs to each filter step and the final computation of the costs is a non-trivial task. Subsequently, we therefore explain the filtering algorithm and its cost assignments in detail using the example document shown in Figure 3 and the two subscriptions defined earlier that have been processed into the match DAG in Figure 2.

The filter starts parsing the example document (see Figure 3) following the XML tree structure. Each found tag is compared to the match DAG (see Figure 2). Recognizing the tag <`book`> in Line 2 it finds the first matching tag in its internal match DAG (introduced to the DAG by Query 1). It also finds the term 'article' in the DAG as possible renaming for 'book' (introduced by Query 2); here we mainly concentrate on the matchings of Query 1. It then finds the tag <`abstract`> and since this is allowed as a renaming of 'title', it follows this route. Note that the renaming costs (4) are not yet added up but noted in the DAG. In the next step, it compares the words "RDF ... XML" of the abstract to the ones specified in the query for title. First, the filter detects a match of "RDF" and notes the additional costs (7) for this level. When continuing comparing the words, the algorithm detects that "XML" matches the same vertex but with no additional cost (0).

A document is successfully parsed if the profile query (using allowed transformation) was completely executed. An unsuccessful document could have, for example, a mismatching root node such as CD instead of book or article in our example. After a document is successfully parsed, its costs are evaluated by ascending the match graph. Whenever two branches meet, i.e., whenever a forest of subgraphs finds a common root, the lowest branch-cost is taken as the cost to be accumulated upwards. Therefore, the algorithm always takes the "best sub-match" to compute the match-quality of the parsed sub-document for a given query.

After finishing the comparison for the abstract and reaching the closing tag <`/abstract`>, the algorithm moves upwards in the DAG, calculating the costs as the sum of insertions, deletions and renamings (i+d+r): On the leaf level (Level 3) it computes the minimum of the costs for "RDF" (0+0+7) and "XML" (0+0+0) as $(min(7;0))$ and decides on the match of "XML". On Level 2 of the DAG, the abstract is now closed and the algorithm moves forward to the next tag in the document.

Next, the two tags <`author`> and <`year`> are processed. They do not add additional costs for Queries 1 or 2 because both terms are matched, respectively. We do not go into detail for these tags but concentrate on the subsequent tag <`title`> in Line 6. The filter algorithm follows the tag <`title`> as requested in the profile and tests the title content. As when filtering the abstract, it computes the costs for the "RDF". The occurrence of "DB" is considered for Query 2, but we will not go into detail for that query. The costs for the leaf level for Query 1 are only the renaming costs for "RDF" $= 0 + 0 + 7$. Moving upwards in the DAG, two branches meet on the next level: 'abstract' and 'title'. Their costs are calculated as the sum of their individual costs and the costs of their children resulting in $0 + 0 + 4 + (0)$ for 'abstract' and in $0 + 0 + 0 + (7)$ for 'title'. The algorithm computes the minimum costs for Level 2 $(min(4;7))$ and decides on the match of 'abstract' (Level 2) followed by "XML" (Level 3). On detecting the close-tag <`/book`>, the overall costs regarding Query 1 for the book structure in the given document are summarized as 4. The costs for Query 2 are also calculated now.

The XML document in this example contains references to two works, i.e., two events are published. This is not necessarily required but it is allowed. The filter algorithm continues parsing the document, now concentrating on the article (starting in Line 8). By detecting the close-tag <`/article`>, the overall costs regarding Query 1 for the article structure in the given document are calculated as $4 + 7 = 11$ (renaming 'book', renaming "XML").

**Notification** A threshold should be defined by the subscriber or a domain expert for limiting the costs that are allowed for results regarding a given profile. Let's assume a threshold of 10 for our example. Documents with costs lower than the threshold are then forwarded to the subscriber of the profile. In our case of Query 1, the reference for the book (cost 4) is selected and the reference for the article (cost 11) is discarded. Consequently, subscriber for Query 1 will receive a notification about Smith's book.

## 4 Technical Design

In this section, we propose two alternative implementations for the ApproXFilter algorithm: a time optimized and a space optimized variant (in Sects. 4.1 and 4.2).

### 4.1 Time-optimized Algorithm

This variant of the algorithm's implementation aims at minimizing the time for filtering a given document. To optimize query evaluation, a permutation of all possible vertex compositions is created (see Figure 4). This includes composition of missing vertices as well as the full query structure as defined by the user's profile. Any vertex may be missing except the root vertex.

Each block of boxes in the figure represents a hash set. For each level in the graph, several hash sets can exist. Each hash set but the root has at least one incoming solid arrow (e.g., book and article point to the middle hash-set). The origins of these arrows are all on the same level, which we refer to as the 'current level'. So, for the middle hash-set, the current level equals the root level. A hash-set directly below the current level contains all combinations of terms that can be found anywhere below the current level in the match DAG (when starting from the points of origins of the solid arrows). For our example, the middle hash-set has its origins in the root node; from Figure 2

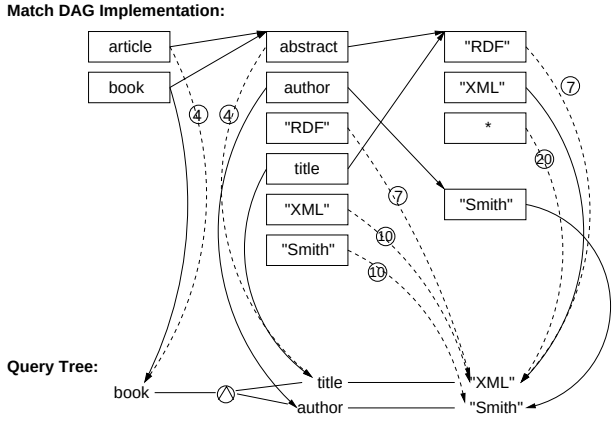


Figure 4: Implementation structure for time-optimized filter algorithm. Solid lines for normal edges ($cost = 0$) and dashed lines for additional edges ($cost > 0$). Dotted lines for skippings of hash sets; Costs greater zero given in circles.

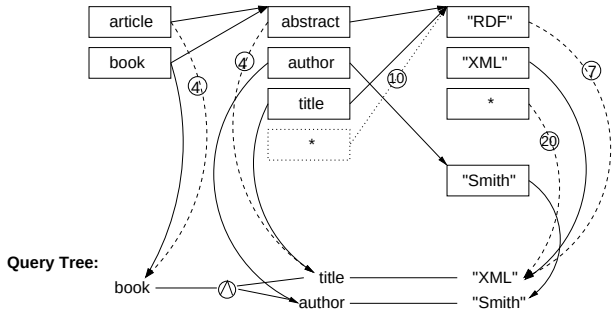


Figure 5: Implementation structure for space-optimized filter algorithm. Solid lines for normal edges ($cost = 0$) and dashed lines for additional edges ($cost > 0$). Dotted lines for skippings of hash sets

we see that below the root node, 6 terms may occur. That means, that the skipping of the title tag has been directly encoded by offering all possibilities of the lower levels also on this level. For this reason, the key "XML" on the middle level carries a cost of 10.

Considering the second level as the current level, below 'abstract' and 'title' in the match DAG, two possible terms can occur ("RDF" or "XML") or the term could be skipped. The skipping has to be made explicit here on the leaf level; the costs for skipping are denoted as 20 as defined.

The dashed arcs in Figure 4 are references to the profile's vertices providing transformation costs whereas the full arcs represent zero costs. Taking our example from above, the arrow pointing from key "RDF" (in the middle hash-set) is annotated with costs for renaming "XML" to "RDF" (7).

Using the structure shown here, the time for evaluating a document is $O(n)$; the space required is $O(n^2)$ where $n$ is the number of vertices in the match DAG as shown in Figure 2. The number of vertices in the DAG could vary considerably depending on the number of profiles $p$ and the number of terms, structures, synonyms. A good estimate would be to assume that $n$ is in the same order of $p$.

### 4.2 Space-Optimized Algorithm

This version of the algorithm aims at optimizing space consumption by using smaller data structures. As in the time optimized version, we use hashes to represent the extended query graph. This time, no redundant node entries are allowed in the structure (see Figure 5). Therefore, each hash key is put into the graph only once and in the exact position for representing the original profile structure. All costs are encoded only once.

To skip nodes, we provide wildcard keys (shown as "*" in the dotted box in Figure 5). These keys must be traversed if no hash value matches (using transitive traversal if necessary). For example, the arrow leaving the lowest key in the middle hash-set (with key "*") and pointing to the right upper hash-set (i.e., the hash set with all possible values in title) is annotated with costs for deleting 'title'. If also the author tag would be allowed for deletion, the arrow would also refer to the hash set with the possible author values.

The filter time for this variant is $O(n^2)$, where $n$ refers to the number of vertices. The space required is $O(n)$.

## 5 Implementation

This section describes the prototype implementation of ApproXFilter. We briefly sketch the prototype's architecture as well as its modules and internal data structures. In addition, we discuss the ApproXFilter language and explain its use for creating a profile.

### 5.1 Components

The prototype of ApproXFilter is written in Java. We use Xerxes[1] for parsing XML documents. There are three main modules in our implementation as shown in Figure 6:

**Profile Service** The profile service receives and parses the user-defined profiles that are incoming via the network. It then creates an internal data structure for storing the incoming profiles. The profile service consists of the *profile server* and the *profile worker*. When a connection to the profile server is established, a new profile worker is started. The profile processing incorporates the following steps: worker initialization, profile parsing, profile extension, profile storage, and worker termination.

The profile worker receives and parses the incoming profiles (see upper part in Figure 6). The profile is added to the profile repository. The profile queries are expressed using ApproXFilter; these are translated into an internal profile representation. The profile server manages the list of allowed transformations and their assigned costs. Out of profiles and transformations, the profile server creates the profile match DAG for filtering the profiles.

**Document Service** The document server receives and parses XML documents; it filters them according to the users' profiles. If profiles match, the profile owners are notified. The central document server dispatches the incoming documents to (distributed) worker threads. The server process is responsible for establishing the connection and passing the work to a dedicated thread.

We regard the matching data structure of profiles as relatively static[2]. Therefore, every document worker obtains a local copy of the global data structure. This copy is only updated when the global profile match DAG changes, i.e, whenever

[1] `http://xml.apache.org/xerces2-j`

[2] This is a viable assumption, e.g., for digital libraries where user profiles describe more long-lived user interests, such as research topics and colleagues.

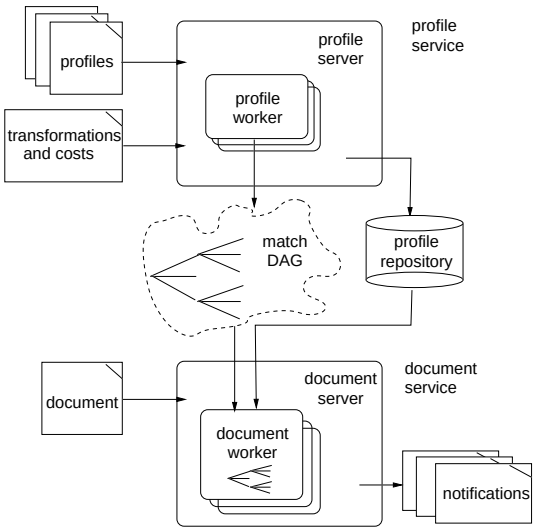


Figure 6: Components of the ApproXFilter engine and their interactions for a set of profiles and a single incoming XML document

the local time-stamp of the match DAG differs from the global time-stamp due to changes by any profile workers. The event processing incorporates the following steps: worker initialization, document parsing, profile evaluation, and worker recycling.

While traversing the incoming XML document, the local data structure is updated with the found vertices and values. The costs of the detected vertices are calculated using local copies of all profiles. To reduce the performance load for updating or initialization, we use time-stamps to detect if the vertex was matched during the current process. If so, we recalculate the costs for this vertex, i.e., we only update the vertex if the new costs are less than the current ones. At last, the complete document costs are calculated by summarizing the costs of all vertices processed in this sequence, i.e., affected by the current document. If a requested vertex was not found in the current sequence, additional costs are added. Additional costs are calculated for insertions as required (i.e., for vertices found in the document that are not mentioned in the profiles). After filtering the document and calculating the costs of the document for all profiles, the costs are compared to the thresholds set for the profiles. Notifications are sent to those subscribers where the document costs are lower than the profile threshold.

**Internal Data Structures** Effective internal data structures are important for efficient filtering. As seen in Figure 6, a number of internal data structures are held: compact profile trees (bottom), a match structure for filtering document structures (left), and a content-synonym set (top). For the structural matches, we implemented a simplified version of the space optimized DAG; see Figure 7. For the value synonyms (e.g., "RDF" instead of "XML") we use an additional content-synonym set. For simplicity, in this proof-of-concept implementation we support stricter filtering than the two versions introduced in Section 4 (i.e., fewer skippings). Consequently, the algorithm is more efficient.

For the match-DAG, we maintain a list of all vertices and their synonyms and a compact profile tree structure (see left and bottom in Figure 7).

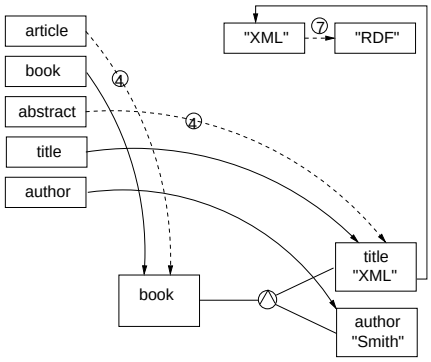


Figure 7: Implemented data structure for matching profile queries; top: value renamings, left: structural renamings, bottom: profile

Each of the vertices refers to all respective profile vertices. For example, 'article' and 'book' both refer to the profile term 'book'. This structure facilitates an efficient document parsing process. In addition, every profile-vertex can automatically detect whether it was filtered via the original path or via a transformed one. The latter case results in additional costs.

The profile vertices (bottom of Figure 7) store the profile-defined values within the same vertex (e.g., 'title' and "XML" together), and not in a separate vertex as initially proposed in Section 4.2. This merging of content vertices with their parent structural vertices prevents false positives. In our example, the profile would otherwise also match documents containing "Smith" in arbitrary vertices and not only in the ones specified directly in profiles and by allowed transformations. Renamings of values are supported by using the additional content-synonym set (shown in the upper part of Figure 7).

The implemented data structure requires less space than the structure for the space-optimized algorithm version (due to more densely stored profiles); and it's performance is between the performance of the space-optimized and the time-optimized version. The performance is $O(m^2 + p)$ and the space requirement is $O(m+p)$ where $m$ is the number of structural vertices in the match DAG (i.e., structures ad their synonyms) and $p$ is the number of value vertices in the match DAG (i.e., values and their synonyms).

| Element | Content |
|---|---|
| query | lexpr |
| expr | lexpr (AND lexpr)* - content (AND content)* |
| lexpr | label LPAREN expr RPAREN |
| label | LNAME |
| content | LITERAL |
| LPAREN | [ |
| RPAREN | ] |
| LNAME | ( 'a'..'z'-'A'..'Z' ) ( 'a'..'z'-'A'..'Z'-'_'-'0'..'9' ) |
| LITERAL | ' " ' ( ~ ' " ')* '"' |

Table 2: ApproXFilter profile definition language

### 5.2 ApproXFilter Language

As already described, we use a subset of the ApproXQL query language for expressing subscriptions in ApproXFilter. Our profile language defines a tree-shaped query string. In our current implementation, we only support conjunctive expressions. The language components used in our implementation are shown in Table 2 as the abstract syntax tree that we used for creating the profile parser using ANTLR [3].

Every profile query consists at least of a labelled expression, "lexpr", having a expression "expr", which is a "content" element. Labels define structural filters, where the label name may consists of any combination of alphanumeric values (see LNAME). "Content" refers to value filters, where a value may be any string enclosed in inverted commas without containing the inverted commas itself (LITERAL).

Translated into our graph profile representation, this describes a single vertex with some content. The query language supports the specification of query strings in which at least one vertex's content-element has to be specified, whereas the parent vertices may be described as simple containers. That is, at least one value filter has to be defined; an arbitrary number of structural filters is allowed.

## 6 Evaluation of ApproXFilter

In this section, we present the results of the evaluation of our implementation of the ApproXFilter algorithm. We performed functional and quantitative analyses, which are discussed the next two sections.

It is beyond the scope of this paper to reason about the quality of the filter results using structural and/or term-based synonyms; this would reach far into a discussion of IR methodologies and criteria. Therefore, we like to refer instead to similar work done for approximative querying on XML: the quality of the results is the same, since only the filter direction is changed (documents on profiles vs queries on documents). For an extensive discussion see (Schlieder 2003).

The quantitative tests have been performed on a local installation. For a distributed approach, we refer to the multitude of literature for routing algorithms for publish/subscribe, which could be applied here, such as the profile and event forwarding strategies proposed in (Carzaniga 1998).

### 6.1 Functional Analysis

The functional analysis evaluated the influence of the use of renamings and skippings on the size of the result set. In the first version of the evaluation, the match-DAG was build using the original profiles as defined by the users. In the second version, the profiles were extended using the mentioned transformations.

We tested with a cost-setting for structural conservation, i.e., structural changes cause higher costs then value changes. The costs for this test were defined as follows: skip structure – 15, skip value – 5, rename – 1, insertions – 0. Note that these values are arbitrarily chosen and variable. The results for the filtering of a selection of 50 test documents (using both test versions) are shown in Figure 8. The document IDs appear on the x-axis; the percentage of matched profiles for each document is shown on the y-axis.

The solid boxes represent the proportion of matched profiles for a certain document without transformations. The patterned boxes show the match benefit due to the use of transformations, i.e., the patterned boxes show the added percentage of matched documents based on transformations. Most documents find more matches after profile transformations. Thus, more users are notified about these documents.[4]

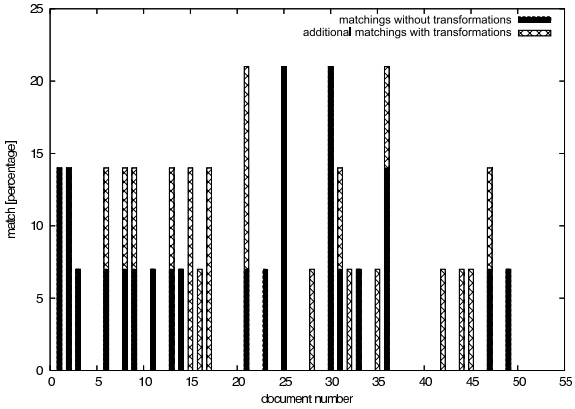


Figure 8: Functional evaluation of the ApproXFilter prototype, matchings with and without transformations

Note that some documents are not matched by any profiles when evaluated strictly, but are matched when approximate matches are allowed (e.g., Documents 15 – 17). These documents originally do not trigger any notifications. On the other hand, for some documents the results are not affected by filter transformations, such as Documents 1 – 3. This means that the similarity between these documents and the profiles was not changed by extending the profiles. Some documents are not matched at all (Documents 37 – 41). For these documents, the similarity between the documents and original profile queries is extremely low, and no similarity is gained by extending the profiles. The algorithms output matched the profile specifications (for details see (Michel & Hinze 2005)). The results of the functional analysis show that the algorithm works as designed: increasing the number of profile matches using approximate filtering.

### 6.2 Quantitative Analysis

The quantitative analysis evaluates the influence of varying profile numbers on the performance and the space requirements of the algorithm. We present here initial results from a series of tests run on ApproXFilter. This information will assist comparison of later implementations of approximative XML filtering engines.

The test setting used here was similar to that in the qualitative test as described above. For every set of profiles tested, 1000 unique documents were created and filtered. Figure 9 shows both the space usage and performance of our implemented prototype. The left hand side of the figure shows a scale for the time and the right hand side a scale for the space. As argued in Section 5, the space requirement directly depends on the number of vertices in the match DAG. For our test setting, that means that it directly depends on the number of profiles.

For each profile set, we show the mean value for the filter time for one document. The maximum and minimum values indicated show the variation between documents. Note that the variations are stronger for small profile sets. This is due to the stronger influence of single terms on the the filter outcome: both

[3] http://www.antlr.org/doc/index.html

[4] The stepwise pattern in the results is due to the selection of documents and not inherent to the algorithm.

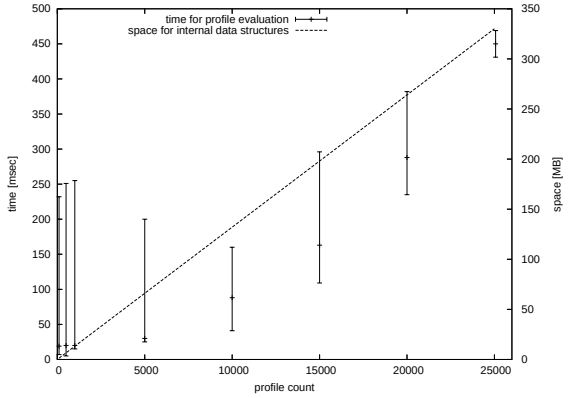


Figure 9: Quantitative evaluation of the ApproXFilter prototype, performance and space requirements depending on number of profiles

documents' structures and profile queries interact to determine the time taken for a filter on one document. Larger samples dampen the effect of this variation.

The performance-related results shown in Figure 9 support our theoretical hypothesis that the algorithm's performance is related to the square of the number of structural vertices.

## 7 Related Work

Research that is directly related to our approach has been discussed in Section 2. In this section, we look at areas of research that are related but complementary to our work. These areas are flexible queries for semi-structured data, information-retrieval extensions for XML query languages, and filter algorithms for XML documents.

The problem of similarity between keyword queries and text documents has been investigated in information retrieval (Baeza-Yates & B.Ribeiro-Neto 1999). We believe, these models cannot be directly applied to XML documents, since they (1) mostly ignore the structure of XML documents and may therefore lower the retrieval precision, and (2) use models based on term distribution that are of little use for data-centric XML documents. For a discussion of these aspects, see (Fuhr, Lalmas, Malik & Szlavik 2005).

As discussed in Section 2, XML query languages incorporate the document structure and are therefore well suited for applications that query and transform XML documents (Bonifati & Ceri 2000). Almost all query languages for XML support regular path expressions, which allow to specify alternative paths through the data graph and to skip certain subgraphs. Although regular path expressions give some additional flexibility, they also require a considerable knowledge about the data. The user must at least know that some subgraphs must be skipped, that alternative paths exist, and how they look like. Consequently, the user needs substantial knowledge of the data structure to formulate queries. XML query languages that support result ranking are XXL (Theobald & Weikum 2002), ELIXIR (Chinenyanga & N.Kushmerick 2002), XIRQL (Fuhr & Großjohann 2000), ApproXQL (Schlieder 2003).

For event notification systems, we distinguish event centered approaches from document-centered approaches. An example for an event-centered system is A-mediAS (Hinze 2003). In document-centered systems, the events are the publication of a new document. Some publish/subscribe systems use XML-encoded the documents, e.g., NiagaraCQ (Chen et al. 2000) and XFilter (Altinel & Franklin 2000). Profiles are expressed using XML query languages such as XML-QL or Xpath. None of these systems supports approximative filtering of XML documents based on similarity measures.

To the best of our knowledge, the only publish/subscribe system addressing approximate matchings is A-ToPSS (Liu & Jacobsen 2002). Its approach is in sharp contrast to our own. A-ToPSS supports approximate matching for attribute-values pairs using probabilistic measures for both documents and profiles. For each attribute, a possibility distribution may be used to express the confidence that the attribute has a given value. This approach is fundamentally different to the one proposed in this paper. We believe it would be of only limited suitability for text-centered structures; the definition for probability distributions for texts is questionable; it would need substantial knowledge and would unnecessarily burden the users. This approach would map particularly poorly onto XML documents, e.g., because structural changes are not supported and the system works on numerical values only.

## 8 Discussion and Future Work

Recent publish/subscribe systems increasingly focus on documents send in XML format; subscribers to these systems have to be familiar with the underlying XML format to create meaningful subscriptions. In this paper, we proposed the use of an approximative language for subscriptions.

We introduced the design our ApproXFilter algorithm for approximative filtering in a publish/subscribe system. We discussed two implementation variations that optimized the space usage and the filter performance, respectively. We implemented a proof of concept ApproXFilter prototype that we subjected to qualitative and quantitative testing. The results of our analyses have shown the effectiveness of our approach. To the best of our knowledge, no other filter algorithm for approximative filtering of XML documents exists.

Having proven the concept of approximative filtering, we have a number of open challenges to address: The definition of cost values is a non-trivial problem. Although there are only five cost-related parameters in our prototype, the adjustments have to be done very carefully. The importance of a missing term depends on the filter application. Using low structure-costs results in a more content-based filtering, while lowering the value-cost parameters will result in a more structural filter. We plan to explore the use of user relevance feedback to adjust the costs. A similar dependence on the application domain exists for the definition of synonyms. For this, we would like to explore the use of domain ontologies and personalised ontologies.

One of the next steps will be an extension of our prototype to also support disjunctions. We plan to further analyse and refine the proposed algorithms. In the future, we would like to explore how ApproXFilter could be used in the context of digital library software (internally using XML document representations). It would also be worthwhile to explore a combination of ApproXQL with the Lucene search engine[5] for querying XML documents with subsequent ongoing filtering queries using the ApproXFilter algorithm.

[5] `http://jakarta.apache.org/lucene/docs/index.html`

## References


Altinel, M. & Franklin, M. (2000), Efficient filtering of XML documents for selective dissemination of information, *in* 'Proceedings of International Conference on Very Large Data Bases (VLDB '00)', Cairo, Egypt.

Baeza-Yates, R. & B.Ribeiro-Neto (1999), *Modern Information Retrieval*, Addison-Wesley.

Bonifati, A. & Ceri, S. (2000), 'Comparative analysis of 5 XML query languages', *SIGMOD Record* **29**(1), 68–79.

Carzaniga, A. (1998), Architectures for an Event Notification Service Scalable to Wide-area Networks, PhD thesis, Politecnico di Milano, Milano, Italy.

Chen, J., DeWitt, D., Tian, F. & Wang, Y. (2000), NiagaraCQ: A scalable continuous query system for internet databases, *in* 'Proceedings of ACM SIGMOD', Dallas, Texas.

Chinenyanga, T. & N.Kushmerick (2002), 'An expressive and efficient language for XML information retrieval', *JASIST* **53**(6), 438–453.

Fuhr, N. & Großjohann, K. (2000), XIRQL: An extension of XQL for Information Retrieval, *in* 'Proceedings of ACM SIGIR Workshop On XML and Information Retrieval', Athens, Greece.

Fuhr, N., Lalmas, M., Malik, S. & Szlavik, Z., eds (2005), *Advances in XML Information Retrieval: Third International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2004, Germany, December 6-8, 2004*, Vol. 3493 of *LNCS*.

Gordon, A. S. & Domeshek, E. A. (1998), Deja Vu: a knowledge-rich interface for retrieval in digital libraries, *in* 'Proceedings of 3rd International Conference on Intelligent User Interfaces (IUI '98)', San Francisco, California, United States.

Hinze, A. (2003), A-MEDIAS: Concept and Design of an Adaptive Integrating Event Notification Service, PhD thesis, Freie Universität Berlin.

Liu, H. & Jacobsen, H.-A. (2002), A-topss - a publish/subscribe system supporting approximate matching, *in* 'Proceedings of International Conference on Very Large Data Bases (VLDB'02)', Hong Kong, China.

Liu, L., Pu, C. & Tang, W. (1999), 'Continual queries for internet scale event-driven information delivery', *IEEE Tranactions on Knowledge and Data Engineering* **11**(4), 610–628. Special issue on Web Technologies.

Michel, Y. & Hinze, A. (2005), ApproxFilter - an Approximative XML-based Filter Engine, Technical Report CS-06/2005, University of Waikato, New Zealand.

Rao, R., Pedersen, J. O., Hearst, M. A., Mackinlay, J. D., Card, S. K., Masinter, L., Halvorsen, P.-K. & Robertson, G. G. (1995), 'Rich interaction in the digital library', *Communications of the ACM* **38**(4), 29–39.

Salton, G. (1968), *Automatic Information Organization and Retrieval*, McGraw-Hill, New York.

Schlieder, T. (2003), Fast Similarity Search in XML Data, PhD thesis, Freie Universität Berlin.

Theobald, A. & Weikum, G. (2002), The index-based XXL search engine for querying XML data with relevance ranking, *in* 'Proceedings of Advances in Database Technology (EDBT '2002)', Prague, Czech Republic.

Yan, T. W. & García-Molina, H. (1995), SIFT - a tool for wide-area information dissemination, *in* 'Proceedings of the USENIX'1995', New Orleans, Louisiana, USA.