ApproXFILTER – an Approximative XML Filter

Yann-Rudolf Michel[1] and Annika Hinze[2]
[1]Dpt. of Computer Science, Freie Universität Berlin, Germany,
ymichel@inf.fu-berlin.de
[1] Dpt. of Computer Science, University of Waikato, Hamilton, New Zealand,
a.hinze@cs.waikato.ac.nz

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. We propose the use of an approximative language for subscriptions. We introduce the design our ApproXFILTER algorithm for approximative filtering in a pub/sub system. We present the results of our analysis of a prototypical implementation.

1 Introduction

Today, event filtering systems are becoming more and more common (see [4]). They inform users on certain events, for example, that a new CD of a certain artist is available. Events differ from domain to domain, but in most cases they are messages being passed to the system from a provider (see Figure 1). Because XML becomes more and more common as a standard data exchange format, we consider only documents, which are XML documents. Users specify so-called profiles which describe the events they are interested in. The event filter notifies the subscribed users, if any provider's document fulfills their profile. This may be realized for example by sending an email.

Until now, research mainly focused on efficient filter algorithms [2,5] or distributed filtering. With current filter mechanisms, it is only possible to detect events that contain the exact values a profile specified but it is not possible to detect those events that

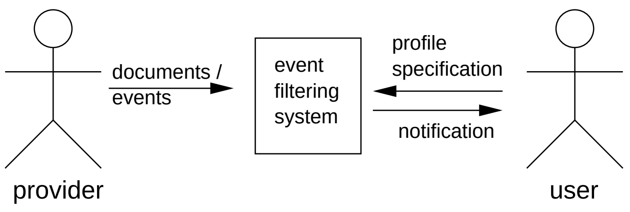


Figure 1: Basic principle of an event filtering system

contain synonym values. In addition it is not possible to detect similar structured documents containing the desired information. As an example, think of someone who is interested in “Rachmaninow”. He or she subscribes to an event notification service, specifying that he/she is interested in new CDs of “Rachmaninow”. The service sends him/her an email when any new CD becomes known to the service in which “Rachmaninow” is the artist. But due to “Rachmaninow” being also a composer, the user would never be informed if there was a new CD with music by “Rachmaninow” that was performed by some other artist. One possibility to solve that problem is to specify several filters but this might be very difficult because the user has to know all possibilities in advance. This might be impossible in some cases. In our example, the query did not describe, what the user wanted to be notified of. This is caused by the linguistic inexactness of our language, which led the user to specifying the query the way they did it.

In this work, we solve this problem by allowing for linguistic variances. That means, the original query is extended with synonym values. In addition, the query is rewritten in parts. This enables our system to detect “content- or structure-similar” documents, which means, these documents will not contain the specified but synonym values or defer in their structure. Existing event notification systems only detect documents where the content matches exactly to the query’s specification. This is why they aren’t capable to solve the problems shown above. To implement a system that is able to act the way our user would expect it to do, we have to change our filtering method. That means, we will change it from an exact and restrictive one to a *fuzzy filter*. This will enable the detection of “content-similar” documents. To define *fuzzy queries*, we use ApproXQL, an approximative query language for XML documents. It was introduced in [7, 8] to query XML documents in an “approximate” way. While common query languages will only match on exact values that were requested, this query language is also capable of matching on differing ones. This is achieved through skipping or rewriting parts of the query using synonyms. These have to be defined in advance. Every deviance from the original query is scored with costs. Therefore it is useful to limit the costs allowed for results regarding a profile. Queries are processed through indexing documents, while event filters index the queries of so called profiles.

This work proposes a concept of an event filtering system using a subset of ApproXQL’s syntax in Section 2. It describes the implementation of ApproXFILTER, an approximative event filtering system for XML documents (Section 3). Results of our analysis and measurements are shown in Section 4. Finally, Section 5 closes this work with a summary and suggestions for future work.

2 Concept of the ApproXFILTER Algorithm

This section describes the principle of the ApproXFILTER algorithm. We additionally introduce two principles for optimizing the internal datastructure.

2.1 Basic principle

As described above, event filtering systems are generally used to limit incoming information for notifying users having predefined their filter criteria in so called profiles. In this section we consider a scenario where an event filtering system is used, maybe at a university. This system informs people about new books, articles or any other print

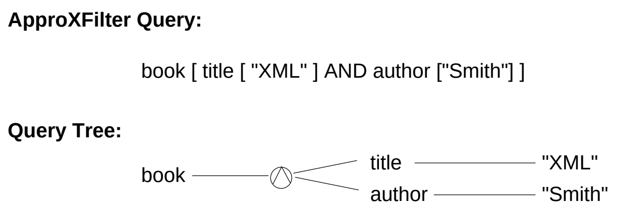


Figure 2: ApproXFILTER sample profile query and its query tree (Query 1)

or online publication. To use this service, a user specifies a profile query to the system. Lets consider a user who is interested in XML topics. They know in advance that “Smith” will publish a book in the near future. But unfortunately, nothing about the final title or other information is known. It is only known that the title must be “something that deals with XML”. Based on this data, we build our sample query which, written in ApproXQL, is shown in Figure 2 as well as the query’s graph representation. We will see more about ApproXFILTER’s syntax in Section 3.2. “Smith” decided to name his book “Database driven RDF newsfeeds”, which is a book about XML technology. However, the word “XML” is not included within the title. An event notification system using a common query language is not able to detect that this publication matches our user’s interests. But due to ApproXFILTER not beeing a query based notification but an event filtering system it is able to serve our users interests. Using ApproXQL, it is possible to define synonyms or renamings in advance. The ones used in our example are shown in Table 1 [1].

original name/value	renaming
book	article
title	abstract
XML	RDF

Table 1: Example renamings

Using predefined renamings, we build the match graph (a DAG - directed acyclic graph) for the user profiles. Taking our example from above we build the graph shown in Figure 3. This graph shows the original query (at the bottom) which was extended with the shown example renamings. Every word in a query is interpreted as a graph vertex.

To evaluate the quality of document regarding a query we introduce “costs”. A costfactor of zero means best quality, i.e., the document matches exactly to what the user defined in the query. The greater the cost-factor the lower the match quality of a document. The costs for a original vertex composition, based on a profile requested word or name, result in zero. In our example, this is illustrated with the full right hand arcs. Additional vertices from synonym-vertices result in additional costs (shown with the dotted left hand arcs). This is because the value/name wasn’t requested by the query definition but extended via a synonym or renaming. Therefore, to restrict bad documents from matching a query, the user is able to specify the maximum cost amount within a profile. (We will see more about the profile’s full structure later.) The

[1]Due to these synonyms vary from domain to domain we discuss the problems of using “the right ones” later in Section 5.1.2.

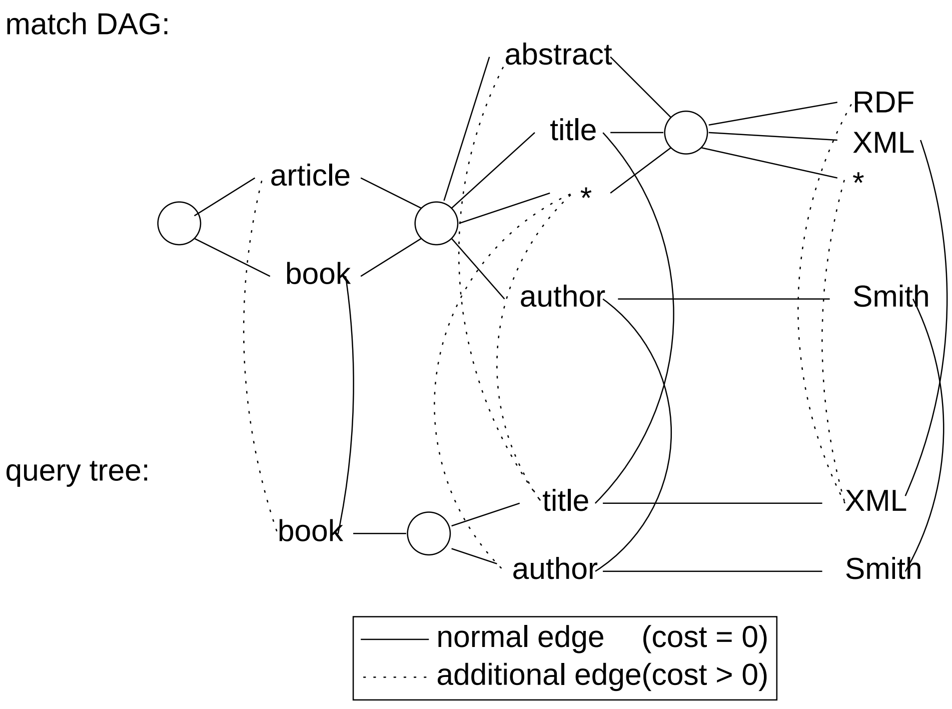


Figure 3: Basic principle of a match DAG and its origin profile/query tree

star-vertices in the graph (visualized with the asterisks “*”), represent that this vertex may be skiped. By default, any vertex may be skiped but the root. Skipping vertices also results in aditional costs

As already described in Section 1, ApproXFILTER is an event filtering system for XML documents. Therefore, events passed to the system are well-formed XML documents. The algorithm now descends through the documents structure while following the match DAG. After a document is parsed, the costs are evaluated by ascending the graph. As the graph is ascended, when two branches meet, the lowest branch cost is taken as the cost to be accumulated upwards. Therefore, the algorithm is always taking the “best match” into account to compute the match-quality of the document for our query. Maybe a document matches the same vertex twice: For example, first, a synonym with additional costs is found, and second, the value itself is found. Of cause, we will add the costs for the exact value (which are zero) to the over all costs. Messages that’s accumulated costs are below a predefined threshold trigger a notification of the user.

2.2 Data structures

To implement this algorithm, i.e., the match DAG, we present two alternatives. One implementation alternative is timoptimal, i.e., the time for evaluating if a document was searched by any query is optimized. The other alternative is to optimize space consumption, i.e., we use smaller data structures.

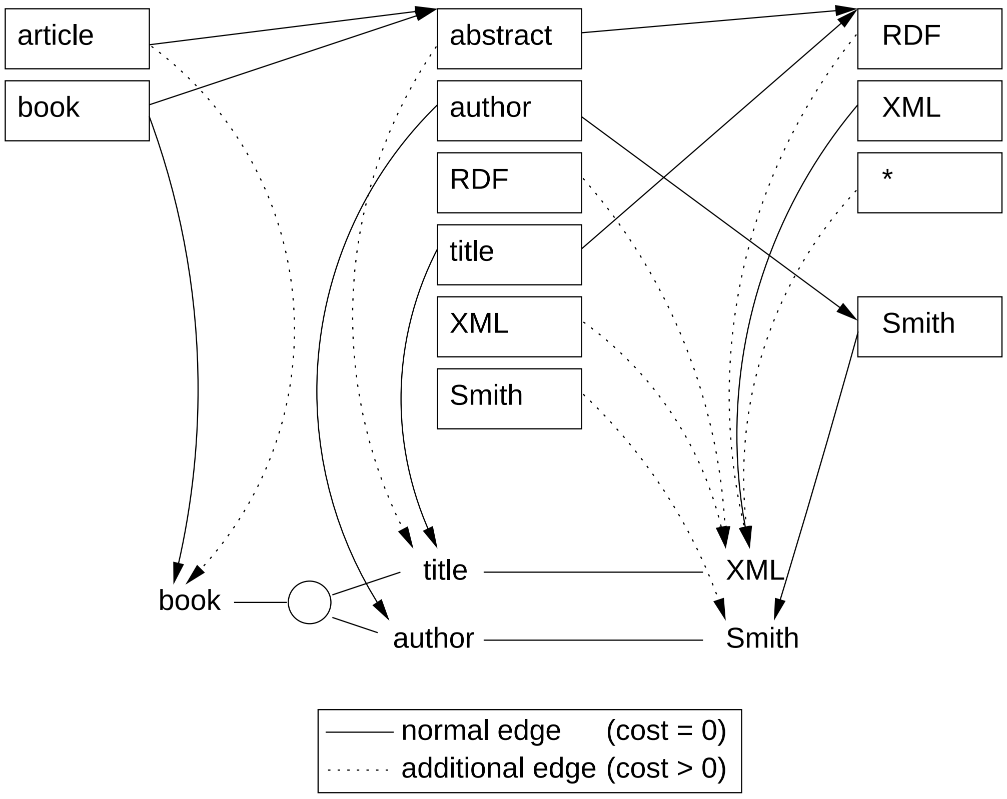


Figure 4: Implementation structure of a time optimized match DAG

Time Optimized Algorithm To optimize query evaluation, a permutation of all possible vertex compositions is created (see Figure 4). This will include the compositions where vertices are missing, rewritings were added and the original query paths specified by users profiles. As we have seen, any vertex may be missing except the root vertex. The only deviance that is allowed for the root is a synonym for it. In this example, it is the synonym 'article' for the root-vertex 'book'. Within the figure, all full arrows are pointers to a hash-set containing all defined key combinations below each level. For example the arrows pointing from 'article' / 'book' to the top of 'abstract' (in the middle) show, that all vertices or contents may appear after them to probably match a users profile. The shown dotted arcs are references to the profile's vertices providing delete- or renaming 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". The time for evaluating a document path is in O(n), but unfortunately, the space required is in O(n^2) where n is the number of vertices.

Space Optimized Algorithm To reduce the space needed to O(n) no redundant entries are allowed (see Figure 5) , i.e., a key is and its synonyms are put into the graph only once and only to the position representing the position of the profile's structure. To delete/skip nodes, we provide wildcard keys (show as asterisks "*"). These keys

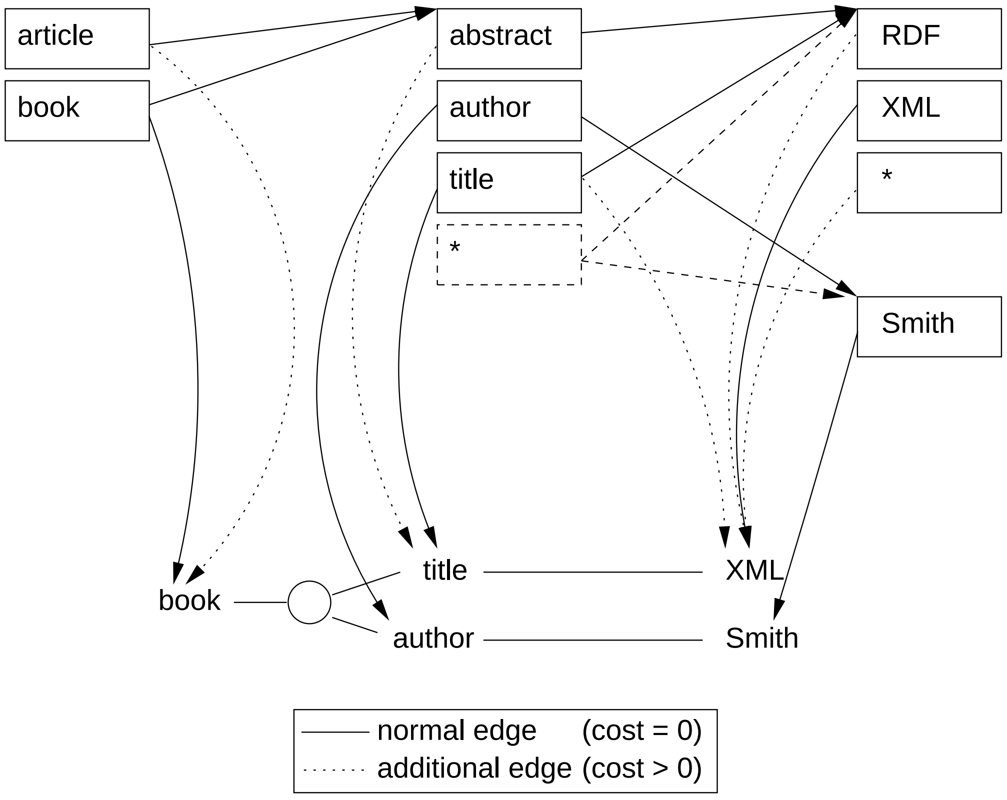


Figure 5: Implementation structure of a space optimized match DAG

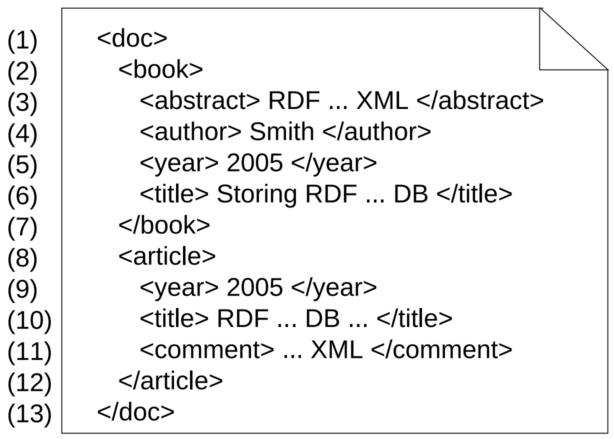
must be transitively traversed if no hash matches and therefore result in O(n^2) time.

Again, any deviance from the profile's definition is scored with costs. For example, the arrow leaving the lowest key in the middle hash-set and pointing to the right upper hash-set is annotated with costs for deleting 'title'.

2.3 Time Optimized Algorithm by Example

We now show the time optimized algorithm in more detail. For this example, we will parse the XML document shown in Figure 6 using the time optimized graph in Figure 4. The filter starts parsing the document by following the XML tree structure. Recognizing the tag 'article' it finds the first matching tag in its internal match DAG. Following the tag 'abstract' and comparing the words "RDF ... XML" of its content to the ones specified in the query (again see Figure 4) first, the filter detects a match of "RDF" and anotates additional costs to the referenced vertex. While continuing comparing the words, the algorithm detects that "XML" matches the same vertex but with no additional costs. Therefore, it updates the costs to zero because the filter algorithm will allways choose the lower costs for the better match detected in a document. But due to 'abstract' matched via synonym (the original query requested 'title') additional costs are added when the close-tag for 'abstract' is parsed.

The filter algorithm continues parsing the document and detects the tag 'author'



```
(1)   <doc>
(2)     <book>
(3)       <abstract> RDF ... XML </abstract>
(4)       <author> Smith </author>
(5)       <year> 2005 </year>
(6)       <title> Storing RDF ... DB </title>
(7)     </book>
(8)     <article>
(9)       <year> 2005 </year>
(10)      <title> RDF ... DB ... </title>
(11)      <comment> ... XML </comment>
(12)    </article>
(13)  </doc>
```

Figure 6: Example document submitted to the publish/subscribe system for filtering

with its content “Smith” that matches exactly our query so this results in zero costs as well. No query requested the existence of ’year’ and no synonym was specified for it, therefore the algorithm will skip the tag. Detecting ’title’ with its content “RDF”, another match for the same vertex is detected. Composing the existing costs (’abstract’ matched via synonym but the content was right) and the resulting costs of matching content (correct vertex but synonym content) the lower costs are taken. The problem on how to size the costs appropriately is discused later in Section 5.1.1. Detecting the close-tag ’article’, additional costs are added because this was a synonym for the requested tag ’book’.

The XML document in this example contains two announces, i.e., two events are published. This is not necessarily required but allowed. The filter algorithm continues parsing the document. Again, ’title’ matches a requested vertex with its content synonym “RDF”. The other tags will not match any of our vertices nor any of our synonyms. By detecting the close-tag ’book’ the costs are evaluated.

At the end of each document, i.e., by closing all opened tags, the costs are summarized. If the costs of a document are below the maximum costs specified in a profile, the document matches the profile and the user will be notified, e.g. via email.

3 Implementation

This section is going to describe how we implemented the ApproXFILTER prototype. We will describe the prototype’s architecture as well as its modules and internal datastructures. We are going to show how events are parsed, i.e., handled and how the profiles are evaluated for user notification. In addition, we will show the used subset of ApproXQL and explain the language’s usage for creating a profile.

3.1 Components

The prototype of ApproXFILTER is written in JAVA. It uses a number of external programs and libraries for parsing [3] XML documents and translating [6] them into an internal representation. As shown in Figure 7, there are three main modules our implementation consists of:

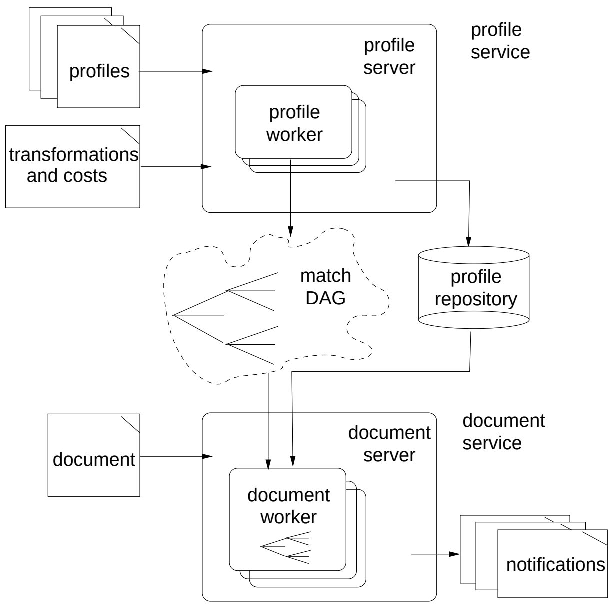


Figure 7: Components of the ApproXFILTER engine and their interactions for a set of profiles and a single incoming XML document

- The *profile service* - receiving user defined notification parameters,
- the *document service* - receiving and parsing events,
- the *internal datastructures* - building a suitable data storage.

These three modules are now described in more detail.

3.1.1 Profile Service

The profile service's function is to receive and to parse the user-defined profiles via network. It also manages the match data-structure of all incoming profiles as well as the the set of profiles.

To make the processing of incoming profiles more scalable. These works are shared among the *ProfileServer*, which manages the match data-structure as well as the profile set, and the *ProfileWorker*, which receives and pareses the incomming profile-documents via network. The complete profile workflow has several steps, that are outlined using the example profile of Figure 2:

Worker Initialization: When a connection to the *ProfileServer* is established, a new *ProfileWorker* is spread, i.e., the server process is only responsible for creating the connection but not for the document reception or parsing.

Profile Reception and Translation: The worker process parses the incoming profile-document. As we assume all documents being well formed XML data, the profiles are subject to be it as well. Therefore a test for "well formed" is implicitly done while the profile is received through the used XML parser [3]. As you can see from Figure 8, in

```
<profile>
   <name> {string}</name>
   <maxCost> {number} </maxCost>
   <queryString> {approXQL-string}</queryString>
</profile>
```

Figure 8: Skeleton of the ApproXFILTER prototype's profiles.

our prototype, a profile consists of a name, a maximum cost-factor and a query string. Later, this may become enhanced by an email address to notify a profile owner via mail or by any other notification parameters (see open topics in Section 5.1.3). Due to the query-string is expressed in ApproXQL, it has to be translated into an internal profile representation. As shown in Figure 2, this is a graph structure.

Profile Submission: The complete profile is then added to the profile set and to the internal match-data-structure. In addition, the profile graph is enhanced by specified synonyms, i.e., the original profile is transformed. Using the ones show in Table 1, this results in a structure shown in Figure 9. In addition, a new time-stamp for the structure is set to document that the data structure has recently changed.

Worker Termination: After the profile processing, the worker is terminated because it is no longer needed. Due to we expect there are only a few profiles incoming but a large amount of events, we terminate it to free memory usage.

3.1.2 Document Service

The documents service's function is to receive documents, i.e., events and to test if any profile's specification matches. If one or more does, it notifies [2] the profile owner(s).

In event notification systems, the number of events, i.e., documents, are clearly more, than the number of incoming profiles. Therefore it is important to have an efficient event processing. In our prototype, there is a central instance, the *DocumentServer*, which task is to dispatch the incoming documents to worker threads. So again, the server process is only responsible for establishing the connection and passing the work to a dedicated thread. As said above, we assume, that in an event filtering system we only have a few new profiles but a large size of incoming documents, i.e., events. Therefore, we regard the match data-structure, based on the incoming profiles, as static which results in every *DocumentClientWorker* having an own local copy of the global data-structure. This copy, i.e., clone is only updated, when the global one changes. To scale, depending on the incoming event rate, it is possible to keep up an existing pool [3] of worker processes. The whole event processing incorporates the following steps:

Worker Initialization: Whenever a new event is passed to ApproXFILTER, a worker process is required. In order to be able to handle several incoming events the same time,

[2]Within our prototype, right now there is not email notification but only a log entry showing that a match occurred.

[3]The amount of worker processes is configurable via a certain config-file. The setting is only recognized during the startup of the filter but not changeable during runtime.

we created a worker pool which can be sized by a configuration parameter. That means, when a document, i.e., an event is passed to our prototype, a free worker is taken out of the pool to process it. As we have said, every *DocumentClientWorker* has its own clone of the match data-structure, e.g. this could be the structure shown in Figure 9. In order to use only the newest version, the local data-structure is updated, whenever its time-stamp defers from the global version updated by any *ProfileWorker*.

Document Parsing: While traversing the incoming XML-document, e.g. the one shown in Figure 6, the local data-structure is updated with the found vertices and values. Having a local copy of all profiles, too, we are able to calculate local costs for each found vertex. To reduce to much updating or initialization, we use time-stamps again, to detect, whether the vertex was filled this time or in any previous sequences of document processing without meanwhile update of the match-structure. Whenever a vertex is filled, it is tested, if we found it before this time but within the same sequence. If so, we recalculate the new 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 adding all vertices costs processed in this seqeuence, i.e., affected by the actual document. If a vertex wasn't found in the current sequence, we add a parameterisable [4] cost value. If we didn't find the requested vertices or values, additional costs are added, too.

Profile Evaluation: After the document was fully parsed, the profiles are evaluated, i.e., they are tested if any profile's max-cost-factor is greater or equal to the calculated document-costs. After that, the profile owners are notified for every profile fulfilling this condition .

Worker Recycling: In the end of any processed document, the worker is recycled, i.e., it is sent back to the worker pool, to parse an other incoming document or to wait for one to process.

3.1.3 Internal Datastructures

As we have already described in Section 2.2, an effective internal datastructure is very important for efficient event processing.

Shown in Figure 7, in ApproXFILTER, we have two internal datastructures. One structure is a set of all profiles being sent to the service. This is used for building the match data-structure and for storing the additional data for user notification and even the maximum allowed costs. The other structure is the match data-structure which enables the *DocumentClientworker*s to efficiently parse and work on incoming documents.

Deliberating about space consumption and path evaluation time, we decided to implement a hybrid structure, i.e., a reduced version of the space optimized DAG. We have a set of all known vertices and their synonyms. Each of it has a list with all profile vertices pointing to it. This principle is shown in Figure 9. This structure facilitates the document parsing process, resulting in fast event evaluation. In addition, every profile-vertex detects wether it was updated via the requested path or via a defering one, resulting in automated costs. Figure 9 shows, that the profile vertices e.g. "title" and "author" include the profile defined value within the same vertex and not in a

[4]We will discuss the problem of cost-factor-sizing later in Section 5.1.1.

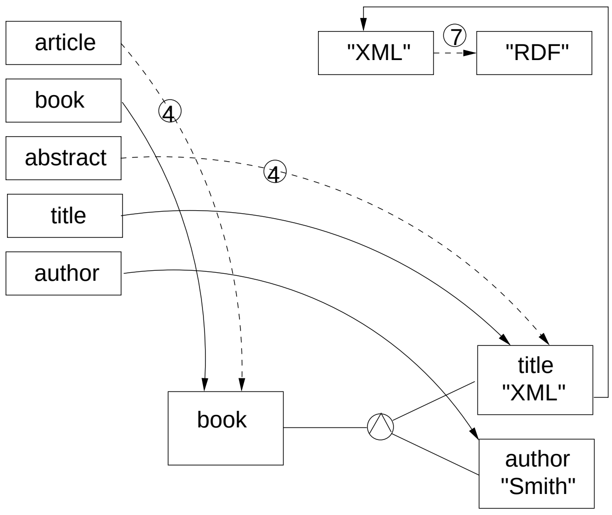


Figure 9: Implemented data structure for matching profile queries; top: value renamings, left: structural renamings, bottom: profile

element	contents
query	lexpr
expr	lexpr (AND lexpr)* — content (AND content)*
lexpr	label LPAREN expr RPAREN
label	LABEL_NAME
content	STRING_LITERAL
LPAREN	’[’
RPAREN	’]’
LABEL_NAME	(’a’..’z’—’A’..’Z’) (’a’..’z’—’A’..’Z’—’_’—’0’..’9’)*
STRING_LITERAL	’”’ (’”’)* ’”’

Figure 10: Grammar of the implemented subset of ApproXQL query language

seperate one like shown in Section 2.2. We decided to merge the “content vertices” with the parent structural ones, because otherwise this would lead to much more “false positives”, i.e., our example profile would also find all Documents containing “Smith” in any vertex and not only in the specified or synonym ones. Nevertheless, we are still able to find content-similar matches, e.g. we find “RDF” instead of “XML”. This is realized by introducing a second content-synonym-set.

3.2 Grammar and Syntax

As already described, we use a subset of ApproXQL’s syntax for specifying the filter criteria of each profile. This language defines a tree fashioned query string. While the original language also enables disjunctive expressions, in out prototype, we only support conjunctive ones right now. The full description of the implemented subset of ApproXQL its components and meanings is shown in Figure 3.2.

Every “query” consists at least of a labeled expression, “lexpr”, having one expression, “expr”, which is a “content” element. Translated into our graph profile representation, this describes a single vertex with some content. A simple, but a bit more complex query string is shown in Figure 2, showing the query string (a) as well as the representative graph structure(b). So our query language specifies a query string, where at least one vertex’s content has to be specified, whereas the parent vertices can only be described as containers.

4 Experimental Analysis

This section deals with the experimental analysis of the implemented prototype shown in Section 3. One approach is to test the quality, i.e., to test the supposed extended match capabilities. A second approach, is to test the quantity capabilities, i.e., to test how the filter scales facing different work loads.

4.1 Qualitative approach

Being able to give a qualitative rating of the prototypes capabilities, we were forced to adopt common, i.e., human readable documents for our events to declare useful profiles and even useful synonyms. In this scenario, we sourced some bib-tex libraries and generated announcements for several published papers. The generated profiles represent subscriptions to a certain point of interest, e.g. one profile subscribed for all papers containing “query” and “optimization”. For our synonyms we used common language synonyms. i.e., we used the synonym “large” for “big” or abbreviated named were fully expanded.

We start our experiments without using synonyms to get a match graph of the normal filter algorithms. Therefore we just leave out our synonyms and pass our sample data to the filter. In the second step, we are adding synonyms to our filter and replay the same documents. The results are shown in Figure 11. 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. 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 (document 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 results of the qualitative analysis prove that the algorithm works as designed: increasing the number of profile matches using approximate filtering.

4.2 Quantitative approach

To test the filters capabilities facing different workloads we generated synthetic test documents and profiles using the ToXgene XML generator [1]. Specifying an extended

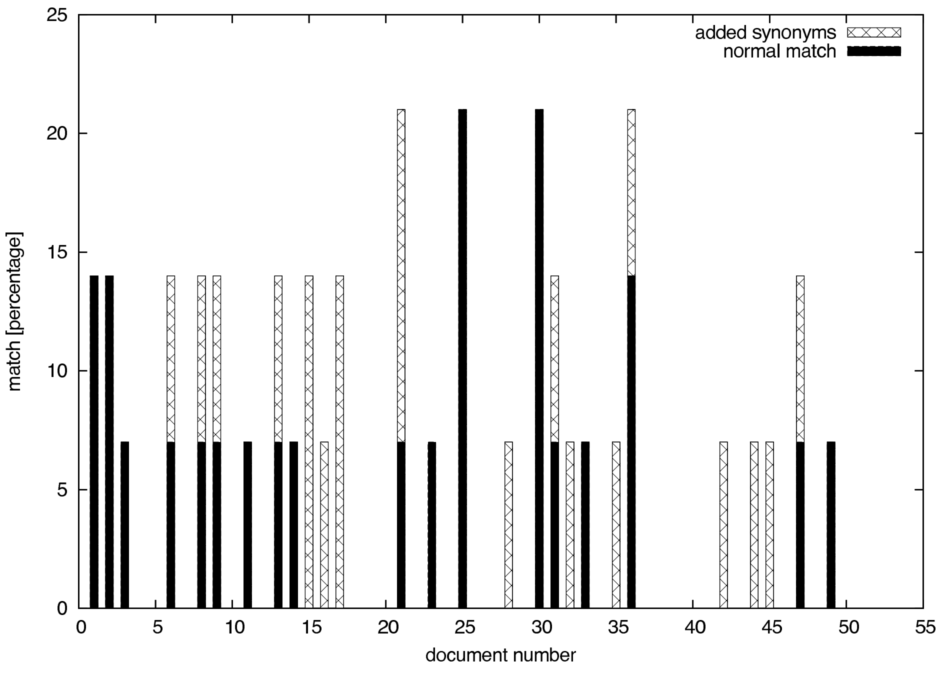


Figure 11: Qualitative analysis showing matched profiles without (solid) and with synonyms (patterned).

XML schema document we generated our documents while still being able to influence the distribution of the contained parameters or profile data.

We generated six test cases with 500, 1000, 5000, 10000, 15000, 20000 and 25000 profiles each processing 1000 documents. Figure 12 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 3, 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 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 Fig. 12 support our theoretical hypothesis that the algorithm's performance is related to the square of the number of structural vertices.

5 Summary and Future Work

We introduced the design of our ApproXFILTER algorithm for approximative filtering in a publish/subcribe 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 quanti-

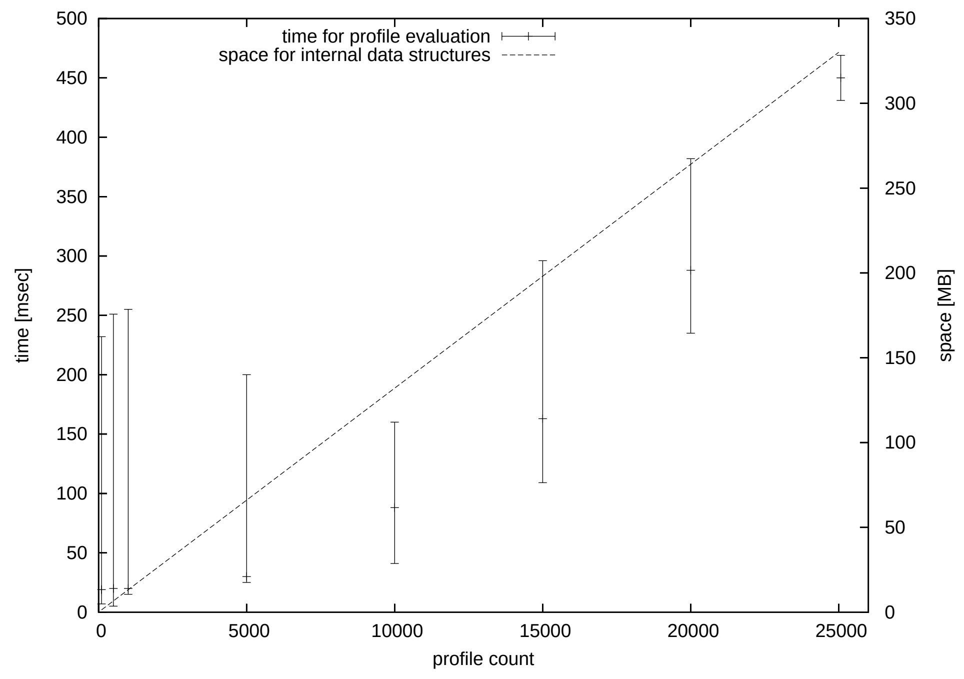


Figure 12: Quantitative analysis showing minimum, maximum and average timing of document processing and the growth of the needed space for the internal datastructures with different numbers of stored profiles

tative testing. The results of our analyses have shown the effectiveness of our approach.

5.1 Open Topics

Having proven the concept of approximative filtering, we have a number of open challenges to address. This section is going to outline them.

5.1.1 Sizing of “Costs”

We discovered the sizing of “costs” to be a non-trivial problem during our experimental analysis. The ApproXFILTER prototype enables several cost parameters to size in advance (see Table 5.1.1). Although there are only five parameters to adjust, this has to be done very carefully. In our experiments, we discovered the shown default values to be a good start but the importance of a missing attribute or a missing content depends on what the filters application domain is. Tuning the attribute’s costs to low would result in a more content based filtering while lowering the content’s cost parameters will result in a more structural filter. Our default values more or less force the documents structure to fulfill the profile’s definition. Therefore the SYNONYM_ATTRIBUTE_COST is sized close to zero. In addition it is very important to get an impression on how the profile’s max costs are defined. That means a user should know the cost parameters in advance to specify the max cost factor of their profile. I.e., knowing these parameters enables the user to decide himself, e.g., how many parameters may be skipped.

Parameter	default value
MISSING_ATTRIBUTE_COST	15
SYNONYM_ATTRIBUTE_COST	1
MISSING_CONTENT_COST	5
SYNONYM_CONTENT_COST	1
WRONG_PATH_COST	5

Table 2: parameters

5.1.2 Linguistic Problems

As described in Section 5.1.1, the cost sizing depends on the application domain of the filter. The same is applicable to the synonym definition. While a synonym *a* in one domain has a similar meaning than the original *b* and maybe vice versa, this may be totally different in an other domain. As we have also seen in our introduction in Section 1 many linguistic problems exist due to our spoken language is not always as precise as we expect it to be.

5.1.3 Extended features

Due to this implementations intention is a proof of concept, we left out some things, that would normally be naturally implemented. The things we left out are:

Protocol Specification: The prototype of ApproXFILTER has no protocol specification for document or profile submission. Right now, it is just a daemon listening on a specified port, receiving all data but without any control procedure or reception messages.

Profile Maintenance: A genuine event notification system would enable the users to maintain their profiles, i.e., to update, or delete them. Today we don't support any of these two functions. Once submitted, a profile stays inside the graph but is not maintainable.

Synonym Submission: Submit synonyms via Network or load them via File on demand. Right now, our ptototype is only able to load the known synonyms once it is started but not during runtime.

Full ApproXQL Syntax Support: Implement the whole ApproXQL syntax, i.e., enable disjunctive expressions as well as the already implemented conjunctions.

5.2 Future Work

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. A similar dependence on the application domain exists for the definition of synonyms. For this, we would like to explore the use of ontologies.

An important item that has to become implemented is the use of a specified protocol to enable controlled client/server interaction and profile maintenance via the appropriate algorithms.

One of the next steps will be an extension of our prototype to support also disjunctions. In the future, we would like to explore how ApproXFILTER could be used in the context of digital library software (internally using XML doument representations), such as Greenstone [9], and in combination with the Lucene search engine [5] for XML documents.

Acknowledgements

The project reported in this technical report has been performed by Yann Michel student project for his Masters degree studies in Computer Science. The project was supervised by Annika Hinze. We like to thank Torsten Schlieder for the valuable discussions and his contributions to this project.

References

[1] Barbosa, Mendelzon, Keenleyside, and Lyons. Toxgene: a template-based data generator for xml. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, Jun 2002.

[2] S. Bittner. Entwurf und analyse eines effizienten verteilten benachrichtigungssystems. Master's thesis, Freie Universität Berlin, 2003.

[3] The Apache Software Foundation. Xerces2 java parser. http://xml.apache.org/xerces2-j/index.html (14.02.2004), 1999.

[4] A. Hinze. *A-MediAS, Concept and Design of an Adaptive Integrating Event Nofification Service*. PhD thesis, Freie Universität Berlin, 2003.

[5] A. Hinze and S. Bittner. Efficient distribution-based event filtering. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, Vienna, Austria, Jul 2002.

[6] T. Parr. Antlr - another tool for language recognition. http://www.antlr.org (14.02.2004), 1989.

[7] T. Schlieder. ApproXQL: Design and implementation of an approximate pattern matching language for XML. Technical Report B 01-02, Freie Universität Berlin, 2001.

[8] T. Schlieder. Schema-driven evaluation of ApproXQL queries. Technical Report B 02-01, Freie Universität Berlin, 2002.

[9] I. H. Witten and D. Bainbridge. *How to Build a Digital Library*. Elsevier Science Inc., 2002.

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