FINITE SEGMENTATION FOR XML CACHING

Adelhard Türling and Stefan Böttcher
Faculty of electrical engineering, computer science and mathematic, Fürstenallee 11, D-33102 Paderborn, Germany, Email: Adelhard.Tuerling@uni-paderborn.de, stb@uni-paderborn.de

Abstract: XML data processing often relies on basic relations between two XML fragments like containment, subset, difference and intersection. Fast calculation of such relations based only on the representing XPath expression is known to be a major challenge. Recently XML patterns have been introduced to model and identify handy subclasses of XPath. We present the concept of ST-pattern segments that uses sets of adapted tree patterns in order to describe a finite and complete partitioning of the XML document's data space. Based on such segmentations, we present a fast evaluation of XML relations and show how to compute a set of patterns for an optimal segmentation based on frequent XPath queries.

Key words: mobile databases; XML; query patterns; XPath; caching.

1. INTRODUCTION

Whenever XML data is exchanged, processed and cached on computers within a network, data management meets new challenges. For example, in networks of resource-limited mobile devices, efficient usage of data storage and data transportation over a wireless network is a key requirement[10-12]. In such a network, a common situation is that a client queries for data of a dedicated source. Within such a network, it may be of considerable advantage to share and exchange cached XML data among several neighboring clients, compared to a solution where data is only transferred between each requesting client and a dedicated server. One of the main new challenges in such a data sharing scenario is the organization of the data space which is shared among the clients. This includes specifying how the data space can be divided into handy segments, how to profit from

distributed data according to these segments, and how cooperative usage in a network can enhance data processing. A basic challenge of fragmentation is to identify a finite set of atomic XML fragments for cooperative usage. Whether or not data segments have to be requested in order to fulfill an operation, must be decided by data processing components on the fly, without losing time for extensive intersection tests and difference fragment computations on XML data. To enable collaborative use of a so called segmentation, we identify two requirements for the segmentation's atomic data units, namely the segments. Firstly, segments can be easily (re-)joined and identified (minimal operating costs). Secondly, most query results can be represented by such segments or joins of such segments with little or no dispensable offset (fitting granularity). Obviously there is a conflict between the requirement of a fitting granularity and the need of a finite and collaboratively accepted segmentation. We address this area of conflict and show how to find an optimal segmentation based on access frequency analysis of XML patterns.

The remainder of our paper is organized as follows. In Section 2, we propose to expand the common definition of patterns towards what we call ST-pattern and give a short introduction in the main features and properties. In Section 3, we show in detail how to use the most frequent patterns as a base to decompose the data space into disjointed segments. In Section 4, we discuss related work. And within section 5, we present the summary and conclusion of our contribution.

```
<!ELEMENT car EMPTY>
<!ATTLIST car
    name CDATA #REQUIRED
    year CDATA #REQUIRED
    price CDATA #REQUIRED
    type (truck | convert | limo)
#REQUIRED
>
<!ELEMENT contact EMPTY>
<!ATTLIST contact
    name CDATA #REQUIRED
    image CDATA #REQUIRED
>
<!ELEMENT offer (seller, car+)>
<!ELEMENT offers (offer+)>
<!ELEMENT seller (contact+)>
<!ATTLIST seller
    town CDATA #REQUIRED
>
```

Figure 1. Example DTD

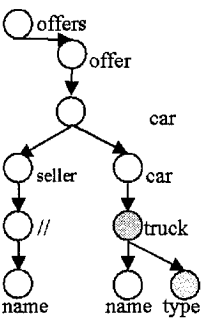

Figure 2. Example for ST-pattern

2. FOUNDATION AND ST-PATTERN

In this section, we shortly review the concept of DTD graphs and XML patterns. We introduce *search-tree patterns* (short ST-pattern) based on additional nodes namely *split nodes* that partition a node's child set. We use ST-patterns as logical data descriptions for data processing that are easy to handle and that allow a good degree of granularity. We here withhold the formal and complete definition of ST-pattern and their operations due to page limitation and refer to future publications. Instead, we give some examples and an overview of properties.

2.1 Definition of the DTD graph

DTDs are schema definitions for XML documents. As long as the DTD is acyclic, such a DTD can be rolled out and represented as a tree. Each element, text-node and attribute occurring in such a DTD is converted to a node in the DTD graph. The parent-child relation (and the attribute-relation) between the elements and the attributes of a DTD are represented by directed edges within the DTD graph. A DTD graph for the DTD of Figure 1 can be seen in Figure 3. In a DTD graph, a '*' is concatenated to a node's label to indicate that the DTD allows the occurrence of that node at that position in an arbitrary quantity, e.g. for `car`, `offer` and `contact`. Ignoring the special annotation '*', a DTD graph can also be seen as an XML pattern.

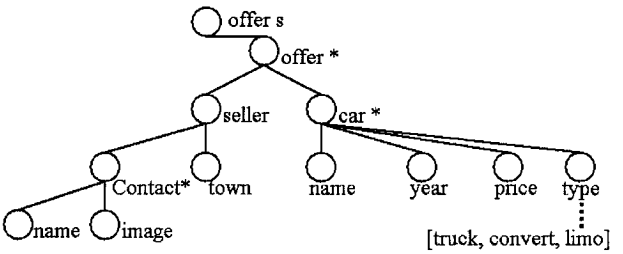

Figure 3. Example DTD graph

2.2 XML patterns

XML tree patterns are used in the context of XML as expressions that describe XML fragments. These patterns can be seen as tree models for XML queries. Nodes of a pattern can be labeled with any tag name, the wildcard '*' or the relative paths '//', where '*' indicates any label and '//' represents a node sequence of zero or more interconnected nodes. Directed edges represent parent → child relations. These edges must correspond to

relations defined in a DTD, e.g. fulfill the restriction of a single incoming edge for each node, to be valid according to the given DTD. Furthermore, we use the same terminology for patterns as used for XML documents. For example, we call all nodes that can be reached by outgoing edges the node's children, the incoming edge leads to the node's parent, all children of a node are in sibling relation and the transitive closure of all nodes reached by outgoing (incoming) edges is called the set of descendent (ancestor) nodes.

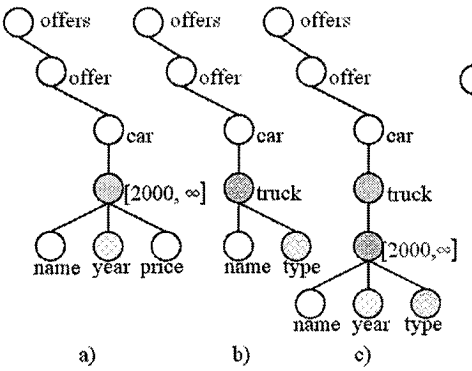

Figure 4. Pattern c) is the intersection of the two ST-patterns a) and b).

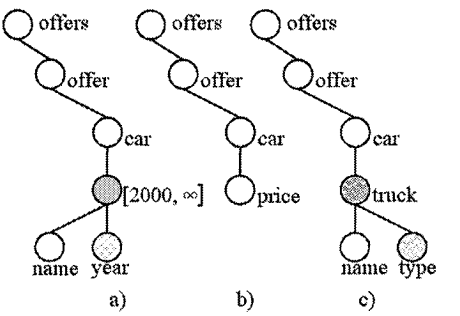

Figure 5. The 3 most frequently accessed patterns.

2.3 ST-patterns

In contrast to basic XML patterns[9], *ST-patterns* are restricted to rooted patterns because they describe XML fragments that correspond to absolute XPath expressions. In addition, we introduce a new node type called *split node* which contains simple selection information as used in XPath filters. As an example in Figure 2, the additional nodes labeled 'truck' and 'type' restrict the pattern to cars of type truck. Such patterns support a minimal subset of possible filter expressions known from XPath, just enough to describe the granularity required. With the DTD given in Figure 1, an XPath query which asks for car-offers of type 'truck' and which is interested in car-names and available car-sellers could be:

offers/offer/*[@type='truck' or self::name()='seller']//@name.

Figure 2 shows the corresponding ST-pattern. We call an XML data-fragment that is selected by an ST-pattern the fragment related to a pattern or for short, *pattern fragment*.

2.4 Operations and properties of ST-patterns

Split nodes are of a specific comparison type and contain specific decision criteria. We distinguish between two types of split nodes: *range-based* and *equality-based* split nodes. See Figures 4a and 4b. Split nodes are related to two nodes in the ST-pattern. The two related nodes are called *split parent* and *reference node* (short ref. node). The ref. node must be a leaf node in the pattern. In our examples throughout the paper, we visualize a split node and its ref. node with an identical texture where the split node is gray and the ref. node is white, indicating that the ref. node must fulfill the split node's decision criteria. The split parent is the first node on the ref. node's ancestor-axis in the pattern that is marked as multiple occurring in the DTD graph. This relation indicates that the sub fragment represented by this split parent is constrained by the split node. The `contact`, `offer` or `car` node might be split parents in our example. A split parent's sub decision tree can have multiple split nodes, which follow a predefined tree-level-based order. We call the complete sub tree of a split parent, its *sub decision tree*.

We define two ST-patterns to be space equal, if they describe the same pattern fragment for a given DTD and for any XML document valid to that DTD. The two operations *compress* and *extend* are used to space equally transform ST-patterns e.g. for normalization purposes.

The three operations *union*, *intersection* and *difference* map two given ST-patterns onto a resulting ST-pattern. For any valid XML document, the resulting ST-pattern describes a pattern fragment that is equal to the result of the given operation applied to the pattern fragments of the two operands.

See Figure 4c as an example for an intersection of the patterns 4a and 4b. We say that two ST-patterns are disjointed, if for every pair of corresponding leaf nodes that they have in common, (1) the two nodes must be ref. nodes and (2) the split nodes they belong to have no overlapping decision criteria.

Evaluating operations on ST-patterns can be done by adapting fast XML match algorithms. Similar to the more complex XPath expressions, ST-patterns are used to select fragments of an underlying XML document and thereby address the document with a fine granularity. For example, any ST-pattern can be split into two patterns, where each of the resulting patterns addresses a fragment with about half the size of the fragment the original patterns addressed. Thus, any fragmentation granularity can be achieved.

3. SEGMENTATION

In Section 2, the definition and some operations for ST-patterns have been introduced. Now we show how these patterns can be used to organize fast XML data processing. Therefore we reconsider that every ST-pattern (based on the DTD) represents a pattern fragment in an XML document (usually in an underlying XML database).

We use pattern fragments as atomic data items in any data processing. In addition, a pattern fragment belongs to a specific pattern segmentation. A pattern segmentation represents a complete decomposition of the whole underlying schema S and is based on the given DTD tree. Beyond the DTD tree detail level, a schema S might even be decomposed by additional equations or ranges on specified node values to support specific requirements. In this section, we shape the requirements for segmentations and show how they are constructed relying on ST-patterns.

3.1 Requirements for a fitting segmentation

It is elementary for the success of data processing to find the appropriate set of patterns which represents the segmentation. Their corresponding pattern fragments are the atomic data units our XML processing is based on. Thus, the patterns shall represent fragments that are handy in the following sense: For a given XPath request, it shall be easy to find the optimal set of patterns where the union U of those patterns relates to an XML fragment that is the smallest possible superset of the XML fragment represented by the XPath request. The parts of the fragment related to U, that are not needed to answer the XPath request, should be minimal or none for frequent requests. We call these parts *clipping offsets* of patterns corresponding to an XPath request. Data transfer overhead caused by the segmentation must be minimal and come out as a clear advantage compared with savings based e.g. on caching.

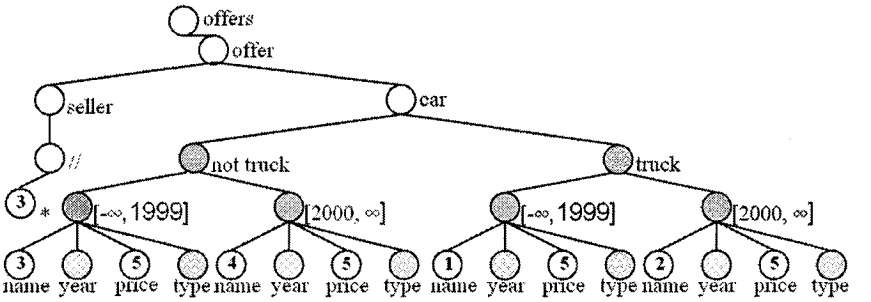

Figure 6. Colored schema graph (numbers represent colors)

3.2 Pattern segmentation

Formally, we define:

A *pattern segmentation* S is a set of pair-wise disjointed ST-patterns $\{p_1, p_2, \ldots, p_n\}$ where the union of all p_i results in a pattern $P_{complete}(S)$ representing the whole data space, e.g. given by the corresponding DTD. The graph representation of $P_{complete}(S)$ is called the segmentation's schema graph. Notice that the DTD graph is a valid schema graph. In general, there are many different valid segmentations for a single schema graph. For the DTD graph, the DTD's pattern itself as well as S = {//*} are valid segmentations with |S| = 1. Figure 8 shows a segmentation with five patterns, Figure 6 shows the corresponding schema graph. To encode the specific segmentation in a schema graph, we introduce colored schema graphs. For example in Figure 6, the numbers inside the nodes represent their colors. All leaf nodes that are not ref. nodes have an associated color identifying the pattern in the segmentation they belong to. The following has to be proven to verify that a set of ST-patterns S is a segmentation:

- For each pair of patterns in S, the intersection test shows that they are disjointed.
- $P_{complete}(S)$ must be space equivalent to the underlying DTD's pattern.

3.3 Glue nodes and the ID constraint

As the set of ST-patterns of a segmentation are pair-wise disjointed, the pattern fragments they describe, describe a pair-wise disjointed partitioning of the XML document's leaf nodes. For a given segmentation, a major requirement is to guarantee that the union, intersection and difference of any two patterns of the segmentation can also be applied to their related XML fragments. Thus, we have to assure to track the pattern fragments'

relationships to each other. In the context of XML trees, one-to-one and one-to-many relationships are supported[4]. For example, in the segmentation of Figure 8, the segment defined by pattern p_3 has a one-to-many relationship to all other segments. The segments defined by the patterns p_1, p_2, p_4, p_5 all have a one-to-one relationship to each other. We define some multiple occurring nodes from the DTD to be *glue nodes* which we require to have a unique key attribute. If the DTD doesn't support the required IDs they can be added in a preprocessing step. We don't have to apply the union, intersection or difference operation to the pattern fragments directly. It is enough to calculate the operations on the corresponding pattern and to use the resulting pattern as a filter for a joined pattern fragment. To guarantee that any two pattern fragments can be joined, we use pairs of IDs and references. In our small example, the `car` node and the `offer` node are both glue nodes and as the DTD does not support ID attributes for them, we have to provide additional ID attributes. To identify a segmentation's glue nodes, each pair of patterns of the segmentation is tested. A pair's glue node is the first multiple occurring node they have in common in the colored schema graph starting at the patterns' leaf nodes. Based on the glue node's ID, any relationship between XML fragments that correspond to a segmentation's patterns can be joined. In our example, the fragment corresponding to the pattern p_3 can be joined with any fragment corresponding to pattern p_2 up to p_5 (one-to-many relationship), by using the `offer` ID as a join criterion. Fragments corresponding to the patterns p_1, p_2, p_4, p_5 can be joined by using the `car` ID as a join criterion (one-to-one relationship). As we see, the number of glue nodes is bounded by the amount of multiple occurring nodes, but can be smaller and is segmentation dependent. For example, the multiple occurring `contact` node is not a glue node, since it is not needed to join pattern fragments.

Input: given DTD graph
Sorted list of most frequent query patterns L= $\{q_1,\ldots, q_n\}$
Initialize: S = {//*}
Ref. node order: O = {Ø}
Max. node index: I_{max} = const. (e.g. 2)
Clip tolerance T = const. (e.g. 0.9)
Max |S|: $|S|_{max}$ = const. (e.g. 100)
10 For each q_i in L do {
11 For each p_J in S do {

[4] XML supports special id/id_ref attributes to support n to n relations. Our techniques support such relationships but are not optimized for them.

```
12  If intersect (q_i, p_j) ≠ Ø  {
13     p_temp1 = compress (intersect (q_i, p_j))
14     p_temp2 = compress (difference (q_i, p_j))
15     If (max_amount_split_node_series(p_temp1) < I_max ) and
16  (max_amount_split_node_series(p_temp2) < I_max ) and
17  ((size(pattern_fragment (p_temp1) / size(pattern_fragment (p_j) < T)) or
18  (size(pattern_fragment (p_temp2) / size(pattern_fragment (p_j) < T)))  {
19     remove p_j from S
20     if not contained(S_temp1.newRefNode,O)
add(S_temp1.newRefNode,O)
21     if not contained(S_temp2.newRefNode,O)
add(S_temp2.newRefNode,O)
22     add(S_temp1, S)
23     add(S_temp2, S)
24  } }
25  break if |S| >= |S|_max
25  }
```

Figure 7. Segmentation algorithm

3.4 Construction of the finite pattern segmentation

The amount of different patterns corresponding to a non-recursive DTD is already finite, if split nodes are not used, because there is a finite set of possible patterns for each set of edges. A pattern with the maximum amount of edges and nodes is the DTD graph itself. As we introduce split nodes, we have to constrain the size of segmentations by a threshold $|S|_{max}$. The value of $|S|_{max}$ correlates with the granularity of the segmentation and must be adjusted application-context dependent, considering DTD complexity and the amount of represented data. In order to keep the pattern set of a segmentation finite, we restrict the amount of segments in a segmentation to a fix maximum $|S|_{max}$. For example, for the DTD graph given in Figure 3, a valid segmentation with $|S| = 5$ is shown in Figure 8. Additionally we might constrain the depth of sub decision trees to limit the segmentation's schema graph complexity and the amount of ref. nodes in a single pattern.

A good solution to establish a fitting segmentation is to analyze the access frequency to certain tree patterns and to build a segmentation according to the most frequently accessed pattern. Our algorithm is based on that concept and takes a sorted list L of the most frequent requests as input. The resulting segmentation can guarantee that any of the requests in L can be answered exactly by joined pattern fragments of the segmentation. The algorithm of Figure 7 creates such a segmentation. Starting with an initial segmentation S = {//*}, it splits patterns until $|S| = |S|_{max}$. For each frequently

requested pattern q_1 of L, it has to be checked with which of the exiting patterns p_i in S it intersects (line12) and for each intersecting pattern p_j the intersection and difference has to be calculated (lines 13, 14). Thereafter, each such p_j is removed from S, and the segments intersect(q_i,, p_J) and difference(q_i,, p_J) are added to the segmentation (lines 19, 22, 23). The ref. node order simply correlates with the sequence in which they are first referenced by a split node (lines 20, 21). Iterating the above steps, we keep the set of patterns in the segmentation disjointed and thus the segmentation valid. Figure 8 shows a possible resulting segmentation for $|S|_{max}=5$. Constructing a segmentation according to the presented algorithm, patterns that are used to answer frequent requests are in general very specific and represent a small segment of the XML document. In comparison, infrequently requested patterns are in general more unspecific in the sense of conglomerations and are related to bigger segments in the XML document.

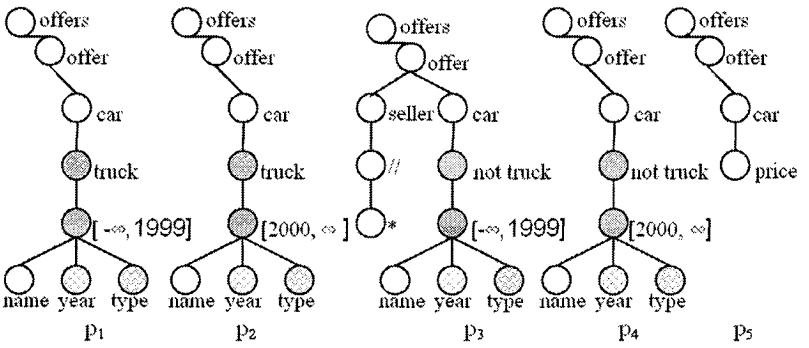

Figure 8. A pattern segmentation for the given DTD

3.5 Thresholds

In addition, the algorithm provides the two thresholds I_{max} and T to adjust the basic segmentation algorithm (lines 15 to 18).

The threshold I_{max} constrains the amount of descendent sequenced split nodes in the resulting patterns. Using this threshold can compensate two drawbacks. First, such sequences can expand a schema graph's sub-tree exponential. Since the schema graph is the 'construction plan' for any further data processing, transmitting and fast processing of the schema graph are important operations and a rather compact graph is preferred. Second, such series result in related XML fragments that are in a one-to-one relation which need to have ID nodes as introduced in 3.3. T is a threshold that

corresponds to the degree of acceptable clipping tolerance. Since the transmission of a slightly bigger pattern fragment is acceptable, this threshold can prevent unneeded granularity. For this threshold, it is important to take the related pattern fragment's size in the XML document into account, rather than just to stick to the pattern's relative fraction of the schema graph. As mentioned, both thresholds must be set up according to the application's domain. The algorithm of Figure 11 creates the valid segmentation shown in Figure 8 for the input L = {5a, 5b, 5c}, T=1, $I_{max} = 2$ and the given DTD.

3.6 Mapping from and to XPath

In general, applications based on XML access XML fragments by XPath expressions. Mapping such an XPath query to patterns of a finite segmentation is easy. Since all patterns of the segmentation are disjointed, we just have to identify which patterns contribute to the result. Therefore, we represent the colored schema graph as an XML document and query it with the given XPath expression. Each color in the result represents a contributing pattern and the join of the related pattern fragments is the minimal superset of the XML fragment selected by the given XPath expression. If after joining the exact result is needed, the obtained superset can be queried by the standard XPath evaluation engine, e.g. a SAX filter. Mapping from ST-patterns to XPath is even simpler. Each node of a pattern represents a node-test, each edge a child-axe and each split node a conjunctive-filter-criterion of its split parent in a corresponding XPath expression.

3.7 Segmentation in application context

As seen in Section 3.5, a fitting segmentation, e.g. for the system introduced in Section 2, can be calculated in a preprocessing step. Thereafter, the colored schema graph, e.g. in number scheme representation form, can be published. Even more, we suggest adapting the segmentation continuously according to query behavior, if the overall clients' focus changes. For example, think of a train schedule where the focus changes naturally with elapsing time. In such cases, an update of the colored schema graph and the information about some invalid pattern segments has to be distributed. A centralized technique, that keeps track of focus changes, is to let the client send the original XPath expression with the request towards the server. The server or an intermediate caching server can analyze the query, can match it with the requested pattern, and can calculate the amount of data that is not needed in the answer-pattern fragment. This information can be used in order to identify inefficient segmentations and can thereby lead to an

adjustment of the segmentation and a reduced response. Cache management for server and client can use the colored schema graph as index structures for lookups and store the pattern fragments in joined form in their memory. Thus, finding the set of missing and locally available segments can be done fast by querying the locally available colored schema graph.

As the originator of a request joins the pattern fragments as they arrive, the IDs introduced in Section 3.3 are used for accurate matching of any two pattern fragments using join optimization concepts[17, 18]. Any cache server contributing more than one segment can even send its pattern fragments in joined form to reduce calculation overhead and transmission of redundant ref. nodes in one-to-one relations. An alternative solution to the usage of filters to obtain the exact answer to the last XPath request as introduced in Section 3.7, is to mark nodes during the join process as 'not belonging to the current request' and thus defining a temporary view.

As well as changes in the segmentation, updates in XML documents must be propagated. A simple solution could be a master server that coordinates updates and distributes the list of effected pattern fragments to indicate that they are outdated. Since ST-patterns guarantee that their decision criterions (the ref. nodes) are included in the pattern fragment, finding and updating affected patterns can be performed in a decentralized manner and thus only a moderate amount of communication between master server and any cache server is needed. For example, after updating an outdated value, the client can decide whether the changed node still belongs to the original pattern or belongs to a different pattern and can publish that information.

3.8 Properties of finite segmentations

Besides the properties of ST-patterns discussed in Section 2, patterns of a finite segmentation have some properties which make them perfectly suitable for XML data processing. As seen in Section 3.6, it is easy to find the set of needed patterns to answer any query. The found set is optimal, since all patterns in the segmentation are disjointed. The algorithm of Figure 7 finds a finite segmentation providing pattern fragments that answer frequent queries with no or minimal clipping offset. With finite segmentations, we have the instrument to build fast data processing modules for XML data. As discussed in Section 3.6, the use of ID nodes in coexisting pattern fragments with one-to-one relations turn out as no disadvantage, since they are transferred and stored redundancy free, if accessed in common. The additional IDs introduced to manage union-, intersection- or difference-joining of pattern fragments are an acceptable overhead compared to the achievable savings, e.g. with finite segmentation caching.

4. RELATED WORK

Tree patterns are well known in the context of XML data processing and especially used to improve query response times. To search frequent XML tree patterns in XML documents[8] is a widely adapted technique and is used for various applications, ranging from indexing optimal access paths[1-3] to the formulation of various classes of XML queries[4,5]. We follow these approaches, as we use frequent access tree patterns to achieve optimization goals. With the latter two approaches, we have in common to use tree patterns to specify subclasses of queries. Tree patterns represent the tree-structure of XML query languages like XPath[6] or XQuery[7], and are treated separately from regular expressions also found in such queries. In the context of querying and maintaining incomplete data, Abiteboul[20] shows a solution for XML data. The presented incomplete data trees are similar to our colored schema graphs, in that they use conditions on the elements' data values and are based on DTDs. Different to our approach, their incomplete tree is used for fast calculation of missing parts in a single client.

In comparison to all these approaches, we use tree patterns to identify sets of pattern fragments and include some information also found in regular expressions and handle them in a search tree manner. A caching strategy based on frequently accessed tree patterns is introduced in Yang[9]. We extend the approach of classical patterns presented in Yang[9] to ST-patterns including predicate filters, which enable us to express finer XML granularity. Our approach also differs in that we support cooperative caching by sets of pair-wise disjointed patterns.

A different approach for XML caching is to check whether cached data can contribute to a new request by testing the intersection of cache entries and an XPath query[16] and thereafter compute difference fragments as partial results[19]. Such tests are known to be NP-hard for XPath expressions[13, 15] and difference computations are known to be resource consuming. In comparison, our approach focuses on efficient computation and thereby requires only minimal resource consumption.

5. SUMMARY AND CONCLUSIONS

We expect finite pattern segmentation to be a solution for splitting a huge XML document into handy atomic units to support fast data processing based on simple and fast intersection and containment decisions, e.g. in the area of cashing, replication or query processing. The drawback of using normalized data units is a clipping offset caused by answering a request by a slightly bigger superset. This is acceptable, since frequent requests can be

answered with minimal or no clipping offset based on a well adjusted preprocessed segmentation. Especially in the area of mobile data processing, it is important to minimize communication costs and preserve the mobile client's resources. Besides communication resources, we keep shared CPU resources to a minimum because costly intersection or containment tests are reduced to simple lookups. In the context of collaborative data processing, it is important that participating clients interact and interchange data based on a set of predefined data units. Otherwise, advantages of collaboration will be consumed by adjusting and comparing (slightly) different data objects.

Currently we implement a mobile peer-to-peer approach which will use finite segmentation caching for any data exchange. In our further research, we address the challenge of segmentation adoption and update propagation for the overall system. Adapting ST-patterns towards dependent patterns, not containing the decision criteria, and distributed query processing[17, 18] based on ST-patterns, seem to be further promising steps. We use these search tree patterns (ST-patterns) to model virtual schema expansion which we intend to discuss in detail in future publications. Our solution is especially tailored to adapt to context switches in query behavior supporting, e.g., a fine granularity in hot spot areas.

REFERENCES

1. Chin-Wan Chung, Jun-Ki Min, Kyuseok Shim: APEX: an adaptive path index for XML data. SIGMOD Conference 2002: 121-132 [DBLP:conf/sigmod/ChungMS02]
2. Torsten Grust: Accelerating XPath location steps. SIGMOD Conference 2002: 109-120
3. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, Henry F. Korth: Covering indexes for branching path queries. SIGMOD Conference 2002: 133-144
4. Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, Divesh Srivastava: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002
5. Nicolas Bruno, Nick Koudas, Divesh Srivastava: Holistic twig joins: optimal XML pattern matching. SIGMOD Conference 2002: 310-321
6. J. Clark and S. DeRose. XML Path Language (XPath) version 1.0 W3C recommendation, 1999.
7. D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. XQuery: A Query Language for XML W3C working draft, 2001.
8. L. H. Yang, M. L. Lee, W. Hsu. Mining Frequent Query Patterns in XML. 8th Int. Conference on Database Systems for Advanced Applications (DASFAA), 2003.
9. Liang Huai Yang, Mong-Li Lee, Wynne Hsu: Efficient Mining of XML Query Patterns for Caching. VLDB 2003: 69-80
10. Stefan Böttcher, Adelhard Türling: XML Fragment Caching for Small Mobile Internet Devices. Web, Web-Services, and Database Systems 2002: 268-279
11. Franky Lam, Nicole Lam, Raymond K. Wong: Efficient synchronization for mobile XML data. CIKM 2002: 153-160

12. Douglas B. Terry, Venugopalan Ramasubramanian: Caching XML Web Services for Mobility. ACM Queue 1(1): (2003)
13. Jan Hidders: Satisfiability of XPath Expressions. DBPL 2003: 21-36
14. Georg Gottlob, Christoph Koch, Reinhard Pichler: XPath Query Evaluation: Improving Time and Space Efficiency. ICDE 2003: 379-390
15. Georg Gottlob, Christoph Koch, Reinhard Pichler: The complexity of XPath query evaluation. PODS 2003: 179-190
16. S. Böttcher: Testing Intersection of XPath Expressions under DTDs. International Database Engineering & Applications Symposium. Coimbra, Portugal, July 2004.
17. Yanlei Diao, Michael J. Franklin: Query Processing for High-Volume XML Message Brokering. VLDB 2003: 261-272
18. Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis Viglas, Yuan Wang, Jeffrey F. Naughton, David J. DeWitt: Mixed Mode XML Query Processing. VLDB 2003: 225-236
19. S. Böttcher, Adelhard Türling: Caching XML Data for Mobile Web Clients. International Conference on Internet Computing IC'04, Las Vegas, USA, Juni 2004.
20. Serge Abiteboul, Luc Segoufin, Victor Vianu: Representing and Querying XML with Incomplete Information. PODS 2001 [DBLP:conf/pods/AbiteboulSV01]