A Template-Based Approach to Summarize XML Collections

Gudrun Fischer, Igor Jacy Lino Campista
University of Duisburg–Essen
D-47057, Duisburg, Germany
{fischer,campista}@is.informatik.uni-duisburg.de

Abstract

Existing summarization approaches for XML concentrate on extracting common structure and compressing the data, to optimize storage and speed up queries. Neither compression, nor structure extraction suffices for advanced, content-based summarization tasks. We present a set of tools for semi-automatic summarization of XML collections, where the user can specify semantically relevant features for an XML collection in a template, and define rules for summarization. The system assists the user in generating one or several such templates, selects applicable templates for a given collection, and applies them for automatic summarization. In experiments on the INEX collection (among others), we investigate the merits and limitations of our approach.

1 Introduction

Summarization is the task of creating a shorter representation of a given document or dataset, containing those characteristics of the original which are relevant for a specific user and application context. Previous approaches like [Liefke and Suciu, 2000] concentrate on extracting common structure and compressing the data, in order to optimize data storage and exchange and, in some cases, query processing. Similarly, explicit XML summarization techniques, e.g. [Cannataro *et al.*, 2002a], aim at query optimization and precompute aggregation dimensions, concentrating on the data-centric view of XML.

For other applications, however, this functionality does not suffice. In the context of XML document clustering, for example, the cluster descriptions which have to be generated from the documents in the cluster should give the user a good overview of the content of the cluster, rather than of the structure alone. Similarly, when exploring a set of unknown XML collections, users could profit from concise and meaningful, content-based summaries which should still contain relevant structural characteristics of the collections as well. Therefore, as observed in [Cannataro *et al.*, 2002b], the summarization of XML documents is not only an auxiliary functionality for compression and querying, but a task in its own right.

In this paper, we present a template-based, semi-automatic approach where the user herself can specify the relevant parts and characteristics of XML documents, and how they should be summarized. This information is formulated in an XML template document which can be generated automatically or manually, or interactively using the *Template Designer*, one of the two tools which form our implementation.

Given a new XML collection and one or multiple templates, the *Automatic Summarizer* then assigns each document of the collection to those templates which match the document's structure and content. After the matching step, the Automatic Summarizer applies to each of the found subcollections the respective template, thus generating the desired output for the whole collection.

In comparison with [Cannataro *et al.*, 2002a], we follow the same assumption that the information need of the user will vary between collections and contexts of application, and thus no automatic method will be able to meet this need accurately. In contrast to their approach, however, we concentrate on the document-centric view of XML, and give the user a more thorough control over the summarization process by providing more, and dedicated datatype–specific summarization functionality. In addition, we facilitate template generation and the reuse of existing templates by providing a flexible tool for interactive template manipulation.

The rest of this paper is organized as follows. In section 2, we give an overview over previous XML summarization approaches. Section 3 then introduces our concept of template-based summarization, and in section 4, we describe the two tools implementing it. Section 5 contains experimental results for the INEX collection [Fuhr *et al.*, 2004]. In section 6, we close with an outlook on further research.

2 Related work

Existing approaches dealing with XML summarization can be distinguished by their motivation. XMill [Liefke and Suciu, 2000] uses summarization for efficient compression of XML collections, separating structure from data, and compressing or, as the case may be, summarizing the data according to its datatype. To define transformation rules, they use a simple, dedicated language.

Compression and synthesis for efficient storage and querying is also the main objective of SqueezeX [Cannataro *et al.*, 2002a]. There, the user first specifies the relevant dimensions of XML documents in a negotiation phase. According to these dimensions, the documents are then transformed into datacubes, with aggregated values in the selected dimensions, which can be queried efficiently. SqueezeX thus implements a strictly data-centric view.

Query processing is the target application of [Comai *et al.*, 2004]. Here, however, XML collections are summarized in order to allow for approximate answers from precomputed summaries. A fixed set of transformations

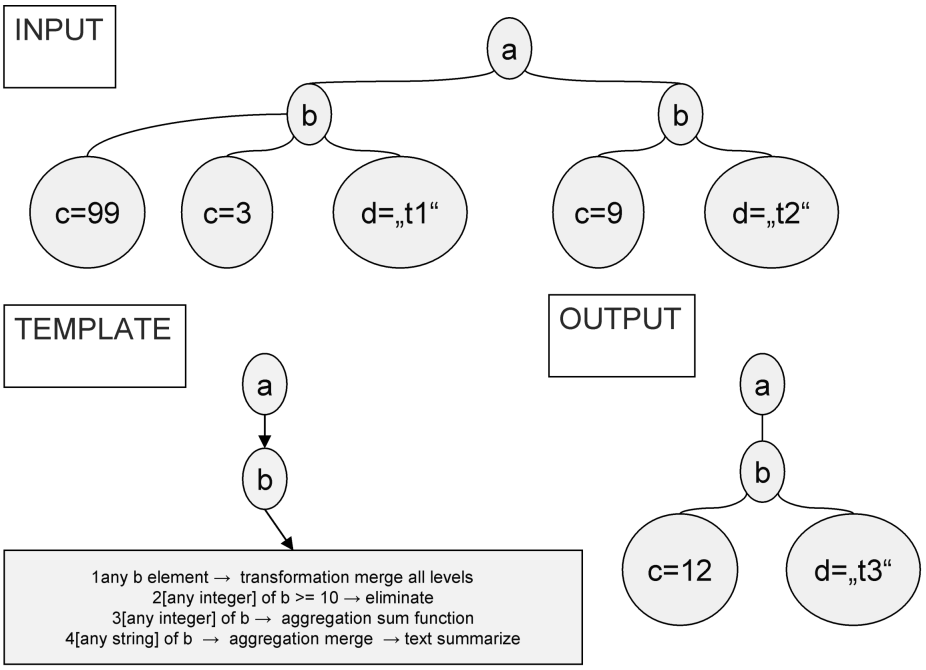

Figure 1: Summarization example

for this purpose is described in the XQuery language. XSKETCH [Polyzotis and Garofalakis, 2002c; 2002a; 2002b], on the other hand, creates statistical summaries of large XML data graphs, to estimate path expression selectivity.

Finally, [Lim *et al.*, 2002] build a tree representation of the input XML (*path tree*) for paths up to a certain length, delete low-frequency paths, and estimate the selectivity of longer paths. Their method allows an approximate representation of the original XML to fit into any amount of memory, and can therefore be seen as a summarization as well.

The above approaches either target XML summarization as an auxiliary functionality for another objective, e.g. compression or query optimization, or they are limited to a data-centric view of XML, or they do not allow user interaction. Our approach, in contrast, is semi-automatic, supporting user interaction, and follows a document-centric view.

3 Template-based summarization

In our template-based approach to XML summarization, the user describes the *elements* she is interested in by defining a pseudo-structure of the input XML (one or multiple documents) containing the relevant features. This pseudo-structure is called a *template* and is an XML document itself. Like the input XML, it can be seen as a tree, with nodes describing elements, and transformation and aggregation rules annotating the nodes. Relevant nodes in the input XML can be selected by a node description using the path, datatype, and/or tag name. For each node thus selected, the user can define further selection rules for descendants, and transformation rules for the content of the node itself, or aggregation rules for descendants.

Figure 1 shows a simple example, using schematic tree representations. Template documents, as well as the input and output of our approach, are specified in XML. In the input XML, the root `a` has two children with tag name `b`. Both `b` nodes have children with tag name `c` and content of type `integer`, as well as children with text content and tag name `d`. The user in this example is interested in a summary of the content of all the `b` elements, differentiating between `integer` and `text` nodes, while the path to the `b` elements is to be preserved unchanged. For content of type `integer`, only values below 10 are relevant.

The corresponding template document first specifies the nodes which should be processed to generate the output, i. e. `a` and `b` nodes, and their path structure. The `b` node is then annotated with a transformation rule, which merges all such `b` nodes into a single `b` element in the output (rule 1). Rules 2, 3, and 4 in the example treat the children of the new `b` node. Nodes with content of type `integer` and value greater or equal to 10 are eliminated (rule 2). `Integer` nodes with values less than 10 are aggregated by calculating the sum (rule 3), and nodes with text content are aggregated by merging and summarizing them with a text summarizer (rule 4). The resulting output XML contains the original `a` node, but only one `b` node with one `c` child and one `d` child, each summarizing several `c` and `d` nodes of the original.

Obviously, the order of processing is important for some rules. In the given example, rule 1 must be applied before all the other rules, so that rules 2, 3, and 4 will work on the merged set of children of all the `b` elements. Similarly, rule 2 has to be processed before rule 3, because only integers below 10 should be considered in the sum. Template trees in our approach are processed in a pre-order direction, i. e. for a given node, all its rules will be processed before any rule of any of its children. Thus, rule 1 in the example will be invoked before the other rules by default. For cases like rules 2 and 3, we allow rules to be grouped into a chain, where each rule in the chain will work on the output of its predecessor. In the example, rules 2, 3, and 4 all consecutively modify the set of child nodes of the single `b` resulting from rule 1. Thus, all three rules can be combined

into a rules chain[1]. The resulting order of processing in the example is shown in figure 1 by the numbering of rules. The rules chain itself is depicted in the screen shot in the next section (figure 2), which contains a more detailed representation of the template document from this example.

As such a template is a valuable result of the user's work and requires time and cognitive effort, ease of reuse is an important issue. Furthermore, for heterogeneous collections, it may be necessary to specify multiple templates, targeting different subcollections. Therefore, we inserted an additional step between the design, and the application of templates: *template matching*. The input to the matching step is a collection of XML documents, and a set of templates. Each input document is assigned to every template which fits its structure and content, i. e. which does not expect any node (described by tag name and/or type) which is not found in the input document. Thus, one input document can either be part of one or several subcollections, or it will be ignored because no applicable template was found. The output of the matching step is a set of matched subcollections, each paired with its corresponding template. These form the input to the *summarization step*, where each subcollection is summarized according to its template. In the following section, we describe the tools we developed to implement these concepts.

4 Prototype

Our implementation consists of two tools, the *Template Designer*, and the *Automatic Summarizer*.

4.1 Template Designer

Templates can be created, manipulated, and tested, using the Template Designer. Figure 2 shows the graphical user interface for the example in the previous section.

The screen is divided into three parts. The template document is shown in the center, either as XML, or graphically as a tree. The `a` and `b` nodes from the example in figure 1 are specified by `NodeDescriptors`. The node for `b` has two children: the `TransformationRule` for merging all `b` elements (rule 1 of the example), and a `NodeDescriptor` without tag name, which will select all children of the new (merged) `b` node. This generic node is annotated by a rules chain containing the remaining rules from the example. The first rule in the rules chain filters out all `integer` nodes with values greater or equal to 10. The second rule will aggregate all remaining `integer` values by calculating their sum. Finally, the third rule will aggregate all `string` nodes by applying a `SummarizationModuleRule`. Obviously, for this example, this third rule will take the text content of the respective nodes and simply merge it.

Templates can be created from scratch, or extracted by the Template Designer from existing examples. We implemented a simple template extraction strategy, which consecutively parses given examples and matches them with the template under construction. Whenever the extraction algorithm encounters a non-matching node in an example, it adds a corresponding `NodeDescriptor` to the template. For each `NodeDescriptor`, the number of example documents containing a matching node is determined, so that the user can prune rare nodes later, if necessary.

Regardless of how a template was created, the user can edit it further in the Template Designer by e.g. adapting it to additional example input documents. Thus, it is easy to reuse existing templates for new XML collections. The left part of the Template Designer shows the current example input XML, either as a tree, or as raw XML. The user can modify the example input, or select a new example input at any time, and test the template on the new document. On the right part of the screen, she can then preview the output which would be generated by applying the current template to the current example input.

4.2 Automatic Summarizer

The Automatic Summarizer consists of two components, implementing the two remaining automatic steps, i. e. the template matching and the summarization itself. The matcher component reads XML documents from a directory and templates from another directory and creates the matched subcollections (i. e. template-subcollection pairs). The summarizer then processes each matched subcollection, and writes the resulting new XML documents to a specified output directory. The summarization of a matched subcollection entails parsing each input document, and in doing so, comparing each element of the input with `NodeDescriptors` in the template, and, if a `NodeDescriptor` matches an element, transforming the element according to the specified rules, and then processing its children. After transforming an XML document or a collection of XML documents in this fashion, the result is saved to the specified output directory. The user can monitor and check the process in a log file.

4.3 Implementation

The prototype implementation in Java has been designed for extensibility. Therefore, most of its components, e.g. the underlying XML parser, the template extraction strategy, and the matching algorithm, are loaded at runtime, while the framework uses abstract classes or interfaces. Currently, we have two parser implementations, based on DOM, and on STaX[2], respectively.

Rules and datatypes are the most basic components used in both the Template Designer, and the Automatic Summarizer. The characteristics of the individual datatypes (e.g. `integer` and `string`) are defined in dedicated XML schemas and implemented in a corresponding Java class hierarchy. Additional datatypes can be added in a straightforward way by defining the characteristics and deriving an implementation from the `AbstractDatatype` class. Similarly, the available rules are implemented in a corresponding hierarchy of Java classes which can be extended to add further rules as required.

To leverage the power of other existing summarizer implementations, these can be plugged into the system by defining a `SummarizationModuleRule`, which invokes the external code that has to be encapsulated in a `SummarizationModule`. In our example, we used the Classifier4J summarization module[3] for text summarization.

5 Evaluation

In this section, we present experimental results and discuss the merits and limitations of our tools. The experiments were conducted on four different test collections:

[1] While it is not important that rule 4 be processed *after* rules 2 and 3, it does not influence the result either if it *is* processed in this order.

[2] E.g. http://woodstox.codehaus.org

[3] http://classifier4j.sourceforge.net/

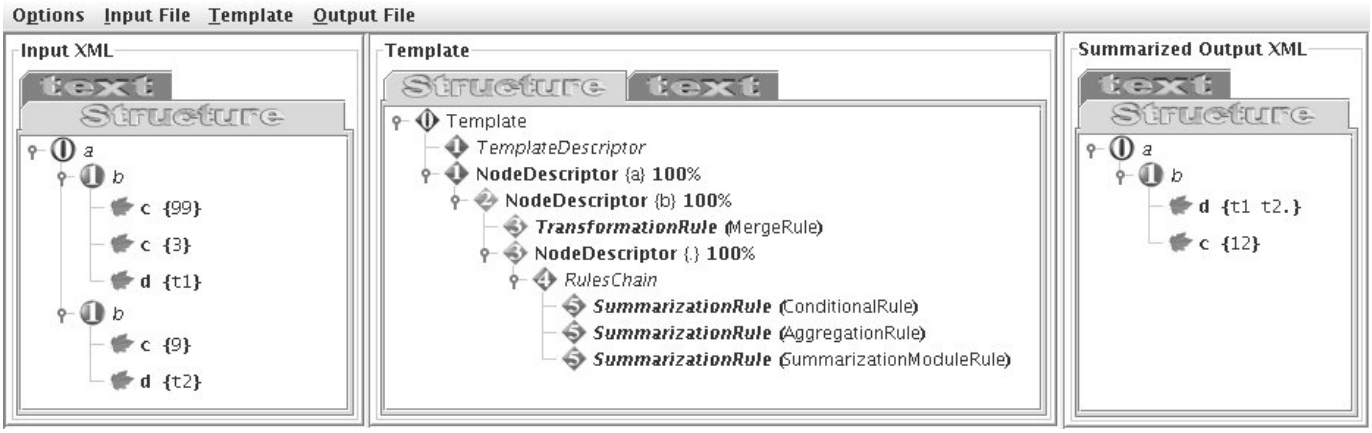

Figure 2: Template Designer

- **INEX** [Fuhr *et al.*, 2004] is a collection of XML full texts with an average of 1,532 nodes per article. For the matching experiments, we used only 45 arbitrarily selected documents with an average file size of 41,249 bytes. For the scalability test (at the end of this section), we took 10,000 documents from INEX 2001.
- **MathPreprints** are 1,002 bibliographical metadata records in XML-coded Dublin Core with an average file size of 2,559 bytes, which we extracted from the corresponding Open Archive[4].
- **CompuScience** [5] is a bibliographic database covering literature in the field of Computer Science and Computer Technology. Our input XML consisted of 11 XML-coded bibliographies, each describing multiple articles and including abstracts. The average file size was 29,885,175 bytes.
- **Shakespeare** is a collection of 37 plays (full texts) coded in XML, with an average file size of 213,887 bytes.

We ran all experiments for both parser implementations and achieved consistently better results for STaX. Because of space limitations, we will therefore concentrate on the most interesting STaX results and refer to our website for a more detailed and comprehensive list of results.

5.1 Template generation

As mentioned in section 4, templates can be extracted from a set of example documents. We therefore investigated the time for extracting a template, varying the number and the source of the examples (and thereby the size as well). Comparing the results for the different test collections, we made the following observations:

- The size of the generated template and the time needed to reach a maximal template depends on the structural heterogeneity of the collection. For the MathPreprints collection, the template was complete after 15 examples, while for INEX, it was still growing linearly after 45 examples.
- The time to generate the template depends linearly on the number of files to be read, and their sizes. This was to be expected, as the Template Designer has to read each example. However, it is encouraging that the growth is only linear.

For 45 INEX documents with a total size of 1,812 K, just 282 milliseconds were necessary. For 10 CompuScience documents (288,723 K), the template generation took 3.9 seconds. Thus, the time for extracting a template from a moderate sample is certainly affordable. As the template size converges after a certain number of samples, we recommend to extract a template from a small sample first (e.g. 20 documents), and consecutively try larger samples, as long as the resulting template does not match the whole collection sufficiently well.

5.2 Matching

Matching is the process of assigning each input document to all templates which match its structure. Thus, it requires comparing each XML document to each candidate template. Figure 3 shows the time required to match a growing number of input files stemming from 3 different collections (INEX, Shakespeare, MathPreprints), in relation to a growing number of templates[6].

In these tests, the time required for matching depends linearly on the number of templates and on the number of input files. Interestingly enough, however, we saw in further experiments that the time does not depend directly on the sizes of the input files. This is intuitive, as matching requires detecting structural compliance, and an input file, regardless how large it is, can be discarded as soon as it does not match a structural condition. Thus, most input files were not read completely for the non-matching templates, and their size could therefore not influence the time as much as the number of files and templates.

For matching 41 input files with 5 templates, the Automatic Summarizer already took 15 seconds. If the user wants to wait for the result of the matching phase (e.g. to select the most appropriate template for a collection), matching a whole collection (probably more than 41 documents) to a large number of templates may require too much time. For these cases, we therefore recommend an interactive approach instead, where for an unknown collection, only a small sample is matched to all candidate templates, a larger sample is matched to the most successful templates of the first iteration, etc. , until only one or two templates are left. These can then be used for the whole collection.

[4] http://www.mathpreprints.com/math/OAI/

[5] http://www.zblmath.fiz-karlsruhe.de/cs/index.html

[6] For the STaX parser implementation, the time required does not change with a growing number of source collections.

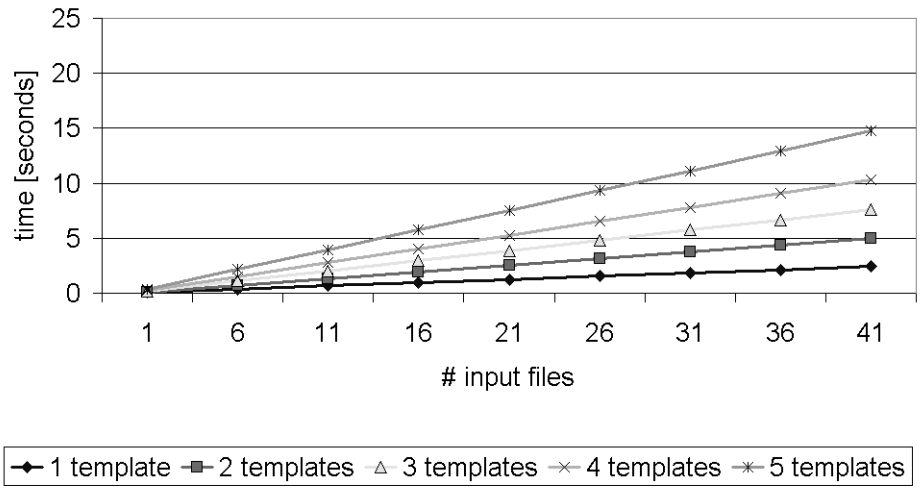

Figure 3: Template matching

5.3 Summarization

The summarization step takes a matched collection (a collection of XML documents, paired with a template) and summarizes it according to the rules in the template. Among other experiments, we compared the following two templates, applied to a growing number of INEX documents:

- **T2** uses a dedicated summarization module implementation which extracts bibliographic fields from INEX documents and transforms them into a format suitable for the Daffodil Digital Library[7].
- **T3** removes infrequent structure and summarizes text using the Classifier4J summarizer module.

Figure 4 shows the total output size, the time spent, and the reduction ratio for 1,000, 2,000, 3,000, 5,000, and 10,000[8] INEX documents. T3 produces a larger output than T2 (and thus, a lower reduction ratio), which is noticeable mainly for the larger collection sizes. Both templates achieve very good reduction ratios (between 98.42 and 98.93 %).

The maximal duration in minutes (for template T2) is 41, for 10,000 documents. Therefore, while this is not fast enough for online summarization (e.g. during online clustering of XML documents), the time is affordable in offline summarization scenarios where e.g. a new collection has been harvested from the Deep Web, and the user wants to gain an overview at leisure, or where e.g. an institution wants to give a content-based description of an XML collection it provides.

6 Conclusion and outlook

Summarization of XML documents is not only a valuable functionality to support query processing and other applications, but also a task in its own right. We have developed a template-based approach to XML summarization, allowing the user to specify which elements of XML documents are relevant, and how they should be summarized. Template creation and reuse is supported by an interactive graphical tool, the Template Designer, where the user can modify templates and/or example input documents and preview the effect immediately. Templates can be created manually from scratch, or they can be automatically extracted from a set of example documents, to be fine-tuned later. Given an existing set of templates, and a collection to be summarized, the Automatic Summarizer tool first creates subcollections by assigning each input document to those templates which match its structure, and then summarizes the resulting subcollections automatically, using their assigned templates. Experiments on the INEX collection (among others) show the applicability of our approach for fast offline summarization, although the effort in time is yet too high for online applications which would require summarization on the fly.

Although we have conducted a considerable number of experiments already, only some of which could be presented here, there are still some open questions. In particular, for the template generation and matching functionalities, we need to run large-scale experiments to determine their maximal capacities in terms of input size, if they have such a limit, regardless of processing time. Furthermore, we would like to compare our results with the computation times and compression ratios in previous work. This will entail additional experiments on their respective test collections.

To ensure the usability of our tools, we will also need to conduct user experiments, first to improve the graphical user interface, and then to assess the user satisfaction. After these preliminary steps to make our set of tools usable in a real-life working environment, the most interesting question will be if our template-based set of tools will really lead to a lesser (i. e. improved) workload for knowledge workers and other users.

[7] http://www.daffodil.de

[8] The slightly smaller numbers of files in the figure are the number of input files which were not rejected as non-compliant to one of the two templates.

References

[Cannataro *et al.*, 2002a] Mario Cannataro, Carmela Comito, and Andrea Pugliese. SqueezeX: Synthesis and compression of xml data. In *ITCC*, pages 326–331, 2002.

[Cannataro *et al.*, 2002b] Mario Cannataro, Antonella Guzzo, and Andrea Pugliese. Knowledge management and XML: derivation of synthetic views over semi-structured data. *SIGAPP Appl. Comput. Rev.*, 10(1):33–36, 2002.

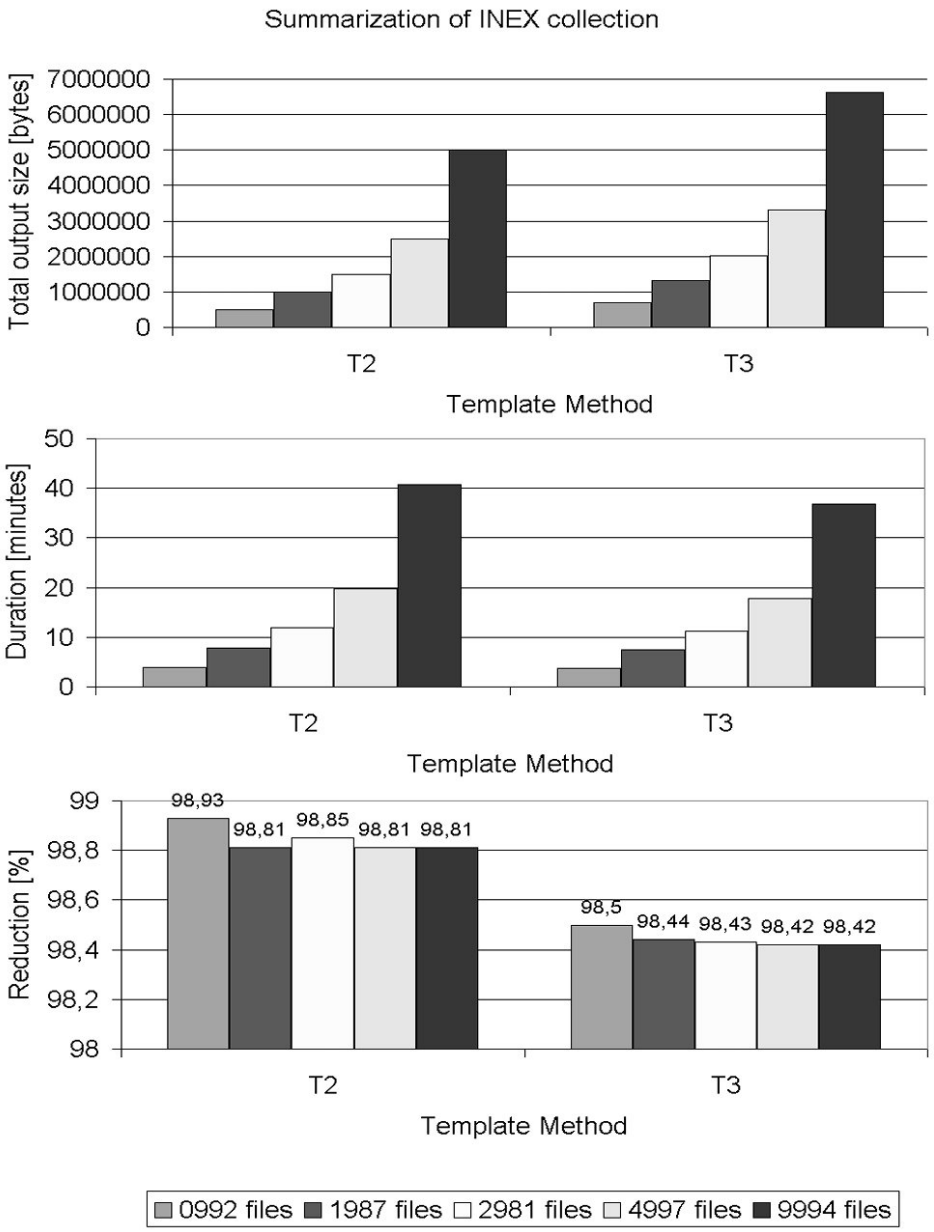

Figure 4: Summarizing INEX

[Comai *et al.*, 2004] Sara Comai, Stefania Marrara, and Letizia Tanca. XML document summarization: Using XQuery for synopsis creation. In *DEXA Workshops*, pages 928–932, 2004.

[Fuhr *et al.*, 2004] Norbert Fuhr, Saadia Malik, and Mounia Lalmas. Overview of the INitiative for the Evaluation of XML retrieval (INEX) 2003. In Norbert Fuhr, Mounia Lalmas, and Saadia Malik, editors, *INitiative for the Evaluation of XML Retrieval (INEX). Proceedings of the Second INEX Workshop. Dagstuhl, Germany, December 15–17, 2003*, pages 1–11, March 2004.

[Liefke and Suciu, 2000] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD. International Conference on Management of Data*, pages 153–164, New York, June 2000. ACM Special Interest Group on Management of Data, ACM.

[Lim *et al.*, 2002] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: an on-line selftuning markov histogram for XML path selectivity estimation. In *Proceedings of 28th International Conference on Very Large Data Bases*. Morgan Kaufmann, 2002.

[Polyzotis and Garofalakis, 2002a] Neoklis Polyzotis and Minos N. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD Conference*, 2002.

[Polyzotis and Garofalakis, 2002b] Neoklis Polyzotis and Minos N. Garofalakis. Structure and value synopses for XML data graphs, 2002.

[Polyzotis and Garofalakis, 2002c] Neoklis Polyzotis and Minos N. Garofalakis. XSketch synopses for XML. In *Proceedings of the 1st Hellenic Data Management Symposium, HDMS'02*, Athens, Hellas, July 2002.