Data Value Storage for Compressed Semi-structured Data

Brian G. Tripney[1], Isla Ross[1], Francis A. Wilson[2], and John N. Wilson[1]

[1] Department of Computer & Information Sciences, University of Strathclyde, Glasgow, UK
[2] Graduate School of Business, University of the South Pacific, Suva, Fiji
{brian,jnw,isla}@cis.strath.ac.uk, wilson_f@usp.ac.fj

Abstract. Growing user expectations of anywhere, anytime access to information require new types of data representations to be considered. While semi-structured data is a common exchange format, its verbose nature makes files of this type too large to be transferred quickly, especially where only a small part of that data is required by the user. There is consequently a need to develop new models of data storage to support the sharing of small segments of semi-structured data since existing XML compressors require the transfer of the entire compressed structure as a single unit. This paper examines the potential for bisimilarity-based partitioning (i.e. the grouping of items with similar structural patterns) to be combined with dictionary compression methods to produce a data storage model that remains directly accessible for query processing whilst facilitating the sharing of individual data segments. Study of the effects of differing types of bisimilarity upon the storage of data values identified the use of both forwards and backwards bisimilarity as the most promising basis for a dictionary-compressed structure. A query strategy is detailed that takes advantage of the compressed structure to reduce the number of data segments that must be accessed (and therefore transferred) to answer a query. A method to remove redundancy within the data dictionaries is also described and shown to have a positive effect on memory usage.

1 Introduction

New directions in the provision of end-user computing experiences make it necessary to determine the best way to share online data. The volume of data available over the Internet grows on a daily basis. At the same time, end users' expectations are increasing, with smartphone users now expecting instant access to information wherever they may be. The array of different processing techniques in use necessitates a standard format for data exchange and the self-describing nature of semi-structured data, in particular XML, has led to its common usage for this purpose. The side effect of this property is that file sizes quickly become large, with a high proportion of this being contributed by the description of the file format. XML compression techniques have partly addressed this by reducing storage requirements at the significant expense of requiring additional processing to access the data contained within the compressed files. However, an entire data structure is often not required, with users typically only interested in a small subset of the data. In such cases it follows that only the parts of the data structure of interest to the user need be accessed by the query processing system. Where the data is appropriately partitioned, or broken into segments, such an approach can limit the volume of data to be processed. A similar effect can be seen with data transfer. By only transporting the data segments directly involved in answering a query, the overall communications bandwidth utilised is also reduced. The effect is multiplied where additional queries can be answered using the data segments already held. In a system allowing sharing between peers, the data can be sourced from another local device rather than the server - again there is benefit in only sharing the segments required to resolve the query. Existing XML physical models are either non-queryable ([1], [2], [3], [4]) or have other drawbacks, e.g. requiring large sections of data to be decompressed in order to access a single value ([5], [6], [7], [8], [9], [10], [11]). In all cases these existing storage models require the entire data structure to be transferred as a single unit to allow any access to the data contained within. To facilitate sharing, a data storage model should be able to partition the semi-structured data into segments and store these in a manner that maximises utilisation of storage space while still making the stored values easily accessible. Semi-structured data can be separated into segments of related data by a process based around bisimilarity [12][13] - where items with similar structural patterns are grouped together. Such segmentation forms the basis of a data storage model that allows individual segments to be shared and recombined as required. Support for queries can be maintained by utilising an independent method of compression for each data segment - i.e. the whole structure should not be required to access the data stored in any one of the segments. The contribution of this paper is to characterise the effect of alternative approaches to bisimilarity on partitioning XML data structures with a view to identifying the most promising approach. Partitioning in this way produces redundant dictionaries and we examine the potential for curbing this problem. Lastly we demonstrate improvements in query performance using partitioned, compressed data structures. Section 2 reviews the previous work in the areas of semi-structured data compression, indexing and structural summarisation. The experimental work is set out in Section 4 and the results are presented and discussed in Section 5.

2 Background

XML documents are fundamentally tree-based structures although it is possible to superimpose links across the tree. This makes it convenient to use a datagraph as a convenient abstract representation of a document. The datagraph provides a graphical representation of the document structure showing each individual XML element as a node within a graph and the structure of the document as edges connecting the nodes. This presents a starting point for approaches to partitioning and indexing XML structures. Work-load aware methods of partitioning

Table 1. Comparison of existing XML compressors

	Queryable	XML Schema	Backend compressor	Decompression unit	Inequality without decompression	Comment
XMill[1]	No	No	gzip	Full	N/A	User must specify groupings and compressors to achieve claimed level of compression.
XMLPPM[2]	No	No	PPM	Full	N/A	Statistical modelling allows better compression than default XMill, but maintaining four models is slow.
SCA[3]	No	Req.	gzip	Full	N/A	Slower and less effective than XMill. Requires XML schema.
XWRT[4]	No	No	gzip	Full	N/A	Can improve compression, but must fully adjust parameters.
XGrind[5]	Yes	Opt.	Huffman	Value	No	Compression requires two passes over document. Querying requires parsing of entire compressed document. Only exact matches can be found.
XPRESS[6]	Yes	No	Huffman/ dictionary	Value	No	Improves querying over XGrind - no need for linear parse of document. Compression still considerably worse than XMill.
QXT[7]	Yes	No	gzip	Container	No	Compressed size is design priority. Must unzip full container before any examination of transformed contents.
XQueC[8]	Yes	No	ALM/ Huffman	Value	Poss.	Prefers advance knowledge of query workload to choose compressors. Requires large auxiliary data structures to permit querying - must manage pointers to individual items within containers.
XQzip[9]	Yes	No	gzip/ dictionary	Block	No	Values held in blocks of 1000. Requires decompression of full block to access single value. Authors recognise this and attempt to compensate with buffer pool.
XCQ[10]	Yes	Req.	gzip	Block	Part.	Stores less structure so smaller compressed files, but requires schema to do so.
ISX[11]	Yes	Opt.	gzip	Block	No	Emphasis on traversal of structural part.

XML data provide benefits in distributed processing of native XML structures [14]. Other partitioning techniques seek to improve query performance by mapping XML into object hierarchies [15]. Path-base partitioning [16] and variations that are tuned to particular query patterns [17] have also been shown to provide significant benefits. Work on DataGuides [18] recognised the repetitive nature of datagraphs and exploited it to aid querying of schema-less semi-structured databases. By extracting only unique paths of nodes that exist within the datagraph, the DataGuide produced is a compact and accurate summarisation of the database structure offering useful information for query authors. Bisimilarity [12][13] exploits patterns in the types of nodes in the datagraph that are connected to each other. Establishing bisimilarity is a two stage process. First, the two nodes must be labelled the same, i.e. the two elements represented by the nodes must be of the same type. Secondly, the paths connected to the two nodes are examined to ensure that the labels of the ancestor nodes (via the incoming paths) are the same for each node and that the labels of the descendant nodes (via the outgoing paths) are the same for each node. A variety of approaches to bisimilarity are possible by varying the direction and length of the paths examined. The A(k)-index [12] is a family of indices created using different levels of backwards bisimilarity (`k`), i.e. it is the incoming paths that are considered for purposes of determining bisimilarity. Forwards bisimilarity

```
<ContactList>
  <Student>
    <Name>Person1Name</Name>
    <Contact>
      <Email>person1name@example.com</Email>
    </Contact>
  </Student>
  <StaffMember>
    <Name>Person2Name</Name>
    <Contact>
      <Email>person2name@example.com</Email>
    </Contact>
  </StaffMember>
  <StaffMember>
    <Name>Person3Name</Name>
    <Contact>
      <Telephone>07780858382</Telephone>
    </Contact>
  </StaffMember>
</ContactList>
```

Listing 1.1. XML Example

(i.e. the comparison of outgoing paths) can be used to exploit the sharing of common sub-trees [13]. Applying forward and backward bisimilarity alternately until a point is reached where no further changes are made to the node groupings, produces an F&B-Index, which is the smallest index that accurately covers all branching path queries (i.e. those where the query does not take a linear path through the index)[19] Limiting the levels of bisimilarity used (as is the case for backwards bisimilarity in the A(k)-Index above) and limiting the number of times the main F&B-Index computation is performed produces a more compact (j,k)-F+B-Index[1] at the expense of accuracy [20]. These methods aid the access to the XML data by supplementing or replacing the structural part of the document. They also provide a basis for approaches to XML compression methods that involve both the document structures and the data values contained within them.

A variety of approaches have been used to design XML compressors that represent both the datagraph structure and the values contained within it. Characterisitcs of key systems are summarised in Table 1. While the non-queryable methods require full decompression before any further processing may take place, the queryable methods require different levels of decompression to answer a query. This is related to the choice of backend compressor, with those methods using gzip having to decompress complete blocks or containers to access a single value, while the methods built upon dictionary or Huffman compression are able to directly access the individual required value, reducing the decompression workload. A desirable feature is the ability to evaluate inequalities without the need to decompress individual values. A partial version of this is found in XCQ which can make use of the maximum and minimum values stored in the block statistics signature to quickly rule out an entire block of values. Thereafter the matching blocks must be decompressed in their entirety to complete the query. In XQueC,

[1] Where `j` and `k` specify the levels of forwards and backwards bisimilarity respectively.

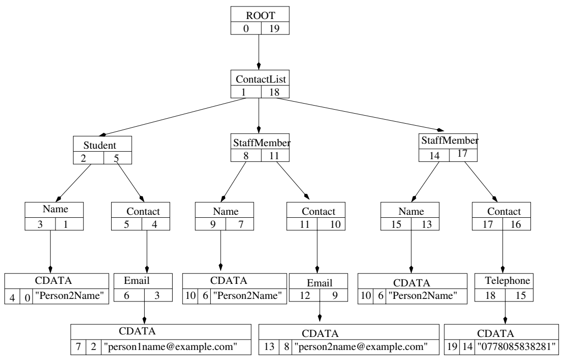


Fig. 1. Structural representation of forwards and backwards bisimilarity

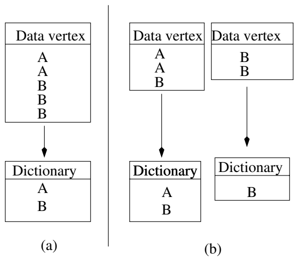


Fig. 2. Effect of data value distribution

the order-preserving nature of compression allows comparison of the individual compressed values. Additional user input in the form of advance knowledge of the query workloads is also required. This additional dependency on the user is also seen in XMill and XWRT, which require the user to manually set a number of parameters to achieve best performance.

The focus of the methods reviewed in Table 1 is the compression of entire data structures rather than methods that are designed to compress data structures that have been split into parts for sharing.

3 System Architecture

Bisimilarity-based structural summarisation naturally separates data into discrete groupings (partitions), which have the potential to form the basis of a system where small sections of a data structure can be distributed in an environment that requires data to be shared. XML data can be partitioned using a variety of approaches based around different charactersitics of the structure [21]. Bisimilarity takes the surrounding structure into account and is shown to provide a flexible method of grouping similar vertices, particularly in the context of real world data structures. The partitioning used is based on the F&B Index in which nodes of the datagraph are deemed to be bisimilar if their labels match and the same is true for both the ancestor nodes (incoming paths) and descendant nodes (outgoing paths) - this is applied repeatedly until a structure with stable node groupings is found. The structure is supplemented by a numbering scheme, which maintains the ordering of the data throughout [22]. Datagraph nodes are grouped by the bisimilarity function described earlier, so each entry within a particular vertex is of the one type and it is this type that appears at the top of each vertex in the diagram. Entries are marked with a pre-order number, which corresponds to the order in which the associated XML elements appear in the original document - thus maintaining the order of the stored data.

```
ROOT
0,19,1,21
ContactList
1,18,2,20
Student
2,5,3,7
Name
3,1,4,3
CDATA
4,0,4,1,''Person1Name''
Contact
5,4,4,4
Email
6,3,5,3
CDATA
7,2,5,1,''person1name@example.com
    ''
.
.
.
```

Listing 1.2. NSIndex File Format

Table 2. Experimental data structures

	Regular	Irregular
Benchmark	Orders[1] Modified Orders[3]	XMark[2]
Real World	Legal[4]	NASA[5] Medline[6] Dream[7] Rat[8] Human[9]

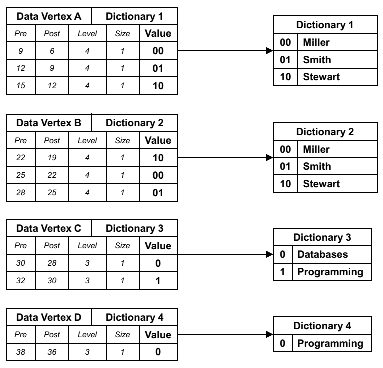
[1]Subset of TPC-H represented as XML (15Mb)
[2]Variable structure and random text (10&30Mb)
[3]Value-based elements of TPC-H subset (15Mb)
[4]Court sentencing data (1&13Mb)
[5]Text of *Midsummer Night's Dream* (0.15Mb)
[6]US National Library of Medicine bib (20Mb)
[7]Astronomical Data centre bibliography (24Mb)
[8]Ensemble rat genome annotation data (25Mb)
[9]Ensemble human genome data (25Mb)

They are also marked with post-order number, obtained from the last time each element is encountered in the document. This may be thought of as the order in which the end tags are found - with data values also numbered. Also recorded in each vertex entry, though not shown in the diagram, is the level at which the entry appears within the structure and the size of the entry, including any sub-trees that appear beneath it in the structure. For example, the `Contact` entry `(17,16)` shown at the right-hand side of Figure 1 will have level 4 (`ROOT` is counted as level 1) and size 3, counting itself and the two entries below it in the structure - `Telephone(18,15)` and `CDATA(19,14,"07780858392")`. This example also shows that the structural part of the `Telephone` element is stored in vertex (`18,15`), while the data value itself is stored in a separate vertex that holds the data entry (`19,14,"07780858392"`).

The structure is extended by a storage module that allows the structure to be written to disk. The physical representation is comma separated values (CSV) format as shown by the example in Listing 1.2. The data type of the vertex is output first, followed by each entry contained within the vertex on subsequent lines. Each structural entry is recorded as a sequence of pre-order, post-order, level and size, with the data value additionally being recorded for data entries.

4 Experimental Work

The main computational challenges addressed by this work are to devise a consistent way of partitioning semi-structured data so that it is possible to represent the associated data values in a compact form and yet retain the ability to address each value separately in this compressed form without the need to decompress the entire data structure.



Data Vertex A		Dictionary 1		
Pre	Post	Level	Size	Value
9	6	4	1	00
12	9	4	1	01
15	12	4	1	10

Dictionary 1	
00	Miller
01	Smith
10	Stewart

Data Vertex B		Dictionary 2		
Pre	Post	Level	Size	Value
22	19	4	1	10
25	22	4	1	00
28	25	4	1	01

Dictionary 2	
00	Miller
01	Smith
10	Stewart

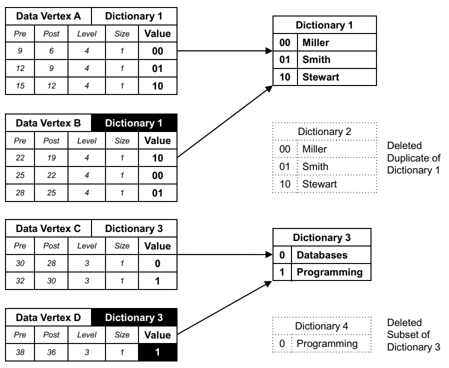
Data Vertex C		Dictionary 3		
Pre	Post	Level	Size	Value
30	28	3	1	0
32	30	3	1	1

Dictionary 3	
0	Databases
1	Programming

Data Vertex D		Dictionary 4		
Pre	Post	Level	Size	Value
38	36	3	1	0

Dictionary 4	
0	Programming

Fig. 3. Data vertices and dictionaries before reduction



Data Vertex A		Dictionary 1		
Pre	Post	Level	Size	Value
9	6	4	1	00
12	9	4	1	01
15	12	4	1	10

Dictionary 1	
00	Miller
01	Smith
10	Stewart

Data Vertex B		Dictionary 1		
Pre	Post	Level	Size	Value
22	19	4	1	10
25	22	4	1	00
28	25	4	1	01

Dictionary 2	
00	Miller
01	Smith
10	Stewart

Data Vertex C		Dictionary 3		
Pre	Post	Level	Size	Value
30	28	3	1	0
32	30	3	1	1

Dictionary 3	
0	Databases
1	Programming

Data Vertex D		Dictionary 3		
Pre	Post	Level	Size	Value
38	36	3	1	1

Dictionary 4	
0	Programming

Fig. 4. Data vertices and dictionaries after reduction (updated values shaded)

To permit the sharing of independent segments of data it is necessary to make use of an appropriate storage method. This must arrange the data into sections and allow these to be transferred and accessed individually. While existing XML compression methods group data into containers for compression purposes, the entire compressed structure must be transferred before any querying may take place, making these methods unsuitable for sharing data segments independently. The method of data storage proposed here combines data partitioning with a system of compression to store the data values. Dictionary-based compression was chosen over character-based compression as a consequence of the overall functionality available with the former approach [23]. The structural summarisation produced by bisimilarity affects the groupings of data values. These experiments evaluate the effect of changing the partitioning scheme on the number and size of data dictionaries required. Since the distribution of data values influences dictionary structure, a range of real-world data sets were used to assess the effect of partitioning. These sets were further classified as being regular or irregular in structure and supplemented by benchmark data (Table 2)Regular benchmark data includes both value based data (ModifiedOrders) and a mix of value and text based (Orders).

Bisimilarity options (no bisimilarity where data entries are grouped by label only, full backwards bisimilarity, full forwards bisimilarity and full forwards and backwards bisimilarity) affect the compressed size of an XML structures partitioned by these approaches. Changing the partitioning has an effect on the number of data vertices over which the data values are distributed. It is this distribution of data values that has an effect on the overall compressed data size, as repeated data values within a single data vertex require only one unique entry in the associated data dictionary, while repeated values that occur across a number of data vertices will have an entry in multiple dictionaries. An example is given in Figure 2 using the data values `A`, `A`, `B`, `B`, `B`. If, as shown in Figure 2(a), these are partitioned in such a way that all of the `B` values fall into the same data vertex, then all three values are described by a single entry in the

```
ROOT
0,19,1,21
ContactList
1,18,2,20
Student
2,5,3,7
Name
3,1,4,3
CDATA, ContactList.0.dic        <-- dictionary reference
4,0,4,1,0                       <-- data value replaced by token
Contact
5,4,4,4
Email
6,3,5,3
CDATA, ContactList.1.dic        <-- dictionary reference
7,2,5,1,0                       <-- data value replaced by token
.
.
.
```

Listing 1.3. NSIndex Compressed File Format

dictionary associated with that data vertex. In this case the total dictionary size will be the size of value `A` plus the size of value `B`. In addition, each entry in the dictionary is related to an integer, which is used as a token to replace the entry in the datagraph structure. If as in Figure 2(b) the `B` values are separated into two separate data vertices, then an entry will be required in the dictionary associated with each data vertex containing a `B`. In this example, the result is that an extra dictionary entry for value `B` must be stored in the second data dictionary, increasing the overall dictionary size. This could have a considerable effect on the overall size of dictionaries, especially when longer data values are involved. A set of associated data dictionaries containing only the unique entries from each vertex was created for each of the four differently-partitioned sets of data vertices for each data set.

The overall size of the compressed data values and the associated dictionaries for each partitioning scheme is of interest here, in particular whether there are any particular bisimilarity options that perform consistently well across the various data sets used. Also of interest is the number of dictionaries produced as, with a view to creating manageable segments of data, the smaller the individual dictionaries, the more there will necessarily be. Listing 1.3 incorporates dictionary references and the encoded data values given in Listing 1.2.

When a datagraph structure is loaded from a file, each data vertex will hold the data entries with their tokenised data values and a reference to the associated dictionary. The dictionary itself is held centrally, which permits multiple vertices to refer to the same dictionary. Top-down queries over the datagraph would lead to a large number of dictionary decode actions and consequent poor query performance. Such an approach would select all data entries of the correct type for each query predicate before linking the entries that matched the last predicate back to the previous one and so on. An example is given in Table 3(a) based on the query `/A/B/"Data"`. Table 3(b) shows the compressed query strategy. At each stage, only those vertices that are children of the vertices at the previous level and of the correct type are considered as potential results. This is in contrast to the uncompressed strategy, which noted all the individual data entries of the correct type for each stage regardless of where in the overall structure they were situated. The compressed strategy takes advantage of the entries being bundled together into vertices and these vertices being linked by edges. Traversing the edges means that only the child vertices are considered at the next stage of query resolution. This greatly reduces the number of vertices to be considered at each stage, which can be important when atomic values are being considered. For each data vertex to be checked, the query data value must be encoded using the appropriate dictionary before the contained data entries can be evaluated - it is therefore an advantage to have limited the number of data vertices to be checked.

Whichever partitioning method is chosen, there is potential for a large number of data dictionaries to be created. The nature of the bisimilarity-based summarisation is such that the logical domains will be split across a number of data vertices and there arises the possibility of repetition of values across the set of data dictionaries. This makes poor use of storage space, particularly where there are duplicate dictionaries but also where one dictionary is a wholly contained subset of another. Removing redundancy is straightforward in the case of duplicate dictionaries - the duplicate is deleted and any data vertices that referenced it are updated to make use of the remaining dictionary. There is no need to change the compressed data tokens contained in these data vertices. For subsets, the process is slightly more involved as the additional values contained within the superset dictionary mean that the values in data vertices previously using the subset dictionary may be represented by different tokens in the superset dictionary. Therefore to rationalise the dictionaries, a translation table must be created to update the old tokens in the data vertex, which refer to values in the subset dictionary, to tokens that point to the same value in the new superset dictionary. An example of these processes is shown in Figures 3 and 4. The compressed data vertices and dictionaries before the reduction process are illustrated in Figure 3 with each of the four example data vertices having an associated dictionary. Dictionaries 1 and 2 are duplicates of each other and the values stored in Dictionary 4 are a subset of those in Dictionary 3. As depicted in Figure 4, the reduction process disposes of Dictionary 2 and points Data Vertex B at Dictionary 1 (no change to the compressed data values is required as these dictionaries are identical). The subset Dictionary 4 is then removed with Data Vertex D amended to point to Dictionary 3. In this case the compressed data tokens stored in the data vertex are updated to reflect the encoding used by the new dictionary. The problem here is to compare each dictionary using the fewest possible file accesses. In the worst case each file would have to be checked against every other file to test for duplication ($n(n-1)/2$ comparisons) and for the existence of a subset (a further $n(n-1)$ comparisons). Comparisons can be reduced by using advance knowledge of dictionary file and token sizes gathered as the dictionaries are created. This means only dictionaries of the same token size are compared while file sizes can be used to determine whether to test for

Table 3. Query example

(a)	(b)
Uncompressed approach	Compressed approach
Note all A entries	Note all A vertices
Note all B entries	Search children of A to find B vertices
Note all entries matching “Data”	Search children of B to find “Data” entries
Retain B entries with links to “Data” entries	Retain B entries with links to “Data” entries
Retain A entries with links to remaining B entries	Retain A entries with links to remaining B entries
Return results	Return results

duplication or a subset and, in the case of the latter, which dictionary to treat as the potential superset and which to treat as the potential subset.

5 Results and Discussion

Figure 5 displays the results produced by the various bisimilarity strategies applied to the sample data sets. For each data set processed using the no bisimilarity option, only one data vertex is produced. With no bisimilarity, datagraph nodes are grouped by their label and just as all nodes of type `StaffMember` or `Student` would be grouped together, so too the `CDATA` nodes (which contain all the data values) are grouped into a single data vertex. The addition of forwards bisimilarity causes no change in the number of data vertices - each data set again having only one. Although forwards bisimilarity may have effects on the partitioning of structural nodes elsewhere in the structure, a method that exploits outgoing paths predictably has no effect upon the data-containing leaf nodes, as these have no descendant nodes for forwards bisimilarity to examine.

Working in the opposite direction, the use of full backwards bisimilarity results in an increase in the number of data vertices produced. By looking back up the datagraph at the ancestor nodes, the names of the XML entities in each data set are taken into account during partitioning and this leads to the single data vertex being split into multiple data vertices. Data set variation in the number of data vertices produced using backwards bisimilarity results from both the number of different XML entity names within each data set and the number of levels contained within each datagraph. The net result of partitioning using this method is that one data vertex is produced for each uniquely-named path through the datagraph. Full forwards and backwards bisimilarity alternately applies the bisimilarity rules in each direction until a stable structure is found. This means that a split caused by forwards bisimilarity higher up the datagraph can have an effect on the data vertices produced at the bottom of the structure when backwards bisimilarity considers the ancestors of each data node. The full forwards and backwards bisimilarity partitioning method is clearly influenced by the semi-structured nature of the test data sets. This is most apparent for XMark-30, where the irregular structure of the data set leads to a large number of data vertices when forwards and backwards bisimilarity is employed. This

Table 4. Path queries evaluated in Figure 8

Q#	Data Set	Query
Q1	XMark10	/regions/africa/item
Q2	XMark10	/regions/*/item
Q3	Legal-1	/sis/pc_age="21"
Q4	Legal-1	/sis/[pc_judge="J1" & pc_category="C1"]
Q5	Orders-1	/T/O_CUSTKEY="370"
Q6	Orders-1	/T/[O_ORDERSTATUS="O" & O_ORDER-PRIORITY="1-URGENT"]

form of bisimilarity produces the highest number of data vertices across all data sets with the exception of Orders-15 and ModifiedOrders-15. These data sets have a simple, regular structure which is unaffected by forwards bisimilarity - there are no variations in outgoing paths for any type of node.

The compressed size is the size of the data dictionaries plus the size of the compressed data values (as tokenised using the dictionaries). Figure 6 shows that as with results for the number of data vertices produced, the compression levels achieved for the data sets are the same for both the no bisimilarity and the full forwards bisimilarity partitioning methods. Again this is expected as the result of the partitioning process is the same for each method.

The distribution of the data values across the data vertices affects the impact of backwards bisimilarity. In the case of XMark-30 the introduction of backwards bisimilarity increases the compressed size over that produced using no bisimilarity. In this case the redistribution of the data values into a greater number of data vertices has led to a reduction in the overall levels of repetition within those data vertices consequently the dictionary-based scheme cannot operate as effectively and the compressed size rises.

The combined use of full forwards and backwards bisimilarity leads to an increase in the number of data vertices. This affects data value distribution and consequently the compressed data size. The increase in compressed size for XMark-30, Legal-1, Legal-13, Medline and NASA is caused by a reduction in the repetition of data values within the data vertices, while Dream, Rat and Human all reduce in compressed size as the increased distribution of data values across the data vertices separates the values in such as way as to allow the use of smaller token sizes. As previously noted, forwards bisimilarity has no effect upon the number of data vertices in the Orders-15 and ModifiedOrders-15 data sets and consequently has no effect on compressed sizes for these data sets either.

On balance it appears that, in terms of compressed data size, the full backwards bisimilarity partitioning method offers the greatest benefit for the majority of data sets. The significant exception to this is XMark-30 which experiences the best compression when using no bisimilarity (grouping by label only), and to a lesser extent the Dream, Rat and Human data sets which all slightly favour the full forwards and backwards bisimilarity method. However, the overall compressed size is only one factor when selecting a partitioning method, the number of data vertices produced must also be considered as this has an effect on individual data vertex size. Therefore, despite the slight adverse effect on the overall compressed size of some data sets, it is considered that the greater number of

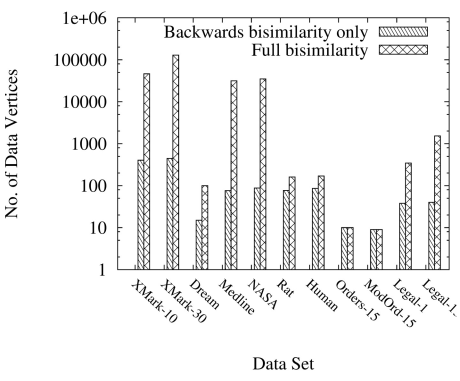


Fig. 5. Effect of bisimilarity on number of data vertices

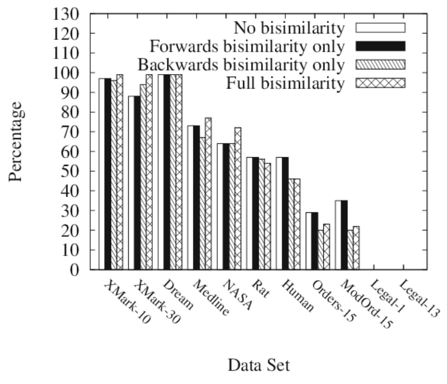


Fig. 6. Effects of bisimilarity on compressed sizes

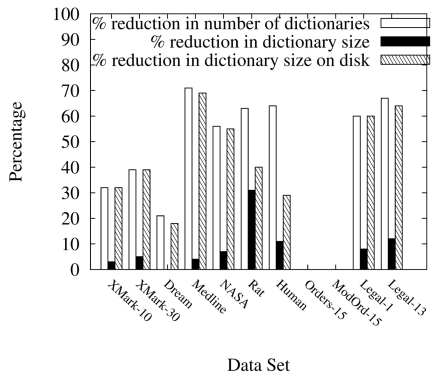


Fig. 7. Summary of dictionary reduction effects

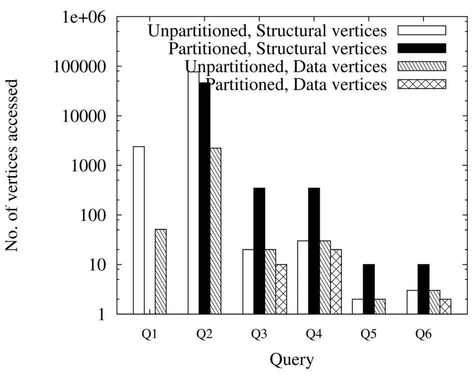


Fig. 8. Effect of partitioning the structure on query performance

data vertices produced by the full forwards and backwards bisimilarity method makes it the most reasonable compromise.

The work on querying the compressed structure, had two key objectives: first to demonstrate that the data values remained accessible in their compressed form and second to show that, in taking the partitioned structure into account, a query strategy could be formed that reduced the segments of the data structure required to be accessed to answer a query.

Query processing was evaluated by comparing the behaviour of the partitioned datagraph with the unpartitioned version. Dictionary structures were generated by using full F&B partitioning (Figure 8). Table 4 lists six queries performed over three of the test data sets - Legal-1, Orders-1 and XMark10. Performance is assessed by recording the number of vertices that need to be accessed to resolve each of the queries.

For structural queries (Q1 and Q2), the number of vertices accessed is reduced because the partitioned strategy accesses only those vertices which appear below the target node. As a purely structural query, neither query method accesses any data vertices. In Q2, the unpartitioned approach accesses all vertices, both structural and data, as it cannot distinguish between data and structural vertices in the context of the wildcard character.

For Q3 to Q6, the number of structural vertices required by each strategy does not differ between the query strategies. Given the short, regular structures of these data sets this is to be expected. There is however, variation in the number of data vertices. Due to the way in which the unpartitioned query strategy processes the query, each data vertex in the structure is accessed to check for matching data values. By contrast, the partitioned strategy only accesses the data vertices that satisfy the structural part of the query, reducing the pool of potential matches at each stage, so that a much smaller number of data vertices are required. In the case of Q3, the structure-minded strategy is able to narrow its search to only the 10 data vertices which hold `pc_age` values, as opposed to the full set of 346 data vertices that the unpartitioned strategy must access.

Changes to the total dictionary file sizes for each data set as a consequence of dictionary reduction are shown in Figure 7. For the Medline data set, 71% of its dictionaries have been removed by the reduction process, yet this leads to only a 4% reduction in logical dictionary size. This is because the dictionaries that have been removed are small. The large gap between the logical size reduction and the "size on disk" reduction stems from the removed dictionaries being much smaller than the 4Kb unit of disk space allocated by the filing system (the system was implemented on NTFS). Although this applies to every dictionary, as each logical file size is rounded up to the next 4Kb disk unit, the amount of wasted disk space depends on how close the dictionary is to filling the last disk unit allocated to that file. The effect is most pronounced on the smallest dictionaries, where the wasted disk space can form a much higher percentage of the "size on disk" allocated to the dictionary. It is also noted that for those dictionaries under a logical size of 2Kb, the wasted disk space will exceed that logically required by the dictionary entries. Where the logical sizes of the dictionaries removed are larger, such as with the Rat data set, the gap between logical size and "size on disk" is much lower as the wasted disk space forms a smaller percentage of the disk space allocated by the filing system.

A benefit of the dictionary-based scheme is that only those dictionaries actually involved with a particular query need to be accessed by the query system. This has implications where files are being requested over a data connection. In such a scenario the bandwidth utilisation would be limited to only those dictionaries that are useful. This is in contrast to other queryable ([8], [9], [10], [11]) for which the data structure must be transferred as a complete single entity.

It is not simply the total size of the dictionaries that is a potential benefit of dictionary reduction. While these savings in storage space are useful, the removal of the redundant dictionaries also means that the surviving dictionaries may relate to more than one data vertex within the datagraph. This increases the likelihood that additional user queries may be satisfied using dictionaries that have already been acquired.

Further improvements are possible by combining dictionaries with overlapping contents where the two dictionaries use the same size of token. Such a method would reduce the number of dictionaries without impacting on the compressed data size. However, any method of combining dictionaries that causes the token sizes within the compressed data to increase must be treated with caution, as the reduction in terms of overall dictionary size could easily be negated by the consequent increase in compressed data size. A trade-off is possible between dictionary numbers and compressed data sizes that allows for less wastage of the allocated storage space. In some cases, tokens can be represented in fewer bits than are used to represent an integer. In addition, further compression (eg Huffmann coding) may usefully be applied to the dictionaries. This could be particularly helpful where large text elements are contained within the data values, such as those in the Orders data set, as these are currently stored in the dictionaries in their original uncompressed form. While this could have benefits in terms of space occupied by the dictionaries, there is likely to be some impact on query performance.

6 Conclusion

The aim of this work has been the evaluation of a data storage model that facilitates the sharing of individual segments of data while maintaining support for user queries. It was proposed that this could be achieved using a combination of bisimilarity-based partitioning and dictionary-based compression. Work on the integration of data value compression described the steps necessary to build dictionary-based compression into datagraphs. The construction of dictionaries and the subsequent encoding of data values provides the basis for a query strategy that can deal with compressed values and take advantage of the partitioned structure. It was demonstrated that data values remained queryable in their compressed form and that the use of a structure-aware query strategy enabled the evaluation of queries using only the relevant parts of the data structure.

The contribution of this work is the evaluation of a data storage model that combines bisimilarity-based partitioning and dictionary compression methods. The evidence presented suggests that this approach has benefits in terms of data storage. Support for queries is not only maintained but also demonstrated to access only a fraction of the entire data set. The resulting structure is such that it lends itself to future exploitation in a system that shares independent segments of data.

References

1. Liefke, H., Suciu, D.: XMILL: An Efficient Compressor for XML Data. In: Proc ACM SIGMOD, pp. 153–164 (2000)
2. Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Data Compression Conference (DCC 2001), pp. 163–172. IEEE Computer Society (2001)

3. Levene, M., Wood, P.: XML Structure Compression. In: International Workshop on Web Dynamics (2002)
4. Skibinski, P., Grabowski, S., Swacha, J.: Fast Transform for Effective XML Compression. In: Proc CADSM, pp. 323–326 (2007)
5. Tolani, P.M., Haritsa, J.R.: XGRIND: A Query-Friendly XML Compressor. In: ICDE 2002, pp. 225–234 (2002)
6. Min, J.K., Park, M.J., Chung, C.W.: XPRESS: A Queriable Compression for XML Data. In: Proc ACM SIGMOD, pp. 122–133. ACM (2003)
7. Skibiński, P., Swacha, J.: Combining efficient XML compression with query processing. In: Ioannidis, Y., Novikov, B., Rachev, B. (eds.) ADBIS 2007. LNCS, vol. 4690, pp. 330–342. Springer, Heidelberg (2007)
8. Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., Pugliese, A.: Efficient query evaluation over compressed XML data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 200–218. Springer, Heidelberg (2004)
9. Cheng, J., Ng, W.: XQzip: Querying compressed XML using structural indexing. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 219–236. Springer, Heidelberg (2004)
10. Ng, W., Lam, W.Y., Wood, P.T., Levene, M.: XCQ: A queriable XML compression system. Knowledge and Information Systems 10(4), 421–452 (2006)
11. Wong, R.K., Lam, F., Shui, W.M.: Querying and maintaining a compact XML storage. In: Proc WWW Conference, pp. 1073–1082. ACM (2007)
12. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In: Proc ICDE 2002, pp. 129–140 (2002)
13. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML. In: Proc 29th VLDB, pp. 141–152 (2003)
14. Schroeder, R., Mello, R., Hara, C.: Affinity-based xml fragmentation. In: 15th WebDB (2012)
15. Alghamdi, N., Rahayu, W., Pardede, E.: Object-based methodology for xml data partitioning (oxdp). In: Proc IEEE AINA, pp. 307–315. IEEE (2011)
16. Marian, A., Siméon, J.: Projecting xml documents. In: Proc 29th VLDB, pp. 213–224. VLDB Endowment (2003)
17. Bidoit, N., Colazzo, D., Malla, N., Sartiani, C.: Partitioning xml documents for iterative queries. In: Proc 16th IDEAS, pp. 51–60. ACM (2012)
18. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB 1997, pp. 436–445 (1997)
19. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann (1999)
20. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering Indexes for Branching Path Queries. In: Proc ACM SIGMOD, pp. 133–144. ACM (2002)
21. Wilson, J., Gourlay, R., Japp, R., Neumuller, M.: Extracting partition statistics from semistructured data. In: Proc 16th DEXA, pp. 497–501. IEEE (2006)
22. Dietz, P.F.: Maintaining Order in a Linked List. In: Proc 14th ACM STOC, pp. 122–127. ACM (1982)
23. Tripney, B., Foley, C., Gourlay, R., Wilson, J.N.: Sharing large data collections between mobile peers. In: Proc 7th MoMM, pp. 321–325. ACM (2009)