Supporting Efficient Query Processing on Compressed XML Files

Yongjing Lin †, Youtao Zhang †, Quanzhong Li ‡, Jun Yang ⋆

† Computer Science Department
University of Texas at Dallas
Richardson, TX 75083

‡ IBM Almaden Research Center
San Jose, CA 95120

⋆ Computer Science and Engineering Department
University of California at Riverside
Riverside, CA 92507

ABSTRACT

XML has been widely accepted as the de facto format for data representation and exchange. However, it is also known for the excessive information redundancy in its representation. While various compression schemes have been proposed and some of them can support query processing over compressed files, it is usually inevitable to perform partial (or full) data decompression which is expensive and in some cases may dominate the query processing time.

In this paper, we propose a new XML compression scheme based on the Sequitur compression algorithm. By organizing the compression result as a set of context free grammar rules, the scheme supports efficient processing of XPath queries without decompression. The experimental results show that this scheme achieves comparable compression ratio as gzip while its query processing time is among the best of existing algorithms.

Categories and Subject Descriptors

H.2.1 [Database Management] Logical Design
E.4 [Coding and Information Theory] Data Compression

Keywords

XML, Query processing, Data compression

1. Introduction

Over the years the self-descriptive eXtensible Markup Language (XML) has been widely accepted as the de facto format to exchange semi-structured data among different systems. However it is also known for the excessive verbosity in its representation. The drawbacks, such as consuming more storage, transmission bandwidth, and computation power, can be serious in various scenarios, in particular, in the resource constrained embedded and mobile systems.

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

XML compression has been recognized as an effective approach for solving the above problem, and different compression schemes have been proposed [5, 11, 6, 1, 3, 2, 9]. Targeting at maximizing size compression, XMill [5] was proposed to separate a XML file into *containers* over which the gzip compressor (or customizable compressors) are applied to achieve good compression ratios. However, XMill does not support direct query processing. The compressed result has to be decompressed before query processing. Clearly this is not desirable when the XML file is large.

To support XML queries without full decompression, XGrind [11] adopted a homomorphic compression scheme which preserves the structure of original XML data. While queries can be evaluated over the compressed file, the compression ratio is usually worse than that of XMill or Gzip. A *reverse arithmetic encoding scheme* was recently proposed in XPRESS [6]. It encodes the label paths of XML data and applies diverse encoding methods depending on the types of data values [6]. XPRESS needs to preprocess the entire XML document which incurs large overhead when the XML data is undergoing active changes. Some other proposed compression schemes achieve fast query processing through partial decompression [3, 1]. For example, XQzip integrates a structure index tree (SIT) in the compressed XML file. Tags and data values are divided into blocks and then compressed using gzip. With the help of SIT, the query processing engine can select a subset of data blocks to decompress and process. It achieves better performance than previous schemes e.g. XGrind. In XQzip, the decompression cost usually dominates the query processing. With in-memory caching of decompressed results, the cost may be amortized by consecutive queries. Unfortunately, this is sensitive to the size of available memory for storing decompressed data, the type of queries and also the similarity of data to be processed by consecutive queries. In the worst case, the decompression cost is still visible to each query.

In this paper, we proposed a new XML compression scheme – XSeq which is adapted from the Sequitur algorithm [8], an existing compression algorithm for compressing text strings. Similar to that in XMill, XSeq first processes an input XML file and separates its XML tokens into containers each of which is then compressed using Sequitur. The compressed result also contains their necessary indices with the help of which XPath queries [12] can be efficiently evaluated. In summary, we made the following contributions.

- XSeq processes queries directly over the compressed file without full or partial decompression. By avoiding the decompression cost, the processing performance is independent of

the distribution of query results in the file and also the data correlation of consecutive queries.

- XSeq avoids the sequential scan of irrelevant compressed data and only processes data values that are to be matched by the given query. This feature is beneficial for queries that process scattered data values in the XML file since only a small portion of the file is touched and processed.
- We implemented and evaluated XSeq over a set of benchmark data sets. The results show that XSeq achieves comparable compression ratio as gzip while the query performance is among the best. In addition, the query performance is independent of the distribution of data values in the file.

The rest of the paper is organized as follows. The Sequitur algorithm is introduced in section 2. We discuss XSeq and its support for query processing in section 3. Section 4 presents the experimental results. We conclude the paper in section 5.

2. The Sequitur Algorithm

The Sequitur compression algorithm [8] is a linear-time online algorithm that forms a context-free grammar for a given string input. Starting with a rule with the non-terminal symbol S at the left hand side, the algorithm continuously fetches symbols from the input and appends them to the right hand side of the starting rule. Duplicate subsequences are checked during processing and a new production rule is generated for each repeated subsequence if this is not done before. After the new rule is generated, the repeated subsequence is replaced by the left hand side non-terminal symbol of the rule. In Figure 1, a new rule X is generated because "abc" appears more than once. The compression result is a set of the grammar rules. Alternatively, the result can be represented by a directed acyclic graph (DAG) (Figure 1).

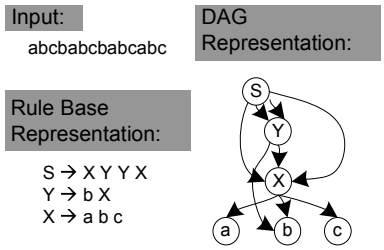

Figure 1: Sequitur Compression.

Two properties are ensured in the compressed grammar representation [8].

- *Diagram uniqueness: no pair of adjacent symbols appears more than once in the grammar.* If, by adding a new input symbol, two adjacent symbols appear more than once in the grammar, a new produce rule will be created to replace both appearances. In the example, after we add the fifth input symbol, we have the starting rule as follows.

  $$S \rightarrow abcba$$

  after adding the sixth "b", we get "abcbab". Since the subsequence "ab" appears twice, it has to be replaced. We then have:

  $$S \rightarrow ZcbZ$$
  $$Z \rightarrow ab$$

- *Rule utility: every rule is used more than once.* The number of times that a rule is used can decrease during the processing. If this number is reduced to one, the rule will be discarded. For example, after adding the seventh symbol in Figure 1, we have

  $$S \rightarrow XbX$$
  $$X \rightarrow Zc$$
  $$Z \rightarrow ab$$

  Since rule Z now is used only once, it gets deleted and its appearance in rule X is replaced by "bc". That is,

  $$S \rightarrow XbX$$
  $$X \rightarrow abc$$

For string inputs, the Sequitur algorithm has the ability to achieve comparable or better compression ratios than LZ family compression algorithms [8]. This is achieved from a distinct property of Sequitur: the compression result is organized as a set of *context-free* grammar rules. Each non-terminal (NT) symbol can be expanded only to one string of terminal symbols and this string is independent of the location where the NT symbol appears. On the contrary, the same value may represent different substrings in the LZ family algorithms [13].

The context free property is also important for supporting efficient query processing over the compressed file. For example, query processing requires the traversal of the DAG for desired values. If we meet a visited non-terminal node from a different path, we can expect the exactly same result if that node is processed again. We therefore can reuse the result from previous visit and skip the scan of the node and its subtrees. Suppose we are looking for "ab" in Figure 1, starting from the start symbol S and then X, we find "ab" in X and mark X after scanning it. When following a different path "Y→bX" to access X later on, we reuse the fact that "ab" is contained in X and save the time/space of further processing.

3. XSeq: Direct Query Processing on Compressed XML Files

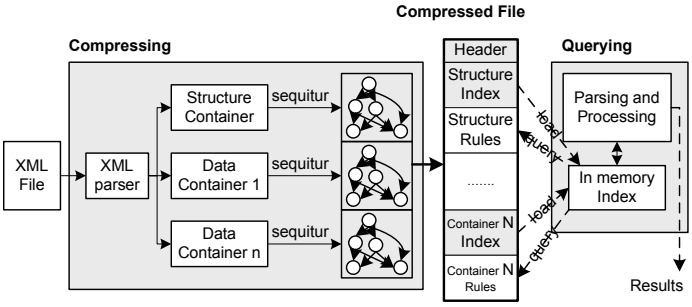

Figure 2: Scheme Overview.

In this section, we first discuss the design motivation of XSeq. We then present the organization of XSeq and discuss how XPath queries are processed with the compressed format.

3.1. Motivations

To achieve better compression ratios, our first design decision is to separate tags and data values into different containers, which is similar to that in XMill [5]. By grouping XML tokens that are of the same type, separation based compression algorithms increase the data similarity in each container and achieve better data compression than others. For example, a XML file is usually compressed to 20% of its original size using XMill. With XGrind, the number is about 50%. The compressed file size of XGrind almost doubles the result of XMill.

However, separating a XML file has negative impacts on query processing. Since typical XPath queries require the match for both tags and concrete data values, the separation complicates the search for the correct value in different containers especially if the container is compressed. It is expensive to decompress the whole container and search only for few values. While recently XQzip can speed up the processing through specially designed indices and also through partial decompression, the decompression cost is still quite significant. In cases where the data values to be processed are scattered in the XML file, it is hard to amortize the decompression cost.

In this paper we took a similar approach as XQzip but tried to avoid decompression cost in the query processing. This is achieved through the context free property of Sequitur algorithm. As the data is stored on leaf nodes of the DAG graph, we only need to go from the root and scan nodes on the path to these leaves. Comparison can be done on these nodes directly without decompression and irrelevant nodes can be skipped without further processing. Decompression is needed only when we want the decompressed representation of query results.

To correlate the data values stored in different containers, we built a structural index through which each data value can be quickly located in the container without decompression. We kept the indices within the compressed file and had them loaded into the memory before processing the rule contents.

3.2. Compressing XML Files with Sequitur

An overview of XSeq is shown in Figure 2. The original input XML file is first parsed using SAX parser. Similar to that in XMill, XML tokens are assigned to different containers with the tags and attributes sent to the structure container. Each container is then compressed using Sequitur algorithm producing a set of context free grammar rules. The compressed file contains both the rules for these containers and the necessary indices.

Three types of indices are stored in the compressed file: the file header, the index for the structure container, and the index for each data value container. They are defined in Figure 3. The header index contains two entities: one is a list of pointers each of which points to the entrance of one container in the file. The other is a mapping table between each tag and its assigned tag code. After assigned the tag code, each tag appearance is replaced by this code in the structure container (container number 0) e.g. T2 is assigned to "title" and therefore all appearances of "title" will be replaced by T2.

The second type is used for the structure container while the third type is used for each of data value containers. Their formats are briefly described as follows (Figure 3). A "rule_count" field is used to specify the number of rules in its associated container. Next, there are an array of entries that specify the summary information of each rule in the container. The summary has a "rhs_count" field which specifies the number of right hand side rule items. The rest of the fields in structural index specifies the tags appear in the expanded string of each rule (non-terminal symbol), together with the frequency of each appeared tag. These indices are used for fast query processing. We will discuss their usage in the next section. The summary of each rule is generated when the compressed file is created. There is a "value_count" field in the index for each data value container. It specifies how many starting positions of XML data values in that rule. It differs from "rhs_count" in that an XML data value may be a text string and thus consists of multiple RHS items. For the simplicity of discussion, we skip it and assume all values are of the unit length.

Instead of the actual address in the file, we store the number of RHS items of each rule. For example, we store "2" for rule R1 as it has 2 RHS items. The compressed file stores the grammar rules sequentially in the file and thus the starting position of a production rule can be located by counting the number of preceding RHS items. The actual address is maintained in the memory and calculated from this index. When we are to access a new container, we load the rule index into the memory and generate the actual address for each rule.

3.3. Processing Path Queries in Compressed XML Files

In this section we discuss the data correlation of different containers and show how to process XPath queries in XSeq.

As we discussed, a typical XPath query contains the matching requests for both tags/attributes and data values, for example "/dblp/inproceedings[year=2003]". We need to compare tags "dblp" and "inproceedings", and also go to the year container and compare the corresponding value to see if it is 2003. When tags and data values are separated into different containers, we need to find the exact item in a container to match. For example, we may have a list of year values in the year container and there are multiple paths reaching "2003", each of which indicates a different location. The challenge is that after skipping a series of tags in structure matching, e.g. skipping a "injournal" node and all its associated year values, we still need to accurately find the right year value in the year container.

In XSeq we use position counters to correlate the corresponding positions in different compressed containers. For example, if tag "year" has already appeared 99 times, and "$< year > 2003 < /year >$" is the 100th appearance. We can then go to the year container and directly compare the 100th value and skip all other values. By doing this we need two consecutive processing steps. The first step decides to match the 100th value while the second step jumps to the 100th value in the compressed year container without decompression.

3.3.1 *Correlating Values in Different Compressed Containers*

The first step is done with the help of structure index. The structure index stores a small array for each grammar rule of the compressed structure container. It basically summarizes the expanded string of its associated non-terminal symbol. In Figure 3, the structure index indicates that R1 has 3 different tags i.e. T2, T3, T4. T3 has two appearances while T2 and T4 have one appearance each. As the expanded string of R1 is "T2 T3 C1 / T3 C1 / T4 C2 / /" ("/" indicates the end of a tag), this small array exactly catch the summary of R1.

With the help of the structural index, we can skip scanning a production rule if the index indicates no matching data in the expanded rule. For example, if we want to search for T4 in rule R1, we skip two R2s as the summary of R2 indicates T4 does not appear in R2. By skipping R2, it means we do not need to load the corresponding content of R2 and all the data (rules) under R2. On the other hand, if we want to locate T4 in C2, we should match the first value as no T2 has been skipped.

3.3.2 *Counter Based Matching in Compressed Containers*

Given a position value, the second step is to quickly locate the value in a data container. Here we assume each data value has one unit of length. By counting the number of RHS items, we can jump to the right place to compare. For example, suppose we are to fetch the 100th value in a container and we have "ABC" where A and B contain 60 and 20 values respectively. We can skip A and B, go

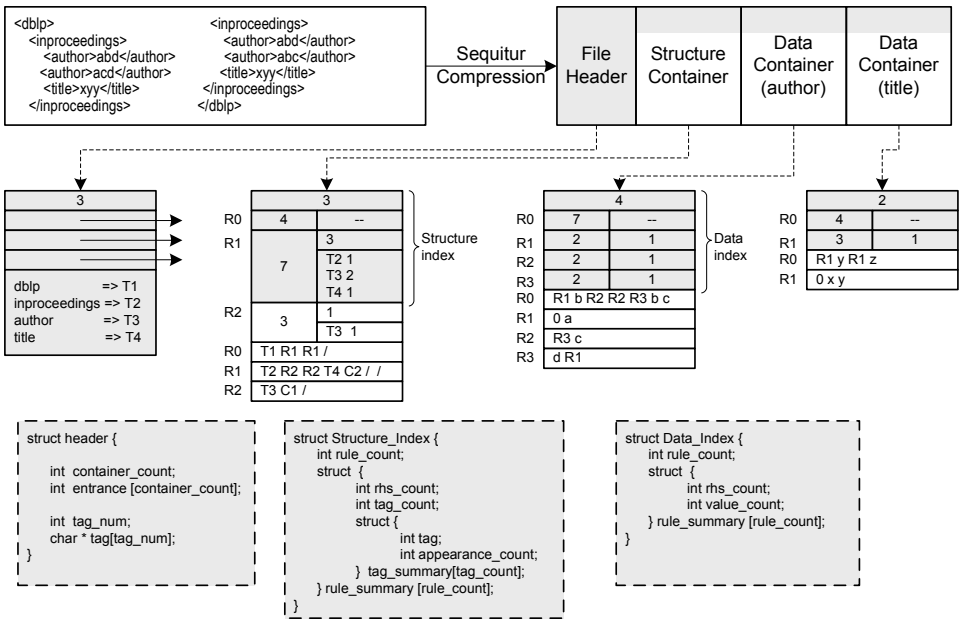

Figure 3: An XML example.

directly to C, and search for the 20th value (100-60-20=20) in rule "C→ ...".

For data values that are not of unit length, we use "value_count" to decide if a non-terminal symbol and its associated data should be skipped. The "rhs_count" here is used to locate the entrance of a rule. As we may load only a small subset of rules in a container, the "rhs_count" and "value_count" provide the power to randomly locate the entrance of a rule and also its length.

4. Experimental Results

4.1. Settings

Benchmark	Size(MB)	Depth	Tags
DBLP	133.9	6	43
XMark	116.5	11	85
Shakespeare	7.6	6	23
SwissProt	114.8	5	101

Figure 4: XML Benchmarks.

To evaluate the performance of XSeq, we implemented the proposed compression algorithm and compared the results to XMill, XGrind, and XQzip. The experiments were performed on a Redhat Linux 7.0 with Pentium IV 2.0GHz processor and 256 MBytes main memory.

We collected a set of four representative benchmarks whose characteristics are summarized in Figure 4. DBLP contains bibliographic information on major computer science journals and proceedings. XMark contains XML data generated from XMark project [7]. Sharespeare [10] is the collection of plays of Shakespeare. SwissProt [4] is a protein sequence database. Three of them are of large size (over 100MB) while XMark and Swissprot have more different tags (around 90).

4.2. Compression Ratio

The compression ratio is defined as

$$Compression\ Ratio = 1 - \frac{size\ of\ compressed\ XML\ data}{size\ of\ original\ XML\ data}$$

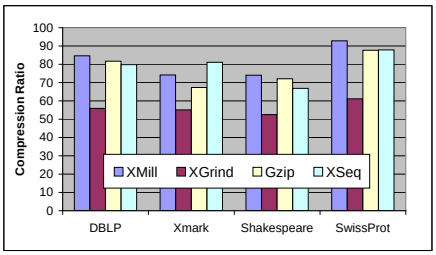

Figure 5: Compression Ratio.

The compression ratios of different benchmarks are summarized in Figure 5. The compressed XSeq file contains the storage for both the rule sets and the indices.

On average, we achieved 81.4% compression ratio for XMill, 56.2% for XGrind, 77.2% for Gzip and 78.2% for XSeq. From the results, we found while XMill has the best compression ratios, XSeq performs comparably well. The comparession ratio is about the same as that using gzip. Three benchmarks have comparable results as Gzip. XMark achieves better compression ratio using XSeq mainly due to its regularity of data values. As reported in [1], XQzip has comparable compression ratios as XMill when the indices are not included and is usually worse than gzip when having the indices included.

4.3. Compression Time

Figure 7 summarizes the compression time results. XSeq has the worst compression time. In the worst case (DBLP), it can be 14 times slower than XMill while on average it is about to double the time of XGrind. This long compression time is mainly due to the hashing cost in the processing of each input item [8]. While the initial compression is slow, XSeq supports online incremental data modifications. This is mainly due to the property that the rule set is **context free**. In the case that part of the data values have been removed or changed, XSeq just needs to update the grammar rules that contain these values. It updates these nodes as well as the indices on the path to the starting symbol. Modifications in the

Benchmark		Path (sec)	Filter (sec)	Index (sec)	Data (sec)	XSeq (sec)	XGrind (sec)	Result count	Size (KB)
DBLP	XQ1	0.251	0.000	0.000	0.150	0.464	13.978	212272	2298
	XQ2	0.261	0.067	0.058	0.494	1.575	7.319	1494	751
	XQ3	0.261	0.079	0.336	2.748	4.161	47.319	54828	12990
	XQ4	0.543	0.016	0.000	0.034	0.732	–	140	3.5
XMark	XQ1	0.100	0.000	0.000	0.003	0.145	3.171	6537	62
	XQ2	0.112	0.002	0.047	0.010	0.224	3.047	1649	88
	XQ3	0.105	0.018	0.085	0.120	0.451	–	6037	1114
	XQ4	0.151	0.013	0.000	0.002	0.234	–	161	3.9
Swissprot	XQ1	0.356	0.000	0.000	1.044	1.526	23.042	344814	8362
	XQ2	0.203	0.056	0.415	0.109	0.883	–	49993	1000
	XQ3	0.168	0.030	0.232	0.337	1.120	–	1098	1153
	XQ4	0.809	0.189	0.000	0.002	1.145	–	36	1.4
Shakespeare	XQ1	0.023	0.000	0.000	0.016	0.047	1.826	30986	266
	XQ2	0.023	0.017	0.000	0.081	0.198	0.508	256	46
	XQ3	0.023	0.016	0.000	0.315	0.459	2.8400	180	23
	XQ4	0.030	0.002	0.000	0.001	0.045	–	60	0.5

Figure 6: Performance Evaluation Results.

original XML files can be localized without cascading update of irrelevant data values.

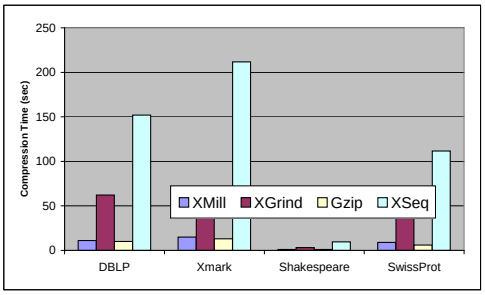

Figure 7: Compression Time.

4.4. Query Processing Time

We evaluated XSeq using four representative queries, XQ1 through XQ4, which are all listed in Appendix A. For each benchmark, XQ1 is a partial matching path expression. XQ2 is a path query with an exact-match or partial-match predicate on the result nodes. XQ3 is similar to XQ2 but it uses a range predicate. XQ4 extends XQ2 and XQ3 with multiple, and nested predicates. We collect detail query processing statistics about XSeq in Figure 6. The evaluation of each query consists of 4 parts, path-time, filter-time, index-time, and data-time. Path-time is the time taken to select the nodes that match the given path. Filter-time is for filtering out the nodes that do not match the predicates. Index-time and data-time give the time taken to retrieve the index and the data value of the result nodes respectively. We also give out the total querying time in column of XSeq, which is a little bit larger than the sum of the path, filter, index and data time. This is due to the fact that the total query time includes the time taken to parse the input query and to load the index information of the compressed file. We also compare it to XGrind. As XGrind does not support these queries directly, we made slight changes and then collect the processing time. We skipped the compilcated queries in XGrind as they require significant changes and the new code may impose performance impacts not related to XGrind. From the results we observed that XSeq performs constantly better than XGrind, with a performance speedup ranging from 2 to 20. The number of query results and the size are also recoded in the last two columns.

The performance comparison with recently proposed XQzip algorithm is shown in Figure 8. To compare with XQzip, we collect the XSeq results for the same queries used in [1]. As we use comparable setting and data source, we use the numbers of XQzip reported in [1] directly. We are aware of possible impacts due to detailed setting difference. We therefore performed additional experiments (discussed in the next section) to study the observation from this initial comparison.

As shown in Figure 8, XSeq and XQzip achieve comparable query performance for most queries. However, there are queries that XSeq performs better than XQzip while the rest XQzip is better than XSeq. We suspect that it is due to the sensitivity of different compression schemes with different query types, and thus would like to have a more detailed study of it.

Benchmark		XSeq (sec)	XQzip [1] (sec)
DBLP	Q1	0.327	0.381
	Q2	1.114	0.345
	Q3	2.832	9.541
XMark	Q1	0.102	0.913
	Q2	0.134	0.934
	Q3	0.117	3.411
Shakespeare	Q1	0.035	0.037
	Q2	0.079	0.038
	Q3	0.203	0.039

Figure 8: Comparison with XQzip (Q1,2,3 are the same as those in [1].

4.5. Processing Different Queries

To study the sensitivity of query performance in different schemes, we designed a study as follows. Given an uncompressed file, we divided the file into 10 segments and then randomly selected 1K and 10K values to process. We also varied the coverage with 10% increment. For example in Figure 9, for coverage 40% and count number 1K, these 1000 values were selected randomly from 4 out 10 segments, for example they can be 10-20%, 30-40%, 50-60% and 60-70% with the total coverage 40%. Each point in the figure is averaged from 10 runs. The compressed file of XSeq is as discussed above. The compressed file for XQzip follows the format in [1] – either a data block contains 2000 different values or its size reaches 2 Megabytes. The number for XSeq contains both the querying time and the time to process the input. The number for XQzip contains the input processing, and decompression cost only. That is, we optimistically assumed the query processing time

in XQzip is negligible.

FIgure 9 shows the results of processing "year" and "booktitle" containers respectively on DBLP data set. From the results, we observed that the processing time of partial decompression algorithms such as XQzip is almost linear to coverage variation. By processing 10K paths, the processing time changes from 5ms to 21ms ("year" result) when these paths are distributed within 10% and 100% of the file respectively. The performance of the latter is much worst because XQzip needs to decompress more compressed chunks. We do not consider buffering/caching across different queries due to the fact that consecutive queries may fall into different containers. In addition, the size of main memory buffers is limited when compared to that of uncompressed XML data.

On the other hand, the processing time in XSeq is almost constant for the change of data coverage – it changes from 5ms to 9ms for querying *year* container. This is due to the fact that there is no decompression involved in the processing. Varying coverage has less impacts since XSeq always finds and processes the individual compressed value directly.

Another observation that we made from the graph is that XSeq is more sensitive to the number of processed values. This is because we adapt a less optimized version in query processing, e.g., we process a visited data value node even it has been visited previously. In the future, we will study further improvements with intermediate result caching.

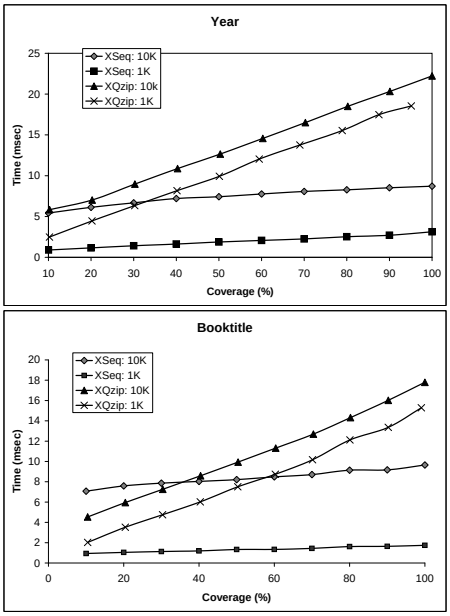

Figure 9: Query Processing with Different Coverage.

5. Conclusions

In this paper we proposed XSeq, an XML compressor that supports efficient query processing on compressed XML files without any decompression. Adapted from the on-line, linear-time Sequitur compression algorithm, XSeq compresses an input XML file into a set of context free grammar rules and their indices. The experimental results show that XSeq achieves comparable compression ratios of gzip and comparable query performance of XQzip. The results also reveal that its query processing is independent of data distribution and coverage. We expect the wide application of XSeq in resource constrained environments and in the processing of queries that access scattered data values in large XML files.

6. REFERENCES

[1] J. Cheng, and W. Ng, "XQzip: Querying Compressed XML using Structural Indexing," In *EDBT 2004*, LNCS 2992, 2004.

[2] A. Arion and *et. al.* "XQueC: Pushing Queries to Compressed XML Data," In *Proceedings of VLDB (Demo)*, 2003.

[3] P. Buneman, M. Grohe, and C. Koch, "Path Queries on Compressed XML," In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.

[4] M. F. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas," In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*, pages 14-23, Orlando, Florida, USA, 1998.

[5] H. Liefke, and D. Suciu, "XMill: An Efficient Compressor for XML Data," In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153-164, May 2000.

[6] J.K. Min, M.J. Park, and C.W. Chung, "XPRESS: A Queriable Compression for XML Data," In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 2003.

[7] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse, "XMark: A Benchmark for XML Data Management," In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974-985, Hong Kong, China, August 2002.

[8] C.G. Nevill-Manning, and I.H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Proceeding of the Data Compression Conference (DCC)*. Snowbird, UT, 1997.

[9] M. Neumuller, and J. N. Wilson, "Improving XML Processing using Adapted Data Structure," in LNCS 2593, 2003.

[10] Shakespeare, http://www.navdeeps.com/shakespeare/, Data Set, 2001.

[11] P. M. Tolani and J. R. Haritsa, "XGRIND: A Query-friendly XML Compressor," In *Proceedings of 18th International Conference on Database Engineering*, February 2002.

[12] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, W3C Recommendation 16 November 1999.

[13] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," In *IEEE Transactions on Information Theory*, May 1977.

Appendix A:

Queries used in Performance Evaluation

DBLP:
XQ1: //inproceedings//booktitle
XQ2: /dblp/inproceedings[booktitle$="SIGMOD"]
XQ3: /dblp/inproceedings[year>="1996"& year<="2000"]
XQ4: /dblp/inproceedings[[booktitle="SIGMOD Conference"]
&[year>="1996"&year<="2000"]&[author/$c>3]]/@key

XMark:
XQ1: //people//person//education
XQ2: /site/people/person/profile[education="College"]
XQ3: /site/people/person[profile/@income<=50000
& profile/@income>=10000]
XQ4: /site/people/person[[//@income<=50000 & //@income
>=10000]& [address] & [name$="G"]/emailaddress

Swissprot:
XQ1: //Entry//Descr
XQ2: /root/Entry/Mod[@type="Create"]
XQ3: /root/Entry[900>=@seqlen>=800]
XQ4: /root/Entry[[Descr?="PROTEIN"]&[900>=@seqlen
>=800]&[org/$c>=5]]/Species

Shakespeare:
XQ1: //PLAY/ACT/SCENE/SPEECH/SPEAKER
XQ2: //PLAY/ACT/SCENE/SPEECH[SPEAKER?="ANTONY"]
XQ3: /PLAYS/PLAY/ACT/SCENE/SPEECH["MARK ANTONY"
<=SPEAKER<="TROILUS"]
XQ4: /PLAYS/PLAY/ACT/SCENE/SPEECH[[STAGEDIR$="Enter"]
&"MARK ANTONY"<=SPEAKER<="TROILUS"]
&[line/$c>2]]/SPEAKER