XPath-Aware Chunking of XML-Documents

Wolfgang Lehner
(EMail: wolfgang@lehner.net)

Dresden University of Technology
(Database Technology Group)
Dürerstr. 26, D-01062 Dresden

Florian Irmert
(florian@irmert.de)

University of Erlangen-Nuremberg
(Database Systems)
Martensstr. 3, D-91058 Erlangen

Abstract

Dissemination systems are used to route information received from many publishers individually to multiple subscribers. The core of a dissemination system consists of an efficient filtering engine deciding what part of an incoming message goes to which recipient. Within this paper we are proposing a chunking framework of XML documents to speed up the filtering process for a set of registered subscriptions based on XPath expressions. The problem which will be leveraged by the proposed chunking scheme is based on the observation that the execution time of XPath expressions increases with the size of the underlying XML document. The proposed chunking strategy is based on the idea of sharing XPath prefixes among the query set additionally extended by individually selected nodes to be able to handle XPath-filter expressions. Extensive tests showed substantial performance gains.

1 Motivation

XML has gained the status of a de-facto standard for wrapping (semi-) structured data and exchanging it via the Internet. Even web surfing, i.e. requesting an HTML page from a web server, implies the point-to-point transfer of an XML document as the payload of an HTTP response, if the XML document follows the standardized XHTML schema definition. Reversing the communication pattern of this simple request/response yields the publish/subscribe pattern to build large scale information dissemination systems ([BeCr92], [FoDu92], [AAB+98], [FJL+01], [BaWi01]). In this scenario, data producers (publishers) on the one hand are exposing data to an information broker. On the other hand, users interested in receiving notifications regarding information about specific topics from potentially anonymous data producers are placing a subscription at the broker. As soon as a data fragment enters the brokering component, all registered subscriptions are evaluated against the incoming data. In the end, only those subscribers with a matching subscription are notified by routing the interesting part of the original message to the corresponding subscriber. Obviously the matching component, comparing registered subscriptions to an incoming data fragment reflects the core of an efficient publish/subscribe system. Based on XML documents as messages being exchanging between publishers and subscribers and XPath expressions as a mean to specify subscriptions, we propose a chunking framework to speed up the filtering process and cut down the time needed for matching subscriptions against incoming information.

Building an Efficient Information Dissemination Framework

The database oriented approach to set up an information dissemination system based on the publish/subscribe communication pattern may exploit the ECA model of a database triggering mechanism ([AgCL91], [HCH+99]): on inserting new (and very well structured) information into a database (the event), deliver the information (the action) if the subscription is satisfied (the condition). Unfortunately, the triggering model is primarily designed to perform complex actions (like checking integrity constraints) for a low quantity of triggers. Registering thousands of triggers (one for each subscription) to implement a large scale dissemination system does not sound feasible. Many extensions on a relational and/or semi-structured data model level, for example the concept of a ›Continual Query‹ in OpenCQ ([LiPT99])/NiagaraCQ ([CDTW00]) were made to reduce the pain of triggers. An alternative solution might be to see subscriptions as materialized views inside a database so that subscription evaluation is be reduced to the incremental maintenance of the corresponding subscription views. Exploiting the techniques of materialized views provides a transparent refresh of individual subscriber data and may additionally yield in internal optimizations like multiple query optimization ([LPCZ01]).

The other extreme of evaluating subscriptions is based on checking regular expressions on completely unstructured or semi-structured data (ASCII text, emails,...). Besides many systems in the information retrieval area, SIFT (Stanford Information Filtering Tool; [YaGa95]) was one of the first prominent filtering system published in the database community. SIFT was accompanied by Yeast ([KrRo95]), Siena ([RoWo97]), Gryphon ([ASS+99]), or the DBIS toolkit ([AAB+98]) focussing on different perspectives like routing messages in a distributed brokering environment or defining transformations of messages, etc.

The arrival of XML promised to close the gap between database oriented dissemination systems referring to well-structured information and pure filtering tools operating on any kind of data in evaluating regular expressions. Examples for XML-based dissemination systems can be found in [PFJ+01] (WebFilter), [AlFr00] (XFilter), and many more. Our filtering approach may be implemented on top of an XML-based dissemination system, if the following architectural requirements according to the publish/subscribe paradigm are satisfied (see figure 1):

- *publisher side*
  Before a publisher may expose information as XML documents to the information broker, the publisher has to register by submitting a schema definition. All following documents have to conform to this schema which is held at the broker.
- *subscriber side*
  After inquiring about the registered XML schemas, a subscriber may submit a subscription consisting of a complex XPath expression. XPath expressions of all subscribers are also stored locally at the broker. An incoming data stream is matched with the XPath expressions and a notification consisting of the interesting part of the original XML message is sent to the subscriber.

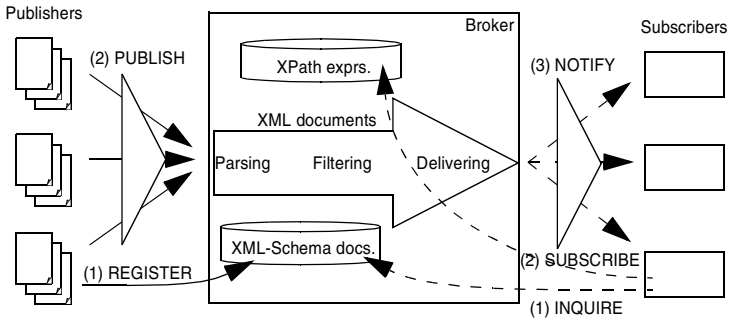


Fig. 1: Subscription Framework

The main idea of the approach discussed in this paper consists in using the schema information of the publishers to optimize the registered XPath expressions and produce a chunking scheme for incoming XML documents so that the filtering process at runtime may operate on multiple chunks instead of one single XML file. In a first step, the following section discusses existing filtering mechanism of XPath expressions, followed by the optimization steps performed during the prepare phase in section 3. Section 4 finally discusses the proposed chunking strategy. Section 5 provides the descriptions of results gained from performing extensive tests based on our ›eXtract‹ system implementation.

2 Related Work

XPath expressions ([BBC+01]) reflect a core building block in querying XML documents, either standalone or within an XQuery expression ([CFR+01]). An XPath expressions consists of two main features. A location step directly reflects a predicate within the hierarchical structure of an XML document where textual information is recursively wrapped by tags optionally holding additional attributes. Starting with a context node (either the root node of an XML document or any other well-specified node), an XPath expression identifies a set of subtrees of the original XML document. Each location step corresponds to a navigation step based on an axis within the document and the application of a predicate. Figure 2 illustrates the semantics of different location steps for a given context node and an hierarchical XML document tree.

The most prominent types of location steps are a ›single downward-step‹ (/ or /child::) and ›any number of downward-steps‹ (// or /descendant-or-self::node()/). For example /a/b retrieves all subtrees starting with a <b>-tag at the second level in the document directly under an <a>-tag. In the opposite, the expression /a//b results in all <b>-rooted subtrees somewhere in a part of the document starting with an <a> tag. Analogously to file systems, two dots (..) navigate to the next higher level.

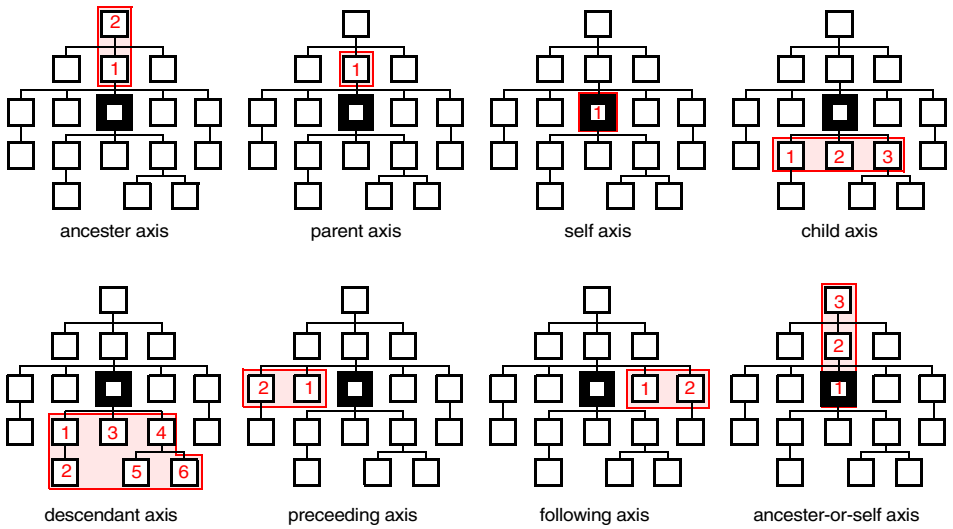


Fig. 2: XPath Location Steps

The application of predicates at a certain step may consist of a simple tag name (like in the example above), of a wildcard (*), and a filter specification within [ ]-parentheses. For example, /a/*/b returns a list of subtrees starting with <b> found at the third level originally rooted by an <a> tag. A filter expression may exhibit again multiple location steps and the reference of an attribute. For example the expression /a[b/c]/d returns all 2nd-level <d>-tag rooted subtrees if the corresponding parent <a>-tag holds a <c> tag as a grandchild with <b> as the direct descendant. Attributes are referenced by prefixing the attribute name with @. The above filter expression may be extended to hold an attribute z with value 'XML', yielding /a[b/c]/d[@z='XML'].

The XPath framework allows to address fragments of an XML document in a very complex manner and is either used isolated or as a core component in the XQuery language to specify more complex operations like joins and aggregates. To efficiently evaluate XPath expressions, multiple implementations and optimization are proposed in the literature. The classical way to evaluate XPath expressions is to create a main memory document object model (DOM; [WHA+00]) for a specific XML document and traverse the internal object structure (like Jaxen; http://www.jaxen.org). However many optimizations exist to speed up the XPath evaluation process.

On the one side, either special index structures like DataGuide ([GoWi97]), the approach of [LiMo01] or the APEX index ([ChMS02]) are proposed or existing multidimensional indexing technology like R-trees is used to support the efficient evaluation of XPath expressions ([Grus02]). On the other hand, filtering techniques are brought into discussion to index the queries, i.e. path expressions, instead of data ([MiSu99], [LaPa02]). The approach of [AlFr00] for example relies on the concept of a finite state machine so that processing a location step means switching a single transition in the machine. Each state con-

sists of a quadruple holding information regarding the current state of processing a single query. Query indexing considers / and // expressions including wild cards and the application of filters in the context of nested path expressions. Extending this idea, the approach of [CFGR02] performs substring decomposition on a syntactical level and defines common subpath expressions as clusters to reduce the number of states. Multiple clustered XPath expressions are additionally indexed using a specialized index structure.

The main idea of our contribution to speed up XPath filter expressions is to provide a preprocessing framework *in addition* to existing XPath filtering techniques by applying the following optimization steps:

- clean and transform registered XPath expressions during a prepare phase using the registered XML document schema.
- analyze XPath expressions and generate a chunking scheme for the incoming XML documents so that individual XPath expressions are evaluated on much smaller XML document fragments.
- exploit parallelism when executing XPath queries on XML data fragments without loosing any filtering capability.

The following section outlines the necessary transformations of the registered XPath expressions during prepare and runtime, while section 4 discusses the proposed chunking scheme.

3 The Basics: XPath Transformation

Once a subscriber registers an XPath expression at the information broker, the expression is added to the subscription database (Figure 1). In a single XPath filtering engine scenario, an incoming document is matched against the XPath expressions and the result is written into an XML result set document (Figure 3a). In the proposed way of applying XPath filters based on chunks, all XPath expressions are partitioned into smaller XPath sets using a stream-oriented interface; each set is then evaluated on a smaller XML document fragment which is sufficient to answer the allotted XPath expressions (Figure 3b).

Without any assumption of the underlying XPath evaluation method, it should be clear that applying less queries on a smaller XML document should speed up the overall filtering process. Even a sequential execution benefits from the preprocessing step done during the prepare phase. It is worth mentioning here that the *prepare phase* transforming XPath expressions and defining the chunking scheme for the XML documents does not influence the time needed to apply XPath filter expressions in a specific document. The time-critical *filtering phase* encompasses the creation of chunks and the evaluation of the registered XPath expressions. Moreover it should be noted that the chunking scheme may be incrementally adapted after a new subscription is added to the subscription base. The revised chunking scheme then applies to the next incoming XML document.

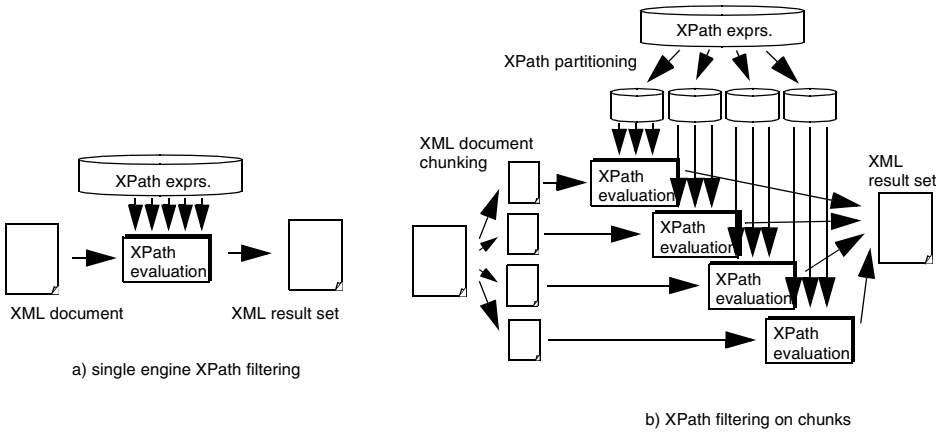


Fig. 3: Single Engine versus Chunked XPath Filtering

Cleaning and Transforming XPath Expressions

The preparing step of a set of XPath expressions applies transformation rules to convert all expressions into a standardized form, which is then subject of computing the chunking scheme. The overall goal is to generate XPath expressions with simple /-location steps and filters to provide a smooth foundation for the prefix-oriented chunking scheme.

In a first phase, all parent statements are eliminated by converting them to filter expressions. For example /a/b/../c identifies the <c>-nodes which comes after an <a>-node, if the <c>-node exhibits a <b>-node as his sibling. This may be equivalently written as /a[b]/c which may be read as: 'If an <a>-node has a <b>-node as a direct child, then give me the <c>-node children from the <a>-node.'

While the cleaning step is a pure syntactical transformation, the following step of resolving wildcards and //-expressions heavily relies on the existence of a schema definition. Therefore consider the sample DTD* given in figure 4 together with the following two expressions /a/b/*/c and /a/b/*/d. Without any further transformation the expressions would lead to the same prefix /a/b yielding no chunking criterion. However, referring to the XML schema, the resolution of the wildcards results in /a/b/l/c, a/b/m/c and a/b/k/d with potentially three different prefixes. Therefore, the overall goal in transforming XPath queries is to gain expressions with long prefixes so that the chunking scheme has many alternatives when deciding for a chunking point.

```
<!ELEMENT a (b+)  >
<!ELEMENT b (l*,m*,k*)>
<!ELEMENT l (c*)>
<!ELEMENT m (c*,a*)>
<!ELEMENT k (d*)>
<!ELEMENT d EMPTY>
<!ELEMENT c EMPTY>
```

Fig. 4: Sample DTD

* We usually rely on XML-schema but prefer the DTD version due to better readability and more compact presentation.

Considering the prefix problem in more detail, it is more important to resolve //-expressions, because expressions like //a or //b do not exhibit (at first sight) a common prefix. A answer to this problem is to find the first appearance of the nodes after an //-location step in each branch. For example, if – according to a given XML schema definition – a <d>-node appears within the document only in the constellation of /a/b/k/d, then we may transform //d into /a/b/k//d. Unfortunately, an XML document may exhibit an infinite depth if the corresponding schema exhibits a recursion. For example, the expression //c may be unrolled to /a/b/m/c or /a/b/m/a/b/m/c and so on. In this case, the transformation step expands the corresponding XPath expression until a recursion appears in the XML schema.

The last check performed during the cleaning and transforming step by matching each XPath expression against the corresponding XML schema is to eliminate expressions obviously evaluating to an empty result set.

In summary, consider the three sample expressions //c, /a/b/*/d and /a/d with regard to the sample DTD from figure 4. After transformation, the XPath set comprises the expressions /a/b/l//c and /a/b/m//c (resolving the //-expression) and /a/b/k/d (substituting wild cards). It is worth mentioning that the third sample expression (/a/d) is removed from the XPath set because the comparison with the corresponding schema does not provide any positive match. Furthermore, it should be noted that the resolution of a wildcard usually results in multiple possible XPath expressions.

4 The Chunking-Scheme

The overall goal to speed up the XPath filtering process by applying filter expressions on multiple smaller fragments of the original XML document requires an adequate partitioning scheme of the XPath expressions additionally implying a chunking scheme of the XML document. This partitioning mechanism depends on the number of chunks to be generated for the filtering process and the set of XPath expressions complying to the following rules:

- *even distribution*
  Each set of XPath expressions operating on the same chunk should have the same cardinality, i.e. all filtering expressions are supposed to be evenly distributed among the XML fragments.
- *small potpourri set*
  Since there may exist XPath expressions for which an assignment to a single chunk is not possible due to prefix incompatibility between chunk and XPath expression, we additionally keep a ›potpourri set‹ of XPath expressions. This set should be as small as possible, because usually less reduction is possible for the underlying XML document. For example if /a/b and /a/c determines the content of two separate chunks, an XPath expression /a/d would be assigned to the potpourri set. However,

since we explicitly consider filters in the chunking scheme (section 4), the expression /a/b[d]/e would be assigned to the chunk defined by /a/b, which in turn would be extended by the single nodes (not subtrees) addressed with /a/d.

Determine the Chunking Point based on XPath Query Trees

The algorithm to produce a chunking scheme for XML documents operates on a query tree, where each element of an XPath query is represented by a single node. The weight of a query tree node is initially set to 1 and increased with each additional XPath expression represented within the tree. For the sake of illustration, figure 5 shows the corresponding query tree after representing filter expressions /a/b/c, /a/b/d, and /a/b/c.

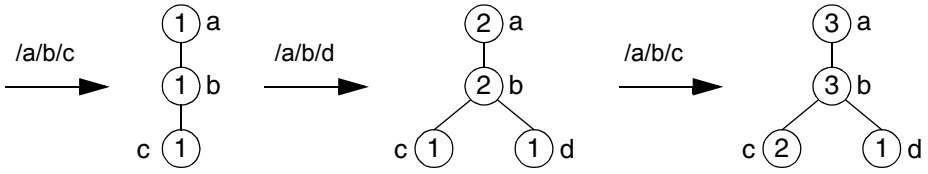


Fig. 5: Populating the Query Tree and Assigning Weigths

The naive approach to generate a chunking scheme would be to sort the nodes by their weight and take the TOP(n)-weighting nodes (with n as the number of chunks) as chunking criteria. In the example above, this would result in /a and /a/b. Unfortunately, since /a represents the root node, the first chunk would be the document itself. Moreover, /a/b also represents (more or less) the whole document, so that the second chunk would not result in any size reduction compared to the original XML document. Obviously the optimal solution for this scenario however would be chunks defined by /a/b/c and /a/b/d.

The revised chunking strategy is illustrated algorithmically in figure 6. To prevent the root node from being selected as a chunking prefix, it is removed from the set of possible chunking candidate nodes in a very first step. The second step consists in finding the node with the highest weight (uheavy) and the largest depth. This is done top-down by walking down a branch if a child has the same maximum weight so that we may get long prefixes finally resulting in small XML document fragments. This heaviest node is selected as a chunking candidate and added to the result list. To prevent the algorithm from picking the same prefix (or part of the prefix) in subsequent runs, we subtract the weight of the candidate node from all nodes (including the node itself) up to the root as long as the overall weight does not yield a negative value.

Referring to the sample query tree of figure 5, node /a/b would be selected as candidate node and added to the result list. The resulting scenario after reducing the weight of all nodes starting at /a/b up to the root (in this case /a) is depicted in figure 7a.

The next iteration in producing XML document chunks picks /a/b/c as a candidate node, because /a/b is no longer a valid choice and /a/b/c is the node with the highest weight. The aligning process of the weights within the query tree yields a reduction of node /a/b by the weight of /a/b/c (figure 7b). However, /a is not aligned, because its weight was already re-

```
Algorithm: GenerateChunkingScheme
Input:      D          // query tree node(name,weight)
            C          // number of chunks to generate
Output:     S          // nodes in the prefix tree representing chunks

BEGIN
    V=∅          // visited nodes
    S=∅          // result set of prefix tree nodes
    DO
        N = traverse(D) - {Root(B)}    // all nodes of the query tree without the root
        u_heavy = 0

        // search for next heaviest node not yet in the result
        FOREACH u ∈ (N-S)
            IF (weight(u) > weight(u_heavy))
                u_heavy := u
            END IF
        END FOREACH

        // while a child has the same weight, take the child
        FOREACH c ∈ (children(u_heavy))
            IF (weight(c) == weight(u) AND c ∉ S)
                u_heavy := c
            EN DIF
        END FOREACH

        // subtract weight from parents
        FOREACH p ∈ (parent(u_heavy))
            weight(p) = weight(p) – weight(u_heavy)
            IF (weight(p) < 0)
                weight(p) = 0
            END IF
        END FOREACH
        IF (u_heavy ∉ V)
            V = V ∪ {u_heavy}
        ENDIF

        // remove lighter nodes from the result set
        FOREACH e ∈ (S)
            IF (weight(e) < weight(u_heavy))
                S = S-{e}
            ENDIF
        END FOREACH
        S = S ∪ {u_heavy}

    // do this while there are chunks to be generated
    WHILE (|S| < C OR |S| = |N|)

    // return the result set
    RETURN (S)
END
```

Fig. 6: Algorithm to Generate the Chunking Scheme

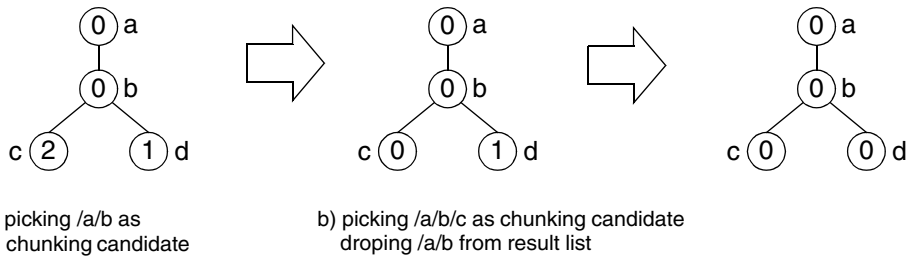


Fig. 7: Sample Query Trees during Developing the Chunking Scheme

duced by the candidate selection process of /a/b. This operation however produces a surprise. The former selected candidate node /a/b now holds a smaller weight than the current candidate node /a/b/c so that the first node does no longer deserve the role of a candidate and is removed from the result list.

The following run of the main DO-WHILE loop produces /a/b/d as the node with the highest weight and the highest depth (compared to /a/b!). Following the same procedure as above, the weight of /a/b and /a are reduced to 0.

Although candidates are potentially removed from the resulting set, the chunking algorithm terminates for any given input, because – in the worst case – all leaf nodes are selected and the weight of the inner nodes are reduced to zero. From a complexity perspective, we obtain $O(n*N)$ with N as the number of nodes in the query tree and n as the number of chunks to create. In the optimal case no candidate node has to be removed from the result list. In the opposite, during the worst case, every node is inserted and removed again on a specific path from the root to a certain leaf node of the query tree. Thus the overall complexity increases to $O(h*n*N)$ with h as the height of the query tree. It may be noted here that the generation of the chunking scheme is performed during the prepare phase and does not count to the time needed to apply a set of XPath filters to an XML document.

Considering XPath Expression with Filters

Considering only simple XPath prefixes as a chunking criterion does not allow the assignment of XPath queries to the corresponding chunks if the queries hold additional filter expressions, so that the chunking scheme would be too strict, i.e. information necessary for evaluating the predicates would be not longer available. Therefore the proposed chunking approach additionally considers filters in adding a set of XPath expressions to identify single XML document nodes to the chunk. For example, if the XPath query is /a/b[c]/d/e and the chunking scheme results in an XML document fragment for /a/b/d, then we add the branch leading to /a/b/c without the remaining subtree, i.e. only the <c>-tagged entry, to the XML fragment.

Figure 8 illustrates the example. The left side shows the XML document fragment holding the part of the original (much larger) document for the chunking prefix /a/b/d. To be able to evaluate the considered XPath query /a/b[c]/d/e during runtime, the chunking scheme is expanded to hold the <c>-tagged node without the subtrees originally rooted by <c>. The

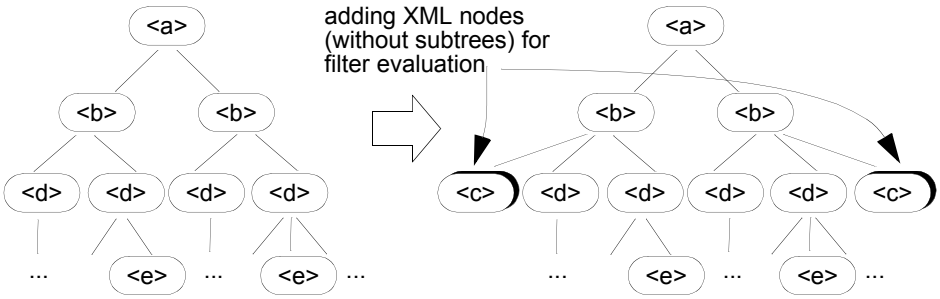


Fig. 8: XML Document Chunks with Nodes Needed to Evaluate Filters in XPath Expressions

capability of resolving filter expressions referring to information from a part of the XML document above the splitting point defined by the chunking criterion requires to retain the structure of the XML document up to the root. This implies that the chunking process for a specific XML document during the filtering phase has a memory requirement to store k XML tag names with k as the maximum of the longest path from the root to any leaf node without any recursion on that path and the depth of the XML document up the first occurrence of a recursion.

Summary of the Process Creating a Chunking Scheme

The transformation steps discussed in the preceding section are a necessary prerequisite to come up with a reasonable chunking scheme shrinking the XML document to a fragment needed to answer a certain set of XPath queries. The chunking scheme is computed based on a weighted query tree with nodes reflecting a single location step. The chunking criteria are iteratively picked so that queries are equally distributed with regard to the generated XML fragments. Locally optimal solutions are discarded and replaced by better solutions during the algorithm. Additionally the chunking scheme considers single nodes of the XML document needed to evaluate XPath queries with filters. This leads to the definition of an XML chunk.

Definition: *XML document chunk*
An XML document chunk C is a tuple $(P, \mathcal{F})$ with P denoting a simple XPath expression identifying the subtrees of an XML document as the base for the XML chunk. The component $\mathcal{F}$ is a set of simple XPath expressions identifying single nodes of the original XML document for evaluating filter expressions.

With the notion of an XML document chunk, we are now able to define a filtering scheme for matching incoming XML documents against a set of registered XPath queries Q split into $n+1$ subsets $Q_1,..., Q_n, Q_{n+1}$ so that the $Q = \bigcup_{i=1}^{n+1} Q_i$.

Definition: *XPath filtering scheme of degree n*
An XPath filtering scheme of degree n consists of a set of $n+1$ tuples with a combination of XML document chunks C_i and a set of XPath query expressions Q_i ($1 \le i \le n+1$), i.e. $\{(C_1,Q_1), ..., (C_n,Q_n), (C_{n+1},Q_{n+1})\}$ so that all XPath expression of Q_i show the same prefix P_i of the corresponding XML document chunk C_i and all filter expressions $\mathcal{F}_i$ occurring in

Q_i can be checked by referring to the nodes specified by the XPath expressions of $\mathcal{F}_i$ for $1 \leq i \leq n$. The potpourri set of XPath query expressions Q_{n+1} can be evaluated referring to chunk C_{n+1}. This potpourri chunk C_{n+1} is either empty (if Q_{n+1} is empty) or corresponds to the original XML document.

5 Performance Evaluations

This section illustrates the core issues of the implementation of the proposed chunking framework together with performance figures comparing runtimes of evaluating multiple XPath queries based on XML document fragments with the original file.

Architecture of the Implementation and Technical Setup

The complete filtering process of XPath expressions is implemented in the context of the ›eXtract‹ project. The eXtract system architecture consists of a collection of different tools written in Java to convert XML documents and/or XPath expressions. Figure 9 gives an overview of the filtering process using eXtract tools.

During the prepare phase, a given set of XPath expressions is cleaned and converted into a standardized format using TRANSTOOL followed by building the query tree and determining the chunking criteria based on the algorithm given in the preceding section (PREPTOOL). The result of the prep tool is the complete filtering scheme with multiple chunk definitions and the associated XPath expressions.

During filtering time, an incoming XML document is given to the stream-oriented CHUNKTOOL which performs a prefiltering step keeping only the parts of the original XML document needed to fulfill the current filtering scheme. Finally, the generated XML fragments are used by the FILTERTOOL to evaluate the XPath queries producing the final result of (in many cases very) small XML documents to be delivered to the subscribers.

The following performance test were carried out using our eXtract implementation, written in Java 1.3.1_03 using the SUN XML Pack Spring 0.2 dev bundle package (http://java.sun.com) and additionally the Universal Java XPath Engine JAXEN 1.0beta8 (http://www.jaxen.org) to evaluate XPath expressions. The tools were running on a WinXP machine with an 800MHz Athlon processor and 384MByte memory.

5.1 Scenario 1: University Organization

The schema of the XML scenario we used to demonstrate the benefit of reducing the size of XML documents before feeding it into the filtering engine is given in appendix A1. The sample XML file counts 130.000 XML tags resulting in 4.85MByte size on disk. As can be seen in the DTD, the scenario holds five different blocks of information (arbitrarily mixed within the XML file).

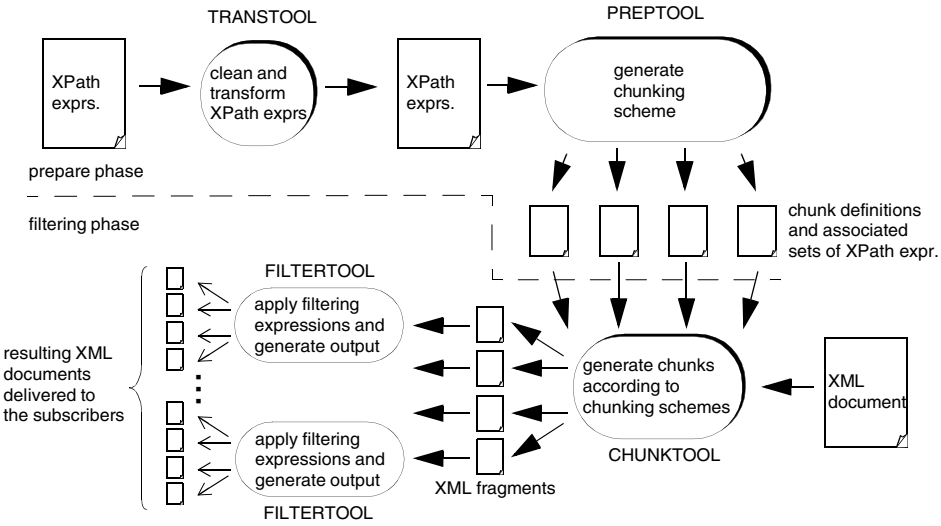


Fig. 9: Base architecture of the eXtract filtering engine implementation

All performance evaluations were carried out using 100, 500, and 2000 registered XPath queries. The query set was synthetically generated so that 30% of the queries exhibit a //-expression and 30% of the queries exhibit an additional filter expression. Figure 10 shows the distribution of the queries referring to one of the five main partitions of the XML document for each scenario separately. Additionally, figure 10 shows the prefixes not selected as real chunks. For example, if we have a chunking degree of 4 (figure 10a), then all students go into the potpourri for the 100 query scenario, the secretaries for the 500 query scenario, and finally professors are making up the potpourri when considering the 2000 query scenario. If the number of real chunks is reduced to 3, then two types of employees are assigned to the potpourri chunk (figure 10b).

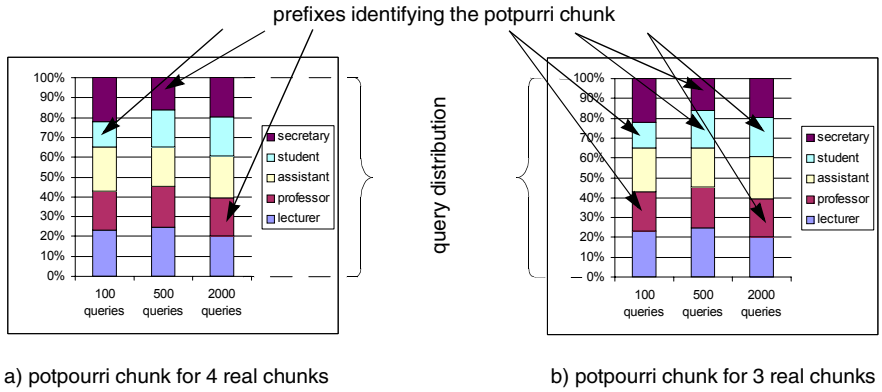


Fig. 10: Query distribution for 100, 500, and 2000 XPath expressions

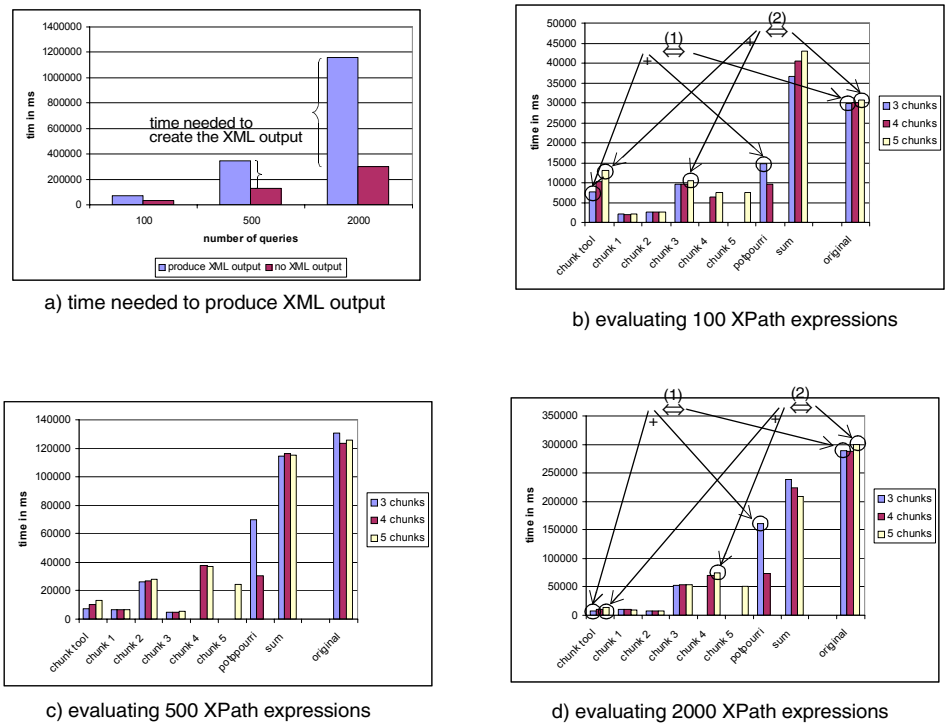
Interpretation of the Performance Measurements

The first diagram of figure 11 shows the different runtimes needed to apply XPath query expressions and to generate the XML output for the subscribers. Although, producing the XML output streams does not influence the results relatively to each other, we omit the result generation when considering the following scenarios to focus on the time needed for the filtering step. Each of the following three scenarios is executed on a filtering scheme with five, four, and finally three real chunks. While the chunking scheme with five real chunks produces an empty potpourri query set, the case with four and three chunks generates a potpourri query set which has to be evaluated with regard to the original XML document.

The first component of the diagram of figure 11 denotes the time needed to perform the chunking of the original document according to the given filtering scheme. The numbers for chunk 1 up to chunk 5 together with potpourri denote the time needed to perform the filtering of the associated XPath expressions. The sum component is used to compare the overall filtering time for each chunking scheme with the time needed to perform the filtering based on the original XML document without any modification and optimization.

While the filtering process for 100 queries (figure 11b) based on the original XML document requires less time compared to the sequential execution of the chunked version, the chunking scheme outperforms the original scenario, if we consider a naive parallel environment. Since parallel execution (in theory) would be bound to the longest running filtering step plus the time needed for the initial chunking of the XML document, the filtering process for the 5-chunk scenario would be finished after 10325ms+13059ms for the chunking compared to 30624ms for the sequential filtering method, thus gaining 24% performance speedup ((2) in figure 11b). For the case with 3 chunks, the required potpourri chunk slows down the execution. However, due to a smaller chunking time, the speedup reaches again 25% ((1) in figure 11b).

As can be seen in the figure 11c and 11d, the higher the number of XPath expressions to evaluate the higher the speedup gained by the chunking mechanism. For 500 and 2000 queries the sequential execution is already faster than the method based on the complete XML document, so that prefiltering sounds attractive. More detailed, the 2000 query scenario reaches a performance gain of 41% for the scenario with 3 chunks and a potpourri chunk (160981ms + 7491ms compared to 289365ms; (1) in figure 11d). The gain rises to 71% when applying the almost optimal filtering scheme with 5 chunks resulting in an empty potpourri chunk ((2) in figure 11d). In this case, chunk 4 needs 74126ms, resulting in a benefit of 71% when considering an initial chunking period of 12829ms and 298980ms needed to perform 2000 XPath queries based on the original XML document.



a) time needed to produce XML output

b) evaluating 100 XPath expressions

c) evaluating 500 XPath expressions

d) evaluating 2000 XPath expressions

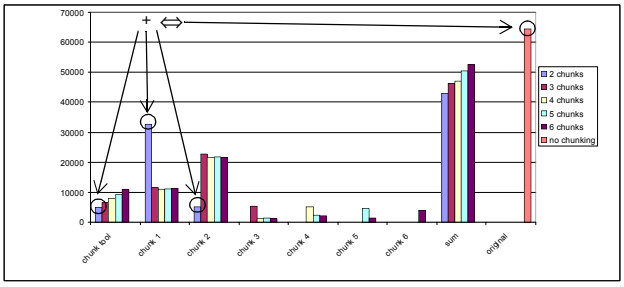
Fig. 11: Performance Measurements for Scenario 1

5.2 Scenario 2: EJB Deployment Descriptor

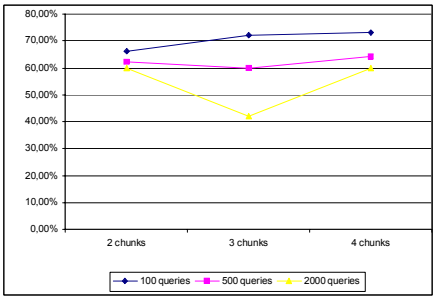
A second scenario demonstrating the feasibility of the proposed chunking approach is based on an EJB deployment description of a large database application project. The DTD is given in appendix A2. The cardinalities of the resulting chunks with the selected XPath prefixes and the query distribution are given in table of appendix A3.

Figure 12a illustrates the result of performance studies for this EJB scenario. For a query set with 100 queries, splitting the original file into two chunks already yields a performance reduction from 64473ms to (5027+(32757+5087))=42871ms for chunking and XPath evaluation with regard to the single chunks. It is interesting to note that in this specific context this splitting scheme seems to be the optimal chunking scheme, because further splits increase the overall time needed to evaluate the XPath queries.

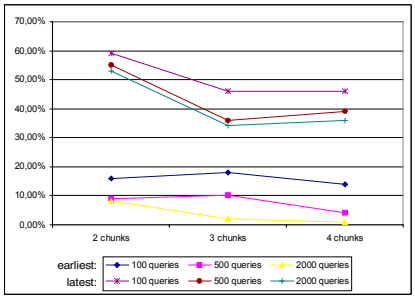
Figure 12b und c show a summary of performance gains. The average runtime needed for a chunking scheme with 2, 3, and 4 chunks is given in figure 12b for query sets with 100, 500, and 2000 XPath expressions. An optimization with only two chunks results in a reduction of the runtime to 60-70% compared to the original case. Since the time needed for the chunking tool and the query execution costs for the individual chunks are summarized



a) 100 XPath queries with a chunking scheme of 2 to 6 chunks



b) ratio of chunk-based compared to regular XPath evaluation



c) earliest and latest result delivery

Fig. 12: Performance Measurements for Scenario 2

and therefore comparable with the execution based on the original XML document, the chunking scheme exhibits another advantage. Individual chunks with the corresponding XPath expressions may be prioritized so that the execution order determines the time of the earliest/latest delivery of the XPath evaluations. Figure 12c holds this information computed by the time needed for the chunking plus the minimum/maximum of the XPath expression evaluation based on the chunks compared to the original runtime. For example, when evaluating 2000 XPath expressions on three chunks, the first query set is ready after 2% of the original query runtime.

6 Summary and Conclusion

Providing an efficient filtering mechanism is the core to build an efficient and large scale information dissemination system. Since XML is the base for exchanging data in loose-coupled information systems, we focus on improving the filtering mechanism for subscriptions specified as XPath expressions evaluated on incoming XML document with known schema. The main idea of our approach is to analyze the registered set of XPath expressions during a prepare phase and generate a filtering scheme to partition the set of queries and splitting the incoming XML document into separate chunks. The size reduction of the

XML document is crucial for applying the filtering mechanism. Comprehensive performance evaluations demonstrate performance gains even in case of a sequential execution of the XPath query sets based on the associated XML chunks. Moreover, the proposed solution might be starting point for priority scheme

The advantage increases when performing the filtering process in parallel based on the customized XML chunks. Since our preprocessing step is independent of the underlying filtering technique, is can be integrated into an existing subscription evaluation system.

References

AAB+98 Altinel, M.; Aksoy, D.; Baby, T.; Franklin, M.; Shapiro, W.; Zdonik, S.: DBIS-Toolkit: Adaptable Middleware For Large Scale Data Delivery. In: *SIGMOD'99*, pp. 544-546

AgCL91 Agrawal, R.; Cochrane, R.; Lindsay, B.: On Maintaining Priorities in a Production Rule System. In: *VLDB'91*, pp. 479-487

AlFr00 Altinel, M.; Franklin, M.J.: Efficient Filtering of XML Documents for Selective Dissemination of Information. In: *VLDB 2000*, pp. 53-63

ASS+99 Aguilera, M.; Strom, R.; Sturman, D.; Astley, M.; Chandra, T.: Matching Events in a Content-Based Subscription System. In: *PODC'99*, pp. 53-61

BaWi01 Babu, S.; Widom, J.: Continous Queries over Data Streams. In: *SIGMOD Record 30(3), 2001*, pp. 109-120

BBC+01 Berglund, A.; Boag, S.; Chamberlin, D.; Fernandez, M.F.; Kay, M.; Robie, J.; Simeon, J.: XML Path Language (XPath) 2.0. Working Draft, Version 2.0, World Wide Web Consortium (W3C), 2001, http://www.w3c.org/TR/xpath20/

BeCr92 Belkin, N.J.; Croft, W.B.: Information Filtering and Information Retrieval: Two Sides of the Same Coin? In: *CACM, 35(12),1992*, pp. 29-38

CDTW00 Chen, J.; DeWitt, D.J.; Tian, F.; Wang Y.: NiagaraCQ: A Scalable Continous Query System for Internet Databases. In: *SIGMOD 2000*, pp. 379-390

CFGR02 Chan, C.-Y.; Felber, P.; Garofalakis, M.N.; Rastogi, R.: Efficient Filtering of XML Documents with XPath Expressions. In: *ICDE 2002*, pp. 235-244

CFR+01 Chamberlin, D.; Florescu, D.; Robie, J.; Simeon, J.; Stedanescu, M. (Hrsg.): XQuery: A Query Language for XML, Working Draft. World Wide Web Consortium (W3C), 2001, http://www.w3c.org/TR/xquery/

ChMS02 Chung, C.-W.; Min, J.K.; Shim, K.: APEX: An Adaptive Path Index for XML Data. In: *SIGMOD 2002*

Fall01 Fallside, D.C.: XML Schema Part 0: Primer. W3C Recommendation, World Wide Web Consortium (W3C), 2001, http://www.w3c.org/TR/xschema-0

FJL+01 Fabret, F.; Jacobsen, H.-A.; Llirbat,F.; Pereira, J.; Ross, K.A.; Shasha, D.: Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In: *SIGMOD 2001*, pp. 115-126

FoDu92 Foltz, P.W.; Dumais, S.T.: Personalized Information Delivery: An Analysis of Information Filtering Methods. In: *CACM 35(12), 1992*, pp. 51-60

GoWi97 Goldmann, R.; Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Database. In: *VLDB'97*, pp. 436-445

Grus02 Grust, T.: Accelerating XPath Location Steps. In: *SIGMOD 2002*

HCH+99 Hanson, E.N.; Carnes, C.; Huang, L.; Konyala, M.; Noronha, L.; Parthasarathy, S.; Park, J.B.;Vernon, A.: Scalable Trigger Processing. In: *ICDE'99*, pp. 266-275

KrRo95 Krishnamurthy, B.; Rosenblum, D.; Yeast: A General Purpose Event-Action System. In: *TSE 21(10), 1995*, pp. 845-857

LaPa02 Lakshmanan, L.V.S.; Parthasarathy, S:: On Efficient Matching of Streaming XML Documents and Queries. In: *EDBT 2002*, pp. 142-160

LiMo01 Li, Q.; Moon, B.: Indexing and Querying XML Data for Regular Path Expression. In: *VLDB 2001*, pp. 361-370

LiPT99 Liu, L.; Pu, C.; Tang, W.: Continual Queries for Internet Scale Event-Driven Information Delivery. In: *TKDE 11(4), 1999*, pp. 610-628

LPCZ01 Lehner, W.; Pirahesh, H.; Cochrane, R.; Zaharoudakis, M.: fAST Refresh using Mass Query Optimization. In: *ICDE 2001*, pp. 391-398

MiSu99 Milo, T.; Suciu, D.: Index Structures for Path Expressions. In: *ICDT'99*, pp. 277-295

PFJ+01 Pereira, J.; Fabret, F.; Jacobson, H.A.; Llirbat, F.; Shasha, D.: WebFilter: A High-throughput XML-based Publish and Subcribe System. In: *VLDB 2001*, pp. 723-724

RoWo97 Rosenblum, D.S.; Wolf, A.L.: A Design Framework for Internet-Scale Event Observation and Notification. In. *SIGSOFT'97*, pp. 344-360

WHA+00 Wood, L.;Hors, A.L.; Apparao, V.; Byrne, *S.;* Champion, M.; Isaacs, S.; Jacobs, I.; Nicol, G.; Robie, J.; Sutor, R.; Wilson, C.: Document Object Model (DOM) Level 1 Specification (Second Edition).W3C Working Draft. World Wide Web Consortium (W3C), 2000, http://www.w3.org/TR/REC-DOM-Level-1

YaGa95 Yan, T.W.; Garcia-Molina, H.: SIFT - A Tool for Wide Area Information Dissemination. In: *USENIX'95*, pp. 177-186

Appendix A: Description of Sample Scenarios

A1) DTD and Chunk Size of Sample Scenario 1

```
<!ELEMENT db (student+,professor+,assistant+,secretary+,lecturer+) >
<!ELEMENT student (name,email?,url?,link?)>
<!ATTLIST student id ID #REQUIRED>
<!ELEMENT professor (name,email?,url?,link?)>
<!ATTLIST professor id ID #REQUIRED>
<!ELEMENT assistant (name,email?,url?,link?)>
<!ATTLIST assistant id ID #REQUIRED>
<!ELEMENT secretary (name,email?,url?,link?)>
<!ATTLIST secretary id ID #REQUIRED>
<!ELEMENT lecturer (name,email?,url?,link?)>
<!ATTLIST lecturer id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ATTLIST url href CDATA #REQUIRED>
<!ELEMENT link (#PCDATA)>
<!ATTLIST link manager IDREF #IMPLIED subordinates IDREFS #IMPLIED>
```

# of chunks / size in KB	3	4	5
chunk 1	170	170	170
chunk 2	206	206	206
chunk 3	1233	1233	1233
chunk 4		1145	1145
chunk 5			1537
potpurri chunk	4969	4969	0

chunk and potpurri size

A2) DTD of Sample Scenario 2

```
<!ELEMENT assembly-descriptor ( container-transaction+, security-role+, method-permission+ ) >
<!ELEMENT cmp-field ( field-name ) >
<!ELEMENT container-transaction ( method+, trans-attribute ) >
<!ELEMENT ejb-jar ( enterprise-beans, assembly-descriptor ) >
<!ELEMENT ejb-ref ( ejb-ref-name, ejb-ref-type, home, remote, ejb-link ) >
<!ELEMENT enterprise-beans ( message-driven, session+, entity+ ) >
<!ELEMENT entity ( ejb-name, home, remote, ejb-class, persistence-type,
                   prim-key-class, reentrant, cmp-field+, primkey-field? ) >
<!ELEMENT message-driven ( ejb-name, ejb-class, message-selector, transaction-type,
                           acknowledge-mode, message-driven-destination, ejb-ref+ ) >
<!ELEMENT message-driven-destination ( destination-type ) >
<!ELEMENT message-selector EMPTY >
<!ELEMENT method ( ejb-name | method | method-intf | method-name | method-params )* >
<!ELEMENT method-params ( method-param* ) >
<!ELEMENT method-permission ( role-name, method+ ) >
<!ELEMENT run-as ( role-name ) >
<!ELEMENT security-identity ( run-as ) >
<!ELEMENT security-role ( description, role-name ) >
<!ELEMENT session ( ejb-class | ejb-name | ejb-ref | home | remote |
                    security-identity | session-type | transaction-type )* >
```

all other not explicitly mentioned ELEMENTs are of type #PCDATA

A3) Chunk Size and Query Distribution of Sample Scenario 2

number of chunks	relative query distribution	prefix selection for individual chunks	chunk cardinality
Chunk 1	54%	/ejb-jar/assembly-descriptor	3,972
Chunk 2	46%	/ejb-jar/enterprise-beans	681
Chunk 1	19%	/ejb-jar/assembly-descriptor/container-transaction	1,351
Chunk 2	35%	/ejb-jar/assembly-descriptor	3,972
Chunk 3	46%	/ejb-jar/enterprise-beans	681
Chunk 1	19%	/ejb-jar/assembly-descriptor/container-transaction	1,351
Chunk 2	18%	/ejb-jar/assembly-descriptor/method-permission	2,583
Chunk 3	17%	/ejb-jar/assembly-descriptor/security-role	41
Chunk 4	46%	/ejb-jar/enterprise-beans	681
Chunk 1	19%	/ejb-jar/assembly-descriptor/container-transaction	1,351
Chunk 2	18%	/ejb-jar/assembly-descriptor/method-permission	2,583
Chunk 3	17%	/ejb-jar/assembly-descriptor/security-role	41
Chunk 4	16%	/ejb-jar/enterprise-beans/session	162
Chunk 5	30%	/ejb-jar/enterprise-beans	681
Chunk 1	19%	/ejb-jar/assembly-descriptor/container-transaction	1,351
Chunk 2	18%	/ejb-jar/assembly-descriptor/method-permission	2,583
Chunk 3	17%	/ejb-jar/assembly-descriptor/security-role	41
Chunk 4	16%	/ejb-jar/enterprise-beans/session	162
Chunk 5	15%	/ejb-jar/enterprise-beans/message-driven	44
Chunk 6	14%	/ejb-jar/enterprise-beans/entity	497