Data Value Storage for Compressed Semi-Structured Data

Brian Grieve Tripney

A thesis presented in fulfilment of the requirements for the degree of Doctor of Philosophy

2012

University of Strathclyde
Department of Computer and Information Sciences

ABSTRACT

Growing user expectations of anywhere, anytime access to information require new types of data transfer to be considered. While semi-structured data is a common data 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 as existing XML compressors require the transfer of the entire compressed structure as a whole.

This thesis 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.

The use of dictionary compression is shown to compare favourably against Huffman-type compression, especially with regard to real world data sets, while a 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.

Having employed the above in a combined storage model, a query strategy is detailed which 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 in terms of disk space usage.

DECLARATION

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed:

Date:

PUBLICATIONS

The results of the experimental work detailed in this thesis have previously been published in the following papers:

Richard Gourlay, Brian Tripney, and John N. Wilson.
Compressed materialised views of semi-structured data.
In Workshops of the Twenty Fourth British National Conference on Databases, pages 75-82. IEEE Computer Society, Los Alamitos, California, 2007. ISBN 978-0-7695-2912-7.

Brian Tripney, Christopher Foley, Richard Gourlay, and John N. Wilson.
Sharing large data collections between mobile peers.
In Gabriele Kotsis, David Taniar, and Eric Pardede, editors, Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2009), pages 321-325. ACM, New York, New York, 2009. ISBN 978-3-85403-261-8.

Brian Tripney, Christopher Foley, Richard Gourlay, and John N. Wilson.
Efficient data representation for XML in peer-based systems.
In International Journal of Web Information Systems, Volume 6, Number 2, pages 132-148. Emerald, UK, 2010. ISSN 1744-0084.

To my late mother, Dorothy

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor, Dr John N. Wilson, for his advice and support throughout the course of this PhD. His words of encouragement during the drafting of this thesis have been invaluable.

Thank you to my fellow students in the Department of Computer and Information Sciences for making the office such a friendly place to work - it has made the process so much less arduous. Thanks also to my friends and colleagues at the SUERC Radiocarbon Laboratory for keeping me going through the redrafting process.

Last, but by no means least, I would like to thank my family for their love and support over the past few years. Thank you to my dad, Peter, for putting up with me on a daily basis, and thank you to my sisters, Gail and Dawn, for casting an eye over this document - I'm glad you found it more interesting than you thought it would be.

CONTENTS

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

LIST OF FIGURES

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 work described in this thesis evaluates a model for the storage of such data. This chapter sets out the general context of the thesis before identifying the hypothesis to be tested and the research questions that will aid in this process. An outline of the thesis structure rounds off the introductory material.

1.1 Context

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 demanding instant access to information wherever they may be. In June 2012, 10.4% of web page views were originated from mobile devices, up from 6.5% in 2011 and 2.6% the previous year[Sta].

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 most useful 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, the 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.

Take for example a museum setting where exhibit information is available via small handheld devices (possibly the visitor's own phone or tablet). A system where data can be sourced from other visitors' devices could reduce user wait times and cut server load such that relatively cheap hardware can be used to fulfil the server role. In such a scenario the server would only be accessed where a data segment required is not available from the pool of visitor devices.

This can be viewed as analogous to a dual layer of caching. In the first instance the device will try to answer the visitor's query using the segment or segments of data it already holds. As the visitor may well be interested in a number of similar exhibits, their device potentially has already gathered the relevant data while answering a previous query. If not, the device expands its search to the second layer cache - its peer devices. Data relating to popular exhibits is liable to be available from other visitors' devices. Even with more specialist artefacts, there is a reasonable chance that other members of the same visiting party would be interested in the same exhibits and have the relevant data segments available.

This sharing model can be expanded to applications covering a larger geographical area, for example work has been done on augmented reality tours of ancient sites such as Pompeii[VVI04]. Participating users could benefit from the reduced use of mobile data services provided they are willing to wait for a user with the necessary data segment to enter their vicinity. In such an environment it is important to ensure that the minimum data is transferred due to the costs involved in the process.

Any system built around the sharing of data segments will require an appropriate data storage model. Existing models are either non-queryable ([LS00],

[Che01], [LW02], [SGS07]) or have other drawbacks, e.g. requiring large sections of data to be decompressed in order to access a single value ([TH02], [MPC03], [SS07], [ABC+04], [CN04], [NLWL06], [WLS07]). 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.

1.2 Hypothesis and Research Questions

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 the usage 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 [KSBG02][BGK03] - where items with similar structural patterns are grouped together. It is proposed that such segmentation forms the basis of a data storage model that allows individual segments to be shared and recombined as required. Support for queries could 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 hypothesis assessed through this thesis is therefore that:

Independent sharing of data segments while maintaining direct query access is effectively facilitated by the combination of bisimilarity-based partitioning and dictionary compression methods.

This hypothesis directs study towards two major technical issues: the validity of dictionary compression in this context and the evaluation of bisimilarity-based partitioning methods as a useful means of segmenting a data structure. A system combining these two methods should allow the evaluation of queries while accessing only a portion of the data structure. Once these processes have been applied to semi-structured data, it is then necessary to explore the way that the resulting data model can be further processed to improve its utility. As a result of these issues, this thesis considers the following research questions:

RQ1: **Are dictionary-based methods a reasonable choice for use in a compressed semi-structured data storage model?**

In order to keep decompression to a minimum, the data storage model must maintain access to the individual values within a compressed data segment such that only values directly involved in the evaluation of a query need be decompressed. This means that either the data must be compressed on a per value basis (dictionary compression) or on a per character basis (akin to text compression) - it cannot be compressed at the segment level. String-repetition that occurs within the data sets may provide an opportunity for dictionary compression to offer improved compression with respect to that available with character-based compression.

RQ2: **What are the effects of varying the partitioning method on the storage of data values?**

Earlier work [GTW07] has considered the effect of changing the type of bisimilarity used in the partitioning process upon structural parts of XML, however this will also have an effect on the distribution of data values across the segments of the data storage model. The number of segments produced and the size of the associated dictionaries are of interest with respect to the future sharing of data.

RQ3: **Can a querying strategy designed around the compressed structure allow queries to be answered with reduced access to the data structure?**

To allow queries to be answered with the minimal transfer of data segments, the querying method used by the storage model must require access only to those segments that could potentially hold a query result. By taking account of the structure of the data during the query process, it is hoped to reduce the number of segments that are required.

RQ4: **With the data split into segments, how might the volume of dictionaries be managed?**

The partitioning of data into segments could potentially lead to a large number of associated dictionaries that may well contain a high degree of repetition within their union. Consideration will be given to the reduction of such redundancy by establishing a methodical manner of identifying and removing duplicate and subset dictionaries.

The research questions above form the basis of the main pieces of experimental work detailed in Chapter 4. A comparison between dictionary-based and text-based compression is set out in Section 4.3, while Section 4.4 evaluates the effects of differing types of bisimilarity-based partitioning. The design of a structure-aware query strategy is given in Section 4.6, and finally a method of reducing redundancy within the set of dictionaries is described in Section 4.7.

1.3 Thesis Outline

The main body of this thesis is organised as follows:

Chapter 2 reviews the previous work in the areas of semi-structured data compression, indexing and structural summarisation with a more detailed picture of the technologies built upon later in this thesis provided in Chapter 3.

The experimental work relating to the research questions above is set out in Chapter 4, which also details the additional coding tasks required to facilitate the main experiments (Section 4.5). The experimental results are presented and discussed in Chapter 5. This chapter also returns to the research questions and hypothesis set out earlier and considers the answers provided by the experimental work in this context. In addition, potential future avenues of research are indicated.

Finally a conclusion to the thesis is provided in Chapter 6.

2. BACKGROUND

Extensible Markup Language or XML was first published as a W3C recommendation in 1998 [XML]. Its aim was to introduce a standard method of storing data within a document that would be straightforward and easy to use. The XML markup consists of tags added to the data values that denote the start and end of each entity that contains data. A simple example of a contact list is given in Listing 2.1. It can be seen that all details pertaining to the student Joe Bloggs are contained between the `<Student> </Student>` tags and similarly for the two staff members between the `<StaffMember> </StaffMember>` tags. Each individual data item is also self-contained, e.g. Email and Telephone.

```
<ContactList>
  <Student>
    <Name>Joe Bloggs</Name>
    <Contact>
      <Email>j.bloggs@example.com</Email>
    </Contact>
  </Student>
  <StaffMember>
    <Name>Prof P. Pending</Name>
    <Contact>
      <Email>p.pending@example.com</Email>
    </Contact>
  </StaffMember>
  <StaffMember>
    <Name>Prof H. Higgins</Name>
    <Contact>
      <Telephone>0141 496 2232</Telephone>
    </Contact>
  </StaffMember>
</ContactList>
```

Listing 2.1: XML Example

As the W3C recommendation requires that XML documents be humanly-readable, and notes that terseness is unimportant, there is scope for XML to be

an effectively self-describing storage medium - making it highly useful as a means of data exchange between programs. However this same verbose quality means that markup can account for a considerable proportion of any XML document, above that required for the data values themselves, leading to large file sizes and a high degree of repetition within the documents.

This chapter categorises and describes the notable advances made in the storage and processing of XML documents that are relevant to the work described in this thesis. The first section looks at techniques that supplement or simplify the XML document structure, while the second section describes methods of compression particularly designed to cope with XML. These are broad fields and the examples detailed below have been selected to be representative of those approaches explored in the past. In particular, the technologies selected below are those that are notable for a novel approach or that are frequently cited in the literature. A summary of additional XML technologies can be found in comparison papers by Ng et al. [NLC06] and Sakr [Sak09].

Those XML processing methods that are built upon or used later in this thesis are explained in more depth in Chapter 3.

2.1 XML Indexing and Structural Summarisation

The first group of XML technologies described in this review concentrates more on the structure of XML documents than on the data they contain. The structural part of the document may be supplemented by an index to speed the querying of the original data structure. Alternatively the structure may be simplified so that it is effectively replaced by a summarised version of itself.

XML data structures can be represented as a tree structure. The first tag acts as the root of the tree with the nested elements thought of as branches emanating from the root. An extension of this XML tree concept is that of a datagraph - a graphical representation of the document structure showing each individual XML element as a node within a graph. The datagraph is used in this chapter as the starting point for describing methods that deal with XML structure and an example depicting the XML code from Listing 2.1 above in datagraph form is

shown in Figure 2.1.

Each element of the XML structure is replicated within the datagraph. Note that the datagraph begins with the root node `1`, the `ContactList` element, and that the edges of the graph are labelled to indicate what type of element appears next. Therefore node `2` is of type `Student`, node `6` is of type `Contact` and node `11` is of type `Email`. Node `11` represents Joe Bloggs' email address, mirroring the path `ContactList/Student/Contact/Email` found in the XML structure from Listing 2.1.

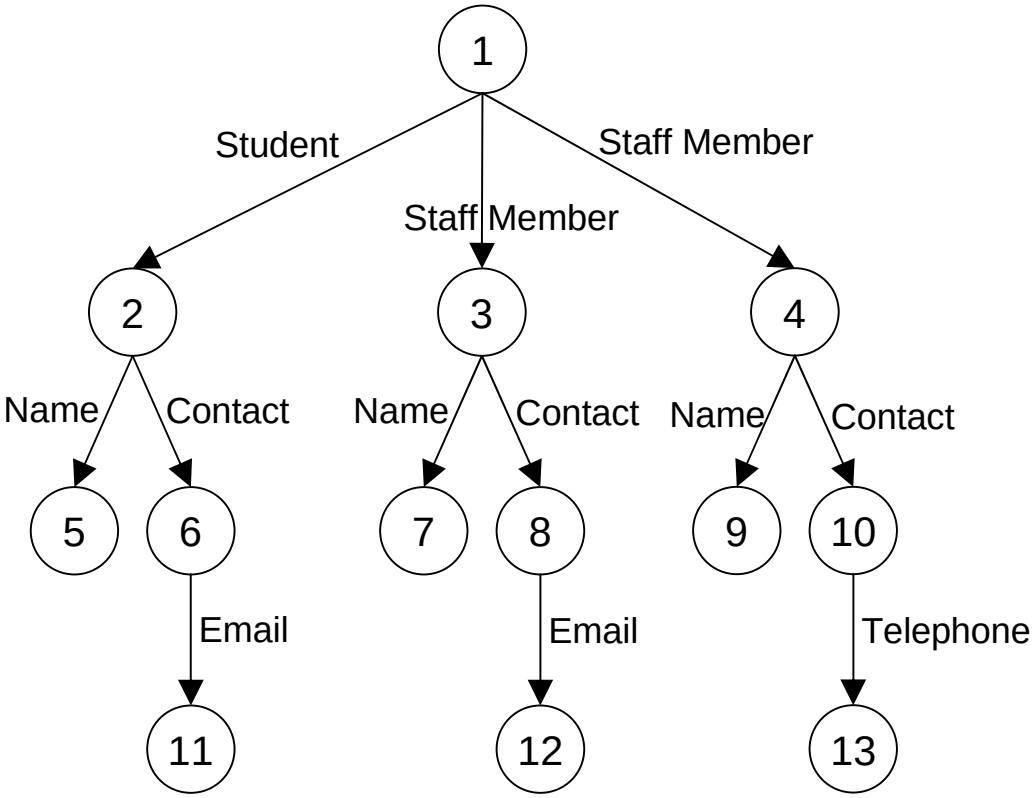

Fig. 2.1: Datagraph Representation of XML in Listing 2.1

Early work that builds upon the datagraph concept was carried out by Goldman and Widom. Centred more generally on semi-structured databases (it predates the XML recommendation), their work on DataGuides [GW97] recognised the repetitive nature of datagraphs and exploited this 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.

Figure 2.2 shows two DataGuide versions of the example datagraph set out in Figure 2.1 above. The DataGuide in Figure 2.2a contains each of the labelled edges from the datagraph once only with no duplication. Figure 2.2b is a minimal version of the DataGuide. As the outgoing edges from nodes `15` and `16` are the same, these nodes can be combined in the minimal version as node `25`, with the `Student` and `Staff Member` edges both appearing between nodes `24` and `25`.

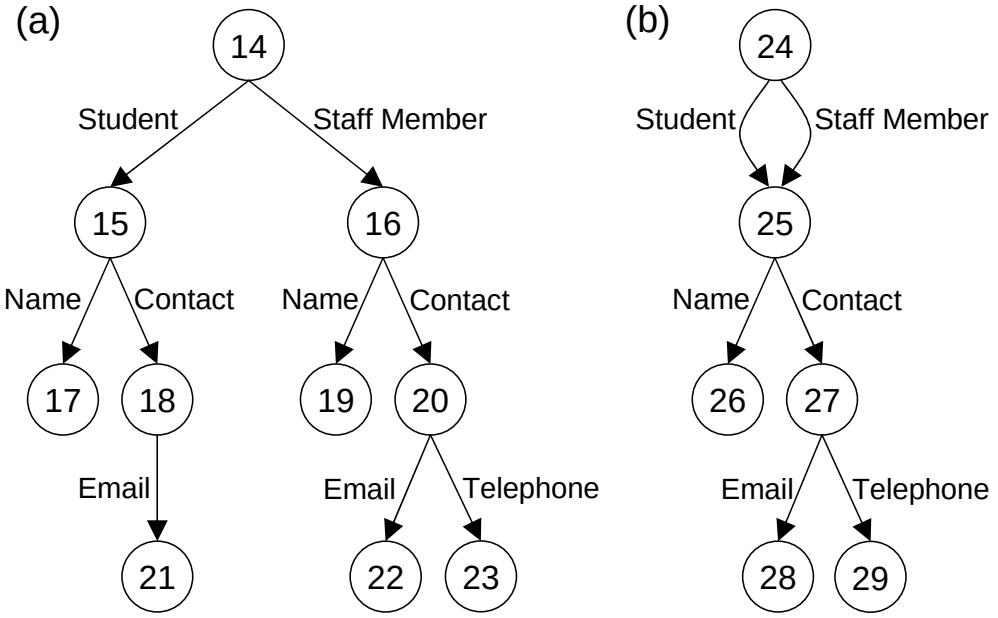

Fig. 2.2: DataGuide Representation of Datagraph in Figure 2.1

The structural summarisation methods listed below simplify the datagraph by exploiting patterns in the types of nodes that are connected to each other - a quality known as bisimilarity [KSBG02][BGK03]. Establishing the bisimilarity or otherwise of any two nodes in the datagraph 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. The methods described below each employ different varieties of bisimilarity, varying the direction and length of the paths examined.

The A(k)-index [KSBG02] 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. This work by Kaushik et al. groups bisimilar entities into a single node in the resulting A(k)-index.

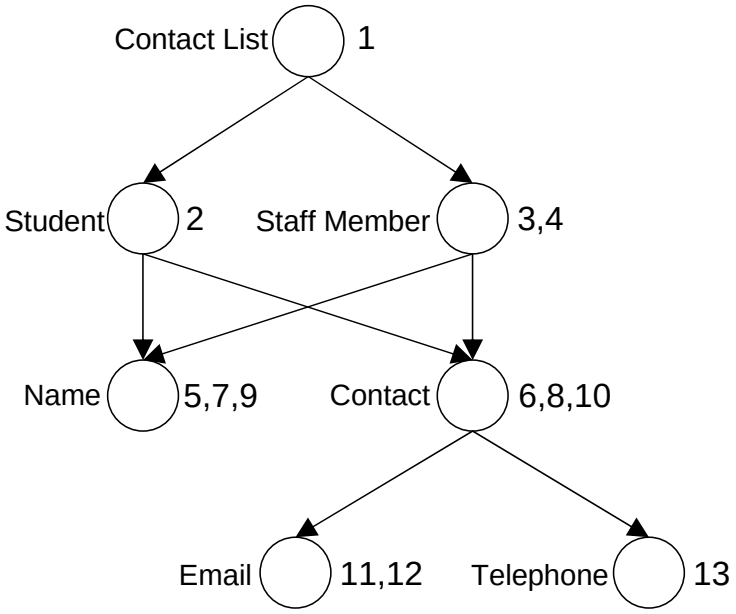

Fig. 2.3: A(0)-Index for Data Shown in Figure 2.1

Figures 2.3 to 2.5 depict the A(k)-indices produced from the example XML. Note that in these diagrams it is the nodes (as opposed to the edges) that are labelled and that the numbers next to each node represent the individual XML elements contained within that node according to the order that they appear in the original XML document.

The initial A(0)-index, as shown in Figure 2.3, simply groups nodes by their label type. For levels of `k` greater than zero the incoming paths to each node are

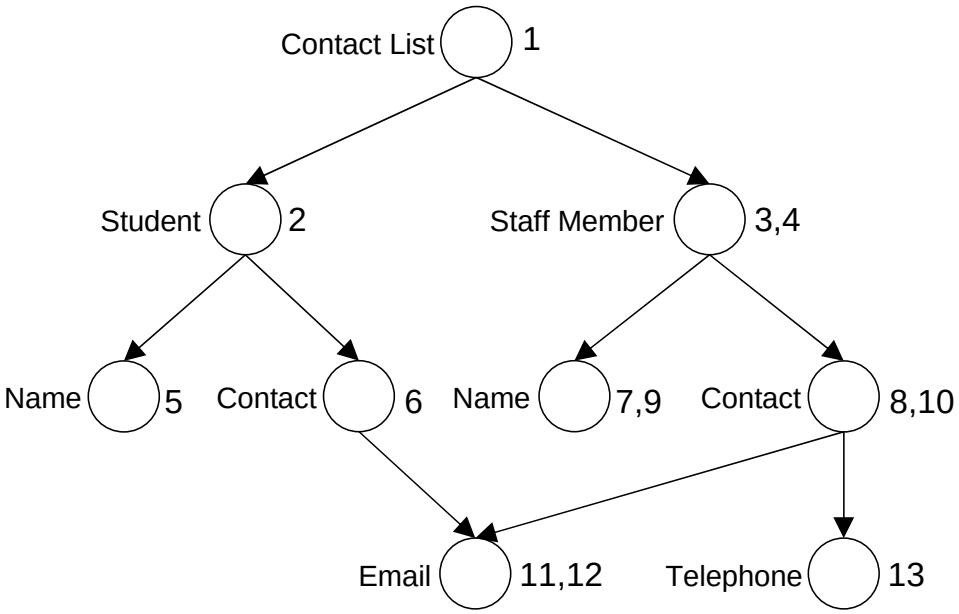

Fig. 2.4: A(1)-Index for Data Shown in Figure 2.1

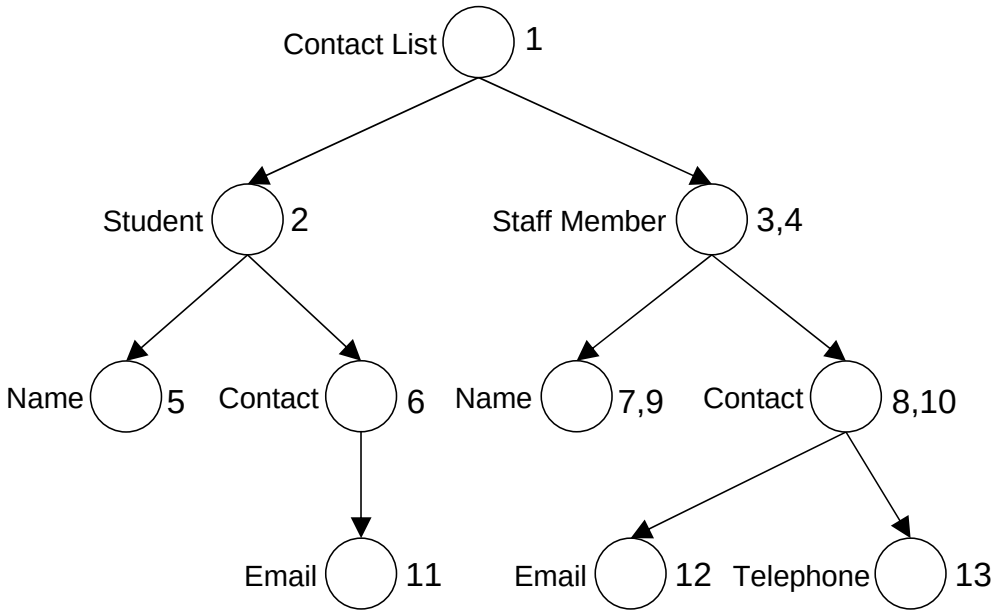

Fig. 2.5: A(2)-Index for Data Shown in Figure 2.1

also considered. Therefore for two nodes to be classed as bisimilar in the A(1)-index they must have the same label and their parent nodes must also have the same labels. This can be seen in Figure 2.4 where the `Name` elements have split into two nodes, one containing element `5` whose parent node has label `Student`, and one containing `7` and `9` which have `Staff Member` as parent. The same is true for the `Contact` elements, however the `Email` node holding `11` and `12` does not split as the parent node of each is of type `Contact`.

For higher levels of `k`, the length of incoming path examined is increased. The A(2)-index (Figure 2.5) looks at the parent and grandparent nodes - thus separating elements `11` and `12` which, although both have `Contact` as a parent node, have different grandparent node types (`Student` and `Staff Member` respectively). This pattern continues with the A(3)-index additionally considering the great-grandparent nodes (though further levels of `k` will have no effect on this example data).

The work of Buneman et al. [BGK03] also makes use of bisimilarity. Although their work is phrased as 'compression' (discussed in Section 2.2 below) only the document structure is dealt with, not the data values, making the concept involved more closely related to structural summarisation techniques such as the A(k)-Index above.

Their technique extracts the structural part of the document, then summarises this 'skeleton' through the sharing of common subtrees. Whereas the A(k)-Index makes use of backwards bisimilarity, this method employs forwards bisimilarity - it is the outgoing (as opposed to the incoming) paths of the nodes that are compared. An uncompressed skeleton version of the example data is shown in Figure 2.6, with the compressed version shown in Figure 2.7.

In Figure 2.7 `Name` and `Email` are included only once as the outgoing edges of each instance of these are the same (i.e. none). Two of the `Contact` elements are also merged into one as they each have the common subtree of `Email`. The multiple edges between `Contact` and `Email` can be replaced with a single numbered edge as shown in the inset part of Figure 2.7 to form a Fully Compressed Skeleton where the numbers indicate how many edges are represented. The order of the elements from the original XML is retained by the skeleton compression

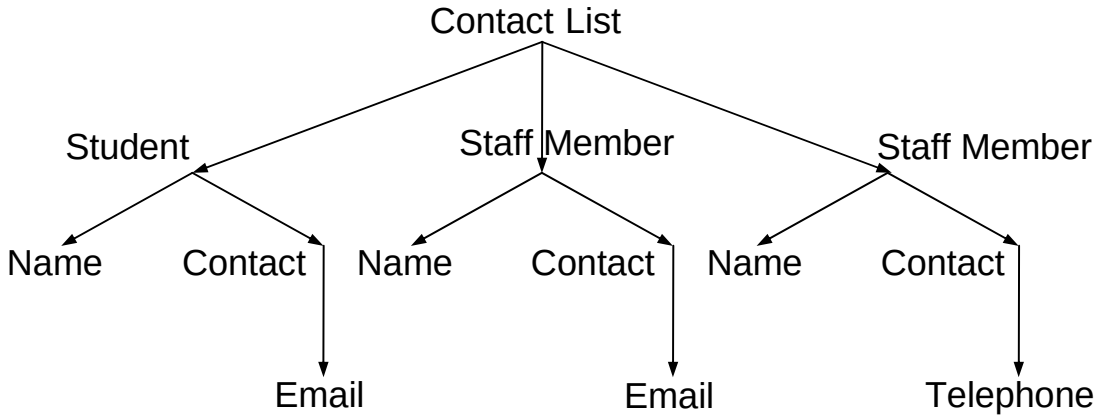

Fig. 2.6: Uncompressed Skeleton

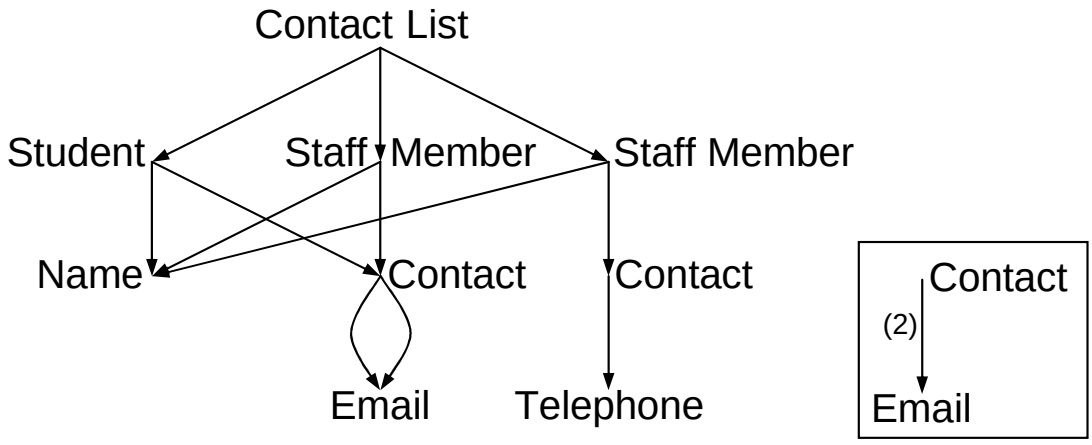

Fig. 2.7: Compressed Skeleton

technique and shown in the diagrams as left-to-right order.

Kaushik et al. [KBNK02] present two methods of employing both forwards and backwards bisimilarity within the same index.

The first is a restatement of ideas previously presented by Abiteboul, Buneman and Suciu [ABS99]. The Forward and Backward Index (F&B-Index) repeatedly applies alternating stages of forwards and backwards bisimilarity to the set of datagraph nodes, with each successive stage using the output of the previous stage as a starting point. This continues until a point is reached where no further changes are made to the node groupings and a stable index is formed.

This F&B-Index is the smallest index for which all branching path queries (i.e. those where the query does not take a linear path through the index) are accurately covered by the index.

Kaushik et al. then suggest a compromise, proposing an index that is considerably smaller than the F&B-Index, but at the expense of accuracy. This is done by limiting the levels of bisimilarity used (as is the case for backwards bisimilarity in the A(k)-Index above) and by limiting the number of times the main F&B-Index computation is performed. Although repeated iterations of this computation increase the length of branching query (i.e. the path length of the deviation away from the main linear path of the query) which can be accurately answered by the index, the size of the index also increases with each iteration. The resulting index is known as the (j,k)-F+B-Index, where `j` and `k` specify the levels of forwards and backwards bisimilarity respectively.

A simple example of the use of both forwards and backwards bisimilarity is shown in Figure 2.8. For this example diagram, the level of bisimilarity in each direction is set at 1. In comparison to the A(1)-Index (Figure 2.4), which uses only backwards bisimilarity, it can be seen that that (1,1)-F+B-Index has separated the two `Contact` elements `8` and `10`. This is because with the addition of forwards bisimilarity the outgoing paths of the nodes are also considered. Element `8` has child node of type `Email`, whereas element `10` has child of type `Telephone` - these are consequently grouped differently.

The methods covered in this section each aid the access to the XML data by supplementing or replacing the structural part of the document. The general

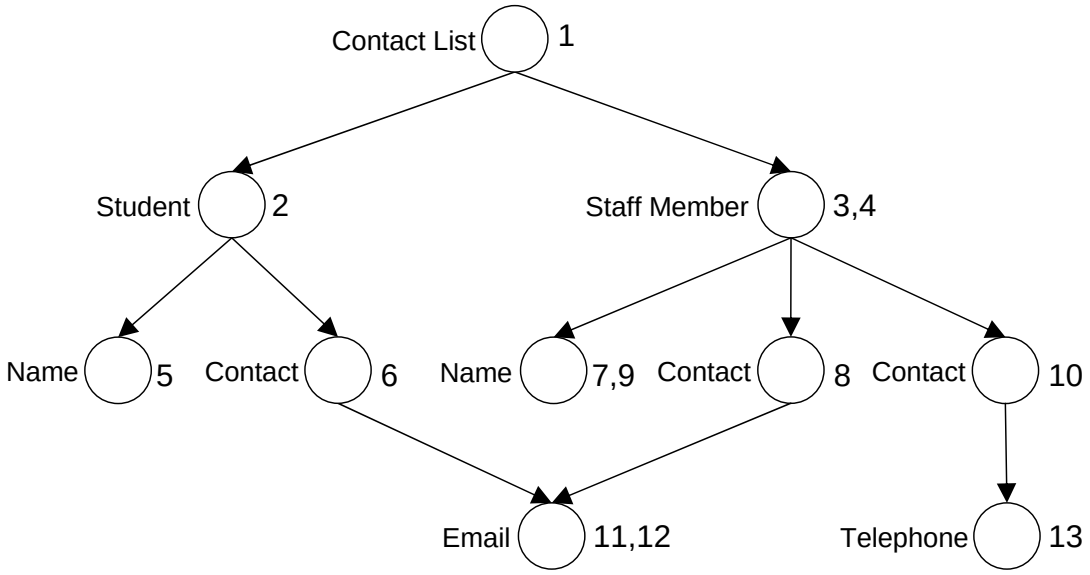

Fig. 2.8: (1,1)-F+B-Index of Data Shown in Figure 2.1

conclusion is that such methods assist in query processing at the expense of having additional, potentially large, structures. The remainder of this chapter is concerned with XML compression methods that involve both the document structure and the data values contained within it.

2.2 XML Compression Techniques

The techniques of the previous section were concerned largely with quick access to data contained within XML documents. To this end there was a focus on using the structural part of the document to move quickly to the data values contained within the XML. The methods examined in this section also have considerable interest in the structural part of the XML document, but additionally take account of the storage of data values. These XML compressors process the whole of the document with a view to reducing the overall file size.

Unlike general purpose compressors (such as gzip, bzip2, etc.), which see an XML file merely as another text file, the compressors noted below are XML-aware and take advantage of the repetitive structure of XML documents. The

XML compressors described here are split into three categories: Non-Queryable, Queryable Homomorphic and Queryable Non-Homomorphic. The Non-Queryable group are those compressors that take advantage of the structure of the XML document to improve compression but the output must be fully decompressed back to the original document before querying may occur. XML compressors that allow querying directly upon the compressed data are divided into two categories depending on how the compressed data is stored - those that keep structure and data values together are termed Homomorphic, while those storing data values separately from the structure are Non-Homomorphic.

2.2.1 Non-Queryable XML Compressors

The compression techniques outlined in this section are concerned solely with reducing the storage requirements of XML. No provision is made by these methods for the querying of data contained within the compressed XML, any document must first be decompressed before any querying can take place. Nevertheless, by taking advantage of the XML structure, a space saving can be made compared to general purpose text compressors. As a result, these non-queryable XML compressors may be considered more suitable for archival purposes.

One of the earliest XML-specific compressors, XMill [LS00], works by separating the structure and data values into a number of containers and compressing these individually. Figure 2.9 shows the process in more detail.

The XML document is read into XMill using a SAX Parser[1] and then passed through the Path Processor, which separates structure from data values. The structural elements are then tokenised and stored within the Structure Container such that each element tag is represented as `T1`, `T2`, etc. and that any occurence of a data value is replaced by a reference to the appropriate Data Container, e.g. `C3`.

Data values themselves are split according to their path in the original XML. The Path Processor then passes the data values for each path through a Semantic

[1] Simple API for XML, a parser that sequentially reads through a XML document and outputs a series of 'events' such as `StartElement x`, `TextData abc`, `EndElement x` as it encounters these.

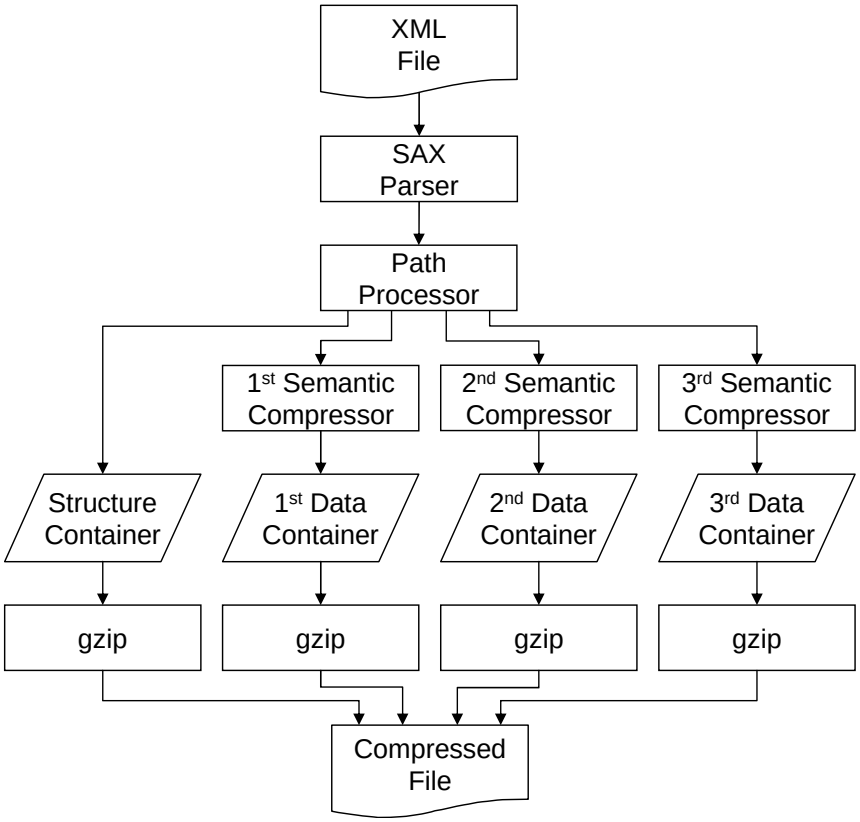
XML
File
SAX
Parser
Path
Processor
1st Semantic
Compressor
2nd Semantic
Compressor
3rd Semantic
Compressor
Structure
Container
1st Data
Container
2nd Data
Container
3rd Data
Container
gzip
gzip
gzip
gzip
Compressed
File

Fig. 2.9: XMill

Compressor - a compressor that is appropriate to the type of data encountered in that path (e.g. date, text, integer) - and stores the resulting data in a number of Data Containers.

A final step sees the Structural Container and each of the Data Containers compressed using gzip and output to a compressed XMill file. The authors report that this method achieves a compression ratio of around twice that of gzip alone.

Whereas XMill keeps structural elements together and separates data into multiple containers, Cheney's XMLPPM [Che01] divides up what XMill would term structure and keeps data values together.

The first stage of XMLPPM encodes the SAX Parser output such that each SAX event is represented as a one byte code. This encoded SAX output (ESAX) is then distributed as appropriate to four PPM encoders that work on the principle of Prediction by Partial Match (PPM). The output from each PPM encoder is based on the statistics built up as the encoding process takes place, i.e. the PPM encoder adapts to predict what the next symbol will be and only needs to record the difference between the predicted and actual symbols.

The first PPM model (Elts) encodes the element structure, the second (Atts) encodes attribute values, the third (Chars) encodes data values and the fourth (Syms) encodes the names of elements and attributes. Although there are four PPM models used for encoding, each makes use of the same set of symbols to prevent the models falling out of step with each other - thus maintaining the document structure.

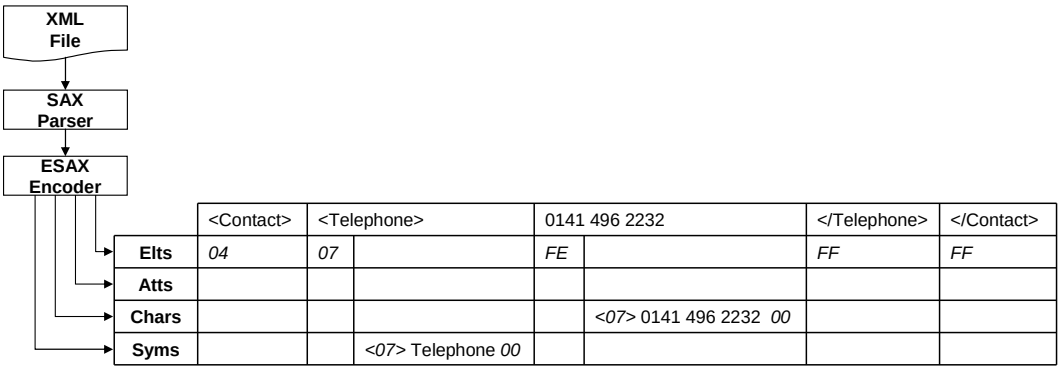

Fig. 2.10: XMLPPM - ESAX Encoding and Distribution

Figure 2.10 shows the ESAX encoding of a section of XML (from the example code in Listing 2.1) and the distribution of these ESAX events between the four PPM models. In this case, the `<Contact>` element has been seen before and is encoded in Elts as `04`, `Telephone` is encoded as `07` but is a new symbol so the element name is also passed to the Syms model. Note that `<07>` is passed to the Syms model to maintain context but is not encoded by the model, the code `00` marks the end of the element name. A code (`FE`) is then passed to the Elts model to mark the start of a data value, with the telephone number itself being passed to the Chars model (again `<07>` maintains context and `00` denotes the end of the text). Finally for this section of XML, the Elts model receives codes to close the `Telephone` and `Contact` elements (`FF`).

The output from each PPM encoder (not shown in Figure 2.10 is the difference between the actual symbol passed to the encoder and the symbol it had predicted would appear. This information is additionally used to update PPM model's statistics in an attempt to improve future predictions. A similar process takes place during decompression - the four PPM models are built and updated using the same method as during compression to allow decoding of the data.

An alternative approach to these previous methods, which extract structural information from the XML document, is to use the declared document structure (the XML schema) to drive the compression process. An example of such an approach can be found in Levene and Wood's Structure Compression Algorithm [LW02]. This approach focuses on the structure of the XML document though the data values (separated from the structure in the first stage and reinstated as the final stage of decompression) are stored and could potentially be processed by a general purpose compressor as simple text.

The structural part of the document is then compressed according to the rules set out in the XML schema as follows: Elements required by the schema are not recorded - these can be inferred during decompression. Optional elements are simply marked as present or absent. Where the schema allows elements to occur multiple times, the number of instances must be be recorded (and, where appropriate, the element types). An example is shown in Figure 2.11. A simplified schema for the running example XML is shown on the left of the

diagram, it dictates that `Contact List` consists of multiple `Student` or `Staff Member` elements while these in turn each hold one `Name` and one `Contact` element. `Contact` itself is made up of an optional `Email` and an optional `Telephone`. The `Name`, `Email` and `Telephone` elements all hold data values.

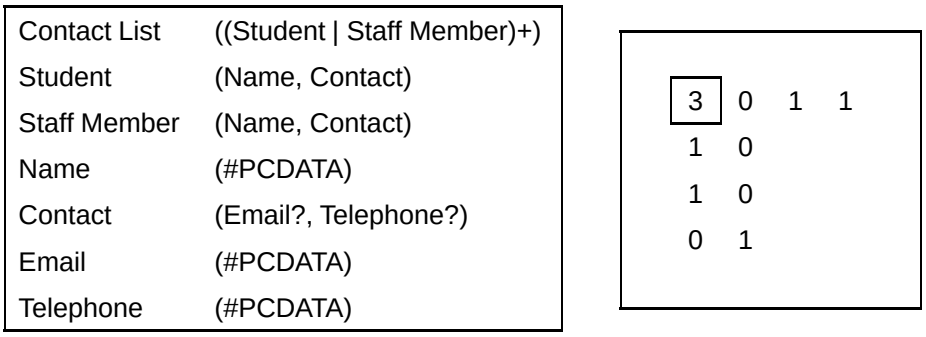

Fig. 2.11: Structure Compression Algorithm

To the right of Figure 2.11 is the compressed version of the example XML. On the first line, the integer 3 records that `Contact List` has three children, the remainder of the line notes (as a series of bit tokens) that there is a `Student` followed by two `Staff Member` elements. No information is recorded about the presence of the `Name` or `Contact` elements (as these are required by the schema). The second line of the compressed structure deals with the optional elements within the first `Contact` element. In this case the first bit denotes the presence of an `Email` while the second bit shows the absence of a `Telephone`. This is repeated on the third line for the first Staff Member's contact details and reversed on line four to show that the second Staff Member has no Email, but does have a Telephone.

The XML Word Replacing Transform (XWRT)[SGS07] by Skibinski, Grabowski and Swacha seeks to simplify the XML document by replacing common phrases with short codes (typically one or two bytes) before applying further compression. A preliminary pass over the file is made to build up a dictionary of common phrases - which can include start tags (but not end tags or numbers). A second pass over the XML allows those common phrases to be replaced with their

dictionary codes and also for numbers and end tags to be encoded.

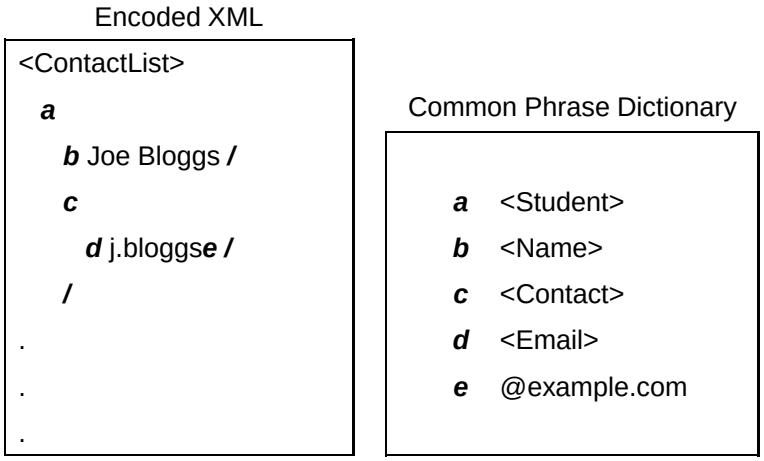

Fig. 2.12: XML Word Replacing Transform

An example XWRT encoding is shown in Figure 2.12. In this diagram common phrases, which may include start tags or data values, have been encoded and are replaced by dictionary codes depicted here by the italicised letters ***a-e***. Note that `<ContactList>` is not encoded as it appears only once in the example XML code, though the other start tags and the repeated text section "@example.com" are encoded.

As XML elements are arranged hierarchically, any end tag must correspond to the previous unclosed start tag - this allows all end tags to be represented by the same one-byte code ('`/`'). Numbers also receive special treatment, a one-byte code introduces a number sequence and records how many of the following bytes represent the value of that number. The entirety of each following byte in that number coding is given over to storing the value, i.e. the number is stored in base-256.

Having applied these changes, the transformed XML is then passed to a general purpose compressor (e.g. gzip) to complete the process.

The XML compressors described in this subsection each require the document to be decompressed back to the original XML before querying can take place. The overhead this implies can be avoided by arranging the compressed structure in

such a way that it can be directly accessed by query processors. The following two subsections describe two categories of XML compressors that allow such querying of the document without resorting to full decompression.

2.2.2 Queryable Homomorphic XML Compressors

The queryable XML compressors detailed in this section are those termed 'homomorphic', i.e. compressors that keep the structure and data values together in their compressed format. The result is compressed documents which are themselves semi-structured in the same way as the original XML documents. By maintaining this same organisation of structure and data values, these homomorphic compressors allow the same parsing and querying techniques as those employed for XML documents to be used directly upon the compressed version of the files. This is in contrast to the non-queryable compression systems described in Section 2.2.1, which must have their output files fully decompressed before querying may take place.

One of the first queryable XML compression systems proposed was Tolani and Haritsa's XGrind [TH02]. At the start, the system checks for any available XML schema. It notes any enumerated attributes, then performs a preliminary pass over the XML file to build Huffman tables[2] for values contained in each element and non-enumerated attribute. The next stage is the actual tokenisation of the XML file. In a similar way to that used in XMill, each element is recorded in the format `T1`, `T2`, etc. using a unique number for each tag type, with attribute types recorded as `A1`, `A2`, etc. Tables are kept of element and attribute names. Enumerated attribute values are then replaced with the appropriate token, while all other attribute values and data values are compressed using the appropriate Huffman table. All end tags are recorded as a single '`/`'.

Listing 2.2 shows a section of the example data from Listing 2.1 in XGrind format. In this diagram the code `huff(...)` indicates that the data value would

[2] Huffman encoding [Huf52] sees each character of the text data being replaced with a binary code, with the most frequent characters receiving the shortest codes. Codes are allocated such that, despite being of varying length, no short code can be mistaken for the start of a longer code.

```
T0
  T1
    T2 huff(Joe Bloggs) /
    T3
      T4 huff(j.bloggs@example.com) /
    /
  /
.
.
.
```

Listing 2.2: XGrind Example Output

be stored in a compressed format using the appropriate Huffman table.

XGrind is used later in this thesis for comparison purposes. A more detailed description is therefore given in Section 3.3.

The XPRESS system [MPC03] was designed to improve upon the path evaluation abilities of homomorphic compressors. The authors, Min, Park and Chung, employ reverse arithmetic coding to represent the unique paths found in each XML document. Each element type is assigned an interval between 0.0 and 1.0, these intervals are then sub-divided according to the frequency of their incoming paths. This is illustated in Figure 2.13 where `Contact` has an interval between 0.4 and 0.7. This is then sub-divided into `Student/Contact` with interval 0.4 to 0.5 and `StaffMember/Contact` (interval 0.5 to 0.7).

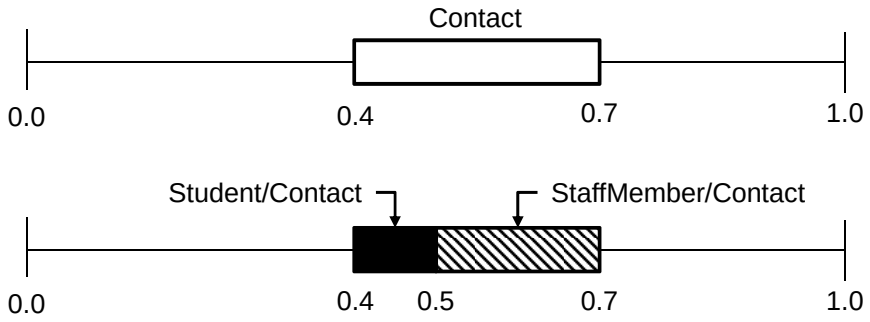

Fig. 2.13: XPRESS Reverse Arithmetic Coding

Thus to evaluate a query involving `Contact`, XPRESS can select the parts of the document contained in interval 0.4 to 0.7. For a more selective query, e.g.

`Student/Contact`, a smaller interval can be selected (in this case 0.4 to 0.5). The authors report that this system of storing path details produces significant improvements in query evaluation times over those of XGrind.

In terms of compression of data values, the XPRESS system attempts to infer the type of data contained in a particular element during a first pass over the data. During the second pass, it encodes the data values using whichever of its built-in encoders is deemed appropriate for that element type. These include three encoders for different lengths of integer, a floating point number encoder and a dictionary-based encoder for enumerated attributes. All other types of data are classed as text and are Huffman encoded.

The Query-supporting XML Transform (QXT) [SS07] proposed by Skibinski and Swacha expands upon their earlier work, the non-queryable XWRT covered in Section 2.2.1, by incorporating features that allow for querying over the compressed data. Like the authors' earlier work, QXT makes two passes over the XML document. The first pass creates a dictionary of common phrases and the second pass replaces these with the relevant tokens (an overview of the QXT process is shown in Figure 2.14). The key point is that QXT differentiates between structural parts of the document (i.e. XML start tags) and other data values. The QXT parser also recognises email and website addresses and treats them as a single phrase.

As with XWRT, end tags are encoded as a single-byte code and number sequences are binary encoded, but QXT introduces new special cases for dates, times, ranges and decimal fractions. For example, dates are stored within three bytes as the number of days since 1st January 1977 (a one byte flag plus a two byte integer), while times are recorded as a one byte flag with an additional byte each for hours and minutes.

Following the encoding of special data types and the application of the common phrase dictionary to the XML data, QXT splits the encoded document into a series of containers, compressing each with gzip (or similar) when it reaches 8MB and storing it on disk. The final QXT compressed file will consist of these containers and the common phrase dictionary. Note that although the document is split into containers, the data values are not separated from the related structural

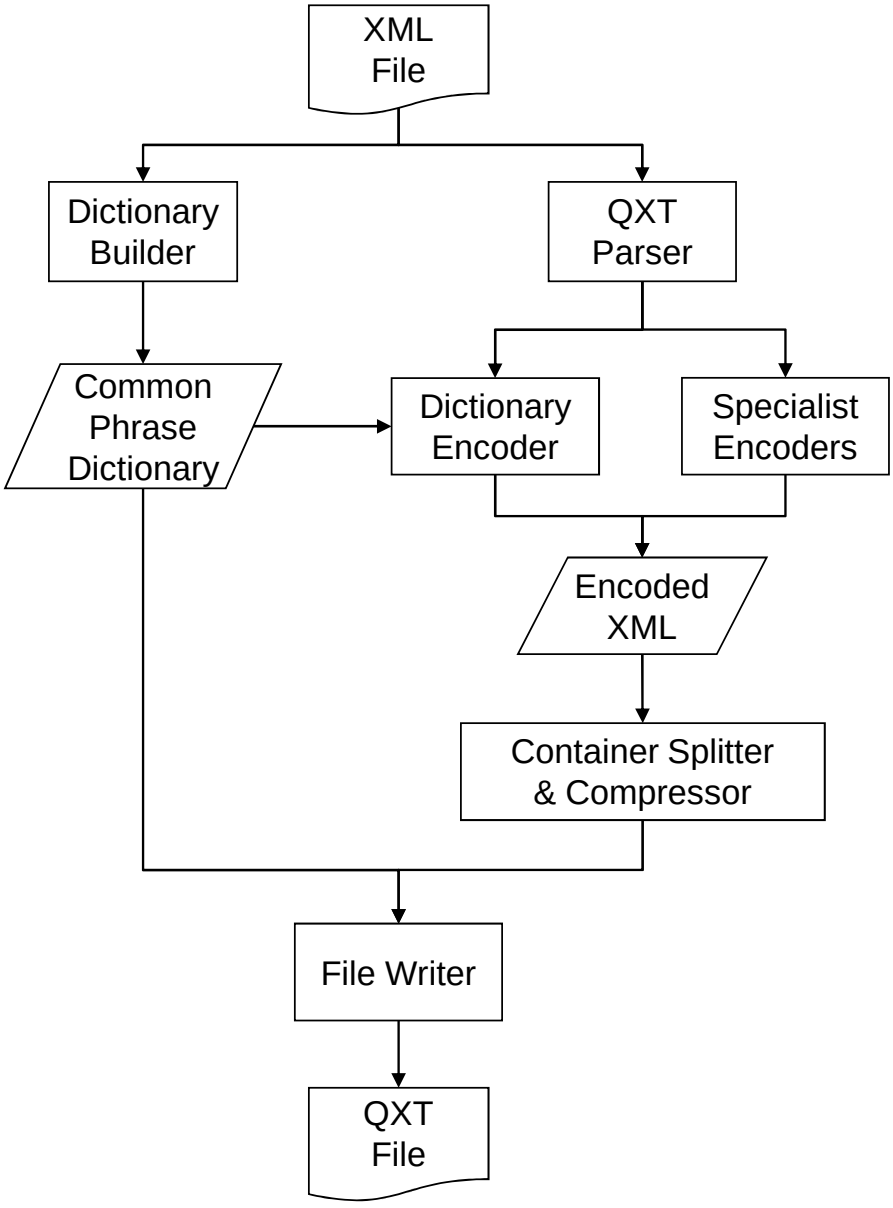

Fig. 2.14: Query-supporting XML Transform

part of the document - QXT is thus presented here as homomorphic, although it does share some characteristics with the non-homomorphic compression methods presented in the next section.

Querying the QXT data is performed by first identifying and decompressing (using gunzip or similar) the relevant containers. The second stage of querying then takes place over the encoded document sections. The final step reverses the QXT transformation and uses the common phrase dictionary and appropriate special data type decoders. This final step takes place only for those encoded elements that are found to match the query.

Queryable homomorphic XML compressors all maintain the integration of structure and data values found in the original uncompressed XML documents, albeit in an encoded form. This allows for the querying of compressed documents at the expense of a reduction in compression ratios[3] with respect to the non-queryable compression methods discussed in Section 2.2.1. The next section describes systems that offer improvements in both querying ability and compression ratios over homomorphic compressors by breaking this link between structure and data values.

2.2.3 Queryable Non-Homomorphic XML Compressors

The queryable homomorphic compression methods described in the previous section only allow querying based on top-down evaluation, i.e. the search through the data must begin at the root of the XML tree and work down through the structure. This is a consequence of keeping data and structure together as in the original XML document and relying upon traditional XML parsing and querying techniques. For more complex queries, such as those containing joins, this can quickly become cumbersome. The compression methods described below overcome this issue by returning to the system of storage favoured by some of the non-queryable compression methods detailed in Section 2.2.1 - splitting the document structure from the data values it contains. This non-homomorphic approach not only allows the use of query strategies other than top-down, but also gives

[3] XGrind achieves on average 77% of the compression offered by XMill. [TH02]

scope for improvements in compression ratios.

One of the earliest such systems is XQueC, proposed by Arion et al. [ABC+04], which follows a XMill-style method for splitting the data values into containers based on their type and path. The structure of the document is recorded by XQueC as a series of node records (storing the node ID, those of its child nodes, its parent node and a code for the type of node) in a B+ search tree along with a table of element and attribute name codes. Data containers consist of a record for each data value along with links to the relevant nodes in the structure tree.

The data value records are stored in alphanumeric order by value, not the order they appear in the document, to allow bottom-up access to the data. Where appropriate, an order-preserving string compression (ALM [Ant97]) is used such that where any two uncompressed values can be compared, the compressed versions can also be compared to achieve the same result. This means that inequality comparisons can be evaluated without having to decompress the values.

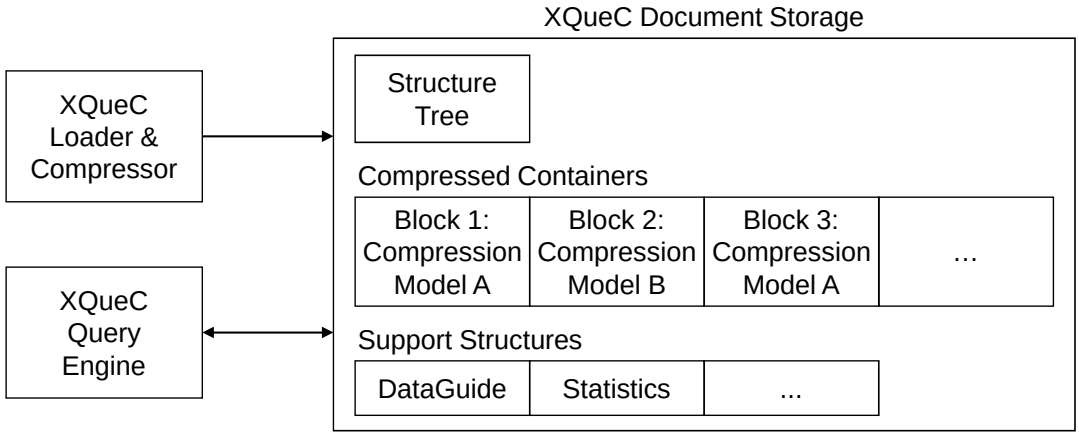

Fig. 2.15: XQueC

Compression models are applied on a per container basis with each value within a container being individually compressed using the same model. XQueC applies a cost function to see if it can apply the same compression model to more than one container without damaging the compression ratio. This can be seen in Figure 2.15, which shows a simplified version of the XQueC architecture, where both Block 1 and Block 3 each use Compression Model A. The diagram also

shows that additional support structures such as DataGuides are employed by XQueC to speed access to the data.

XQzip by Cheng and Ng [CN04] draws upon concepts from the XML structural summarisation methods described in Section 2.1 to form a Structural Index Tree (SIT) in a similar style to the F&B-Index discussed earlier. However, unlike the F&B-Index, the SIT retains the ordering of the data in the form of pointers stored in each SIT node. These pointers indicate the node's parent, previous sibling, next sibling and first child nodes. As in the F&B-Index, duplicate nodes from the XML tree are removed, however the SIT maintains a record of these within the nodes it retains.

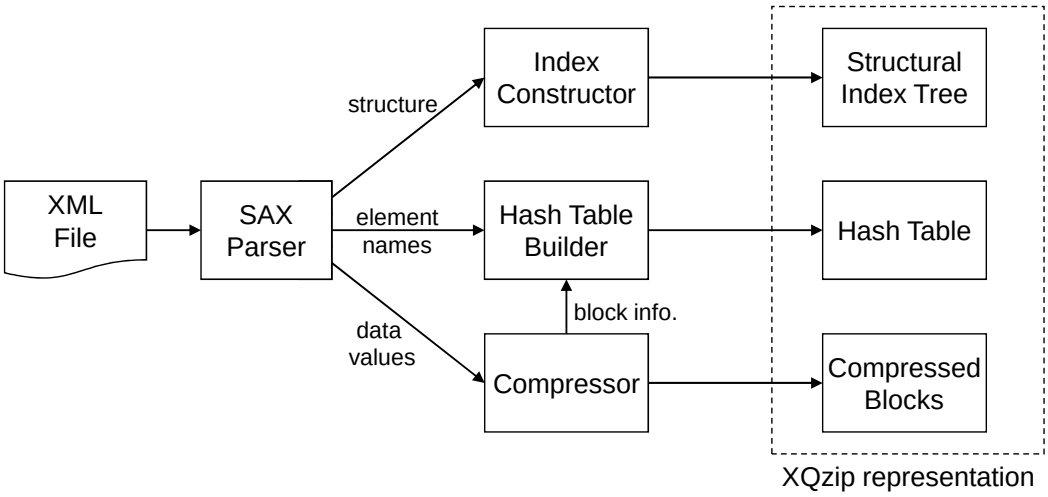

Fig. 2.16: XQzip

As shown in Figure 2.16 an XML document is read into XQzip through a SAX Parser, which passes the structural part to the Index Constructor to create the SIT. The element names are passed to the Hash Table Builder. Data values are split into streams according to type and path, then chopped into blocks of 1000 values (or 2Mb maximum size) and compressed at a block level using gzip. Compressed data block identifiers are stored in the hash table along with some additional details (e.g. start address and size).

Querying is performed initially on the Structural Index Tree with data blocks being accessed through the hash table. The authors contend that compression of

data values at a block level is advantageous as this requires a single decompress operation to allow evaluation of 1000 values. Decompress operations are further reduced in XQzip by keeping the last few decompressed blocks in a buffer pool to allow fast access where subsequent queries need to access the same data blocks.

An alternative method of speeding the query process can be found in the XCQ system proposed by Ng, Lam, Wood and Levene [NLWL06]. XCQ stores the document making use of the XML schema in conjuction with the SAX Parser output to produce a set of path-based data streams and an additional stream containing structural elements. As shown in Figure 2.17, the data streams are subsequently split into blocks containing a set number of data values and compressed by gzip.

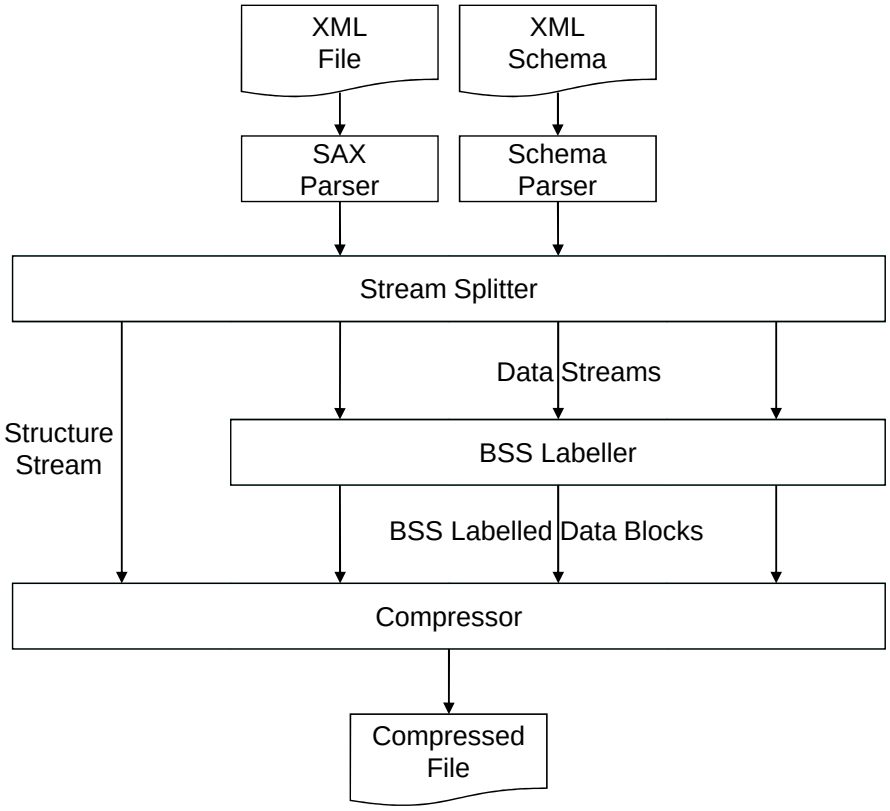

Fig. 2.17: XCQ

The factor that allows the efficient retrieval of data from these compressed blocks is that each one is labelled with a Block Statistics Signature (BSS), con-

taining information about the maximum, minimum and number of values stored within that block. In this way, when XCQ is looking for a particular value in a data stream, blocks whose BSS does not meet the criteria can be quickly filtered out of the search.

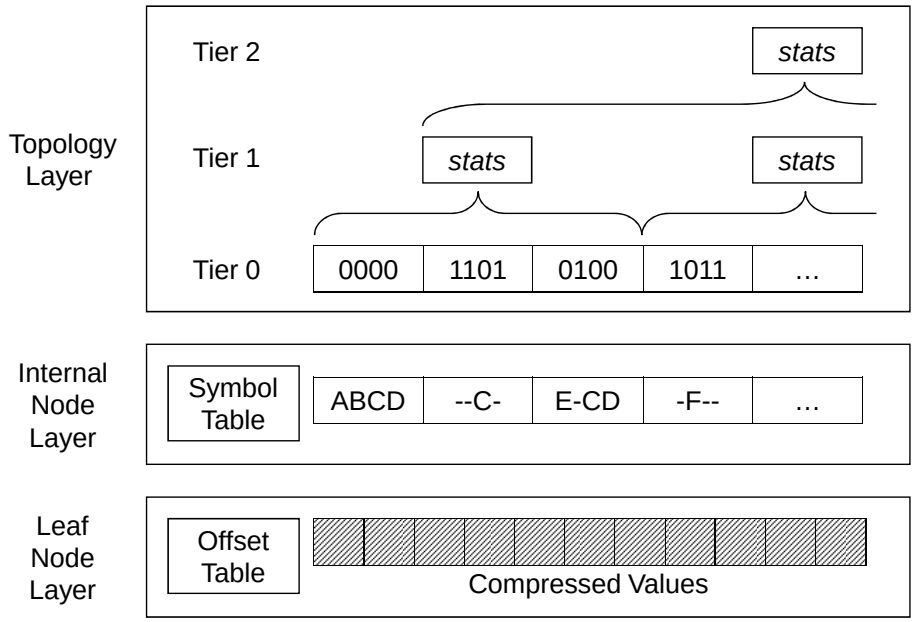

Fig. 2.18: ISX

A novel approach to improving on tree traversal speeds is proposed by Wong, Lam and Shui [WLS07]. The ISX system introduces a three-layer model, shown in Figure 2.18, as opposed to the two-layer data/structure split used by the previous methods discussed.

The Topology Layer holds a balanced parenthesis representation of the XML structure. To represent the parentheses in the lowest tier of the Topology Layer (T0) a `0` is used to record the start of any XML element, attribute or text value, with a balancing `1` denoting the end of any of these. This Topology Layer is employed during the querying process to quickly traverse the XML structure. For example, finding the next sibling node can be performed by comparing the number of `0`s and `1`s encountered. Auxilliary tiers of the Topology Layer (T1 and

T2 in the diagram) hold summary statistics of groups of entries in the lower tiers to further speed access.

ISX stores element and attribute names in tokenised form in the Internal Node Layer, which follows the format of the lowest tier of the Topology Layer. The Internal Node Layer also holds summary hash values for any data values that are stored, compressed by gzip, in the Leaf Node Layer.

The queryable non-homomorphic compression methods described above are distinct from the methods discussed in the previous section in that they each separate the data values from the document structure. In doing so, query systems are freed from the restrictive top-down evaluation methods adopted by homomorphic systems from the querying of uncompressed XML. The departure from this linear representation of the structure also makes way for improved compression ratios[4] over homomorphic systems.

2.3 Comparison of Existing Methods

Table 2.1 summarises the key characteristics of the preceeding methods. The first of these is whether the method is queryable - as discussed earlier, this dictates whether the system is suitable for data access or simply archival purposes. The second characteristic is whether an XML schema is used by the system to improve compression. The following two columns record the underlying compression method used to store data values and the smallest unit that must be decompressed to access a single value, with the penultimate column noting if the compression method allows the evaluation of inequality operations without final decompression of the individual values.

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 relates 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,

[4] On average XQzip offers 16.7% more compression than XGrind. [CN04]

	Queryable	XML Schema	Backend Compressor	Decompression Unit	Inequality without Decompression	Comment
XMill	No	No	gzip	Full	N/A	User must specify groupings and compressors to achieve claimed level of compression.
XMLPPM	No	No	PPM	Full	N/A	Statistical modelling allows better compression than default XMill, but maintaining four models is slow.
SCA	No	Req.	gzip	Full	N/A	Slower and less effective than XMill. Cannot work without XML schema.
XWRT	No	No	gzip	Full	N/A	Can improve compression, but must fully adjust parameters.
XGrind	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	Yes	No	Huffman/ dictionary	Value	No	Improves querying over XGrind - no need for linear parse of document. Compression still considerably worse than XMill.
QXT	Yes	No	gzip	Container	No	Compressed size is design priority. Must unzip full container before any examination of transformed contents.
XQueC	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	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	Yes	Req.	gzip	Block	Part.	Stores less structure so smaller compressed files, but requires schema to do so.
ISX	Yes	Opt.	gzip	Block	No	Emphasis on traversal of structural part.

Tab. 2.1: Comparison of Existing XML Compressors

reducing the decompression workload.

Of the methods which take account of XML schema it must be noted that, while this can be used to improve compression, without access to the schema neither XGrind nor ISX will achieve their best compression and both SCA and XCQ would be unable to process the XML document at all. Given that not every document will have a schema available this is a considerable drawback to these methods.

One desirable feature of some of these compressors is the ability to evaluate inequalities without decompression of the 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, but thereafter the matching blocks must be decompressed in their entirety to complete the query. The full version of this feature is exhibited in XQueC where the order-preserving nature of ALM compression allows comparison of the individual compressed values.

However, this depends on the selection of ALM as the backend compressor which depends on additional user input in the form of advance knowledge of the query workloads. This additional dependency on the user is undesirable but is also seen in XMill and XWRT which require the user to manually set a number of parameters to achieve their claimed compression.

Although each of these methods offer XML compression and each has its own strengths and weaknesses, the important factor is that each of these existing methods have been designed with regard to the compression of data, not the sharing of it. Regardless of the amount of compressed data that must be accessed to answer a query, each of the methods above require the entire compressed document to be transferred as a single unit before any querying may take place making them unsuitable for the sharing of independent segments of data.

2.4 Summary

This chapter has reviewed notable advances in XML processing, beginning with indexing and structural summarisation techniques that seek to augment the ex-

isting XML with a view to improving querying. The chapter then considered methods concerned with the compression of XML with a view to archival (non-queryable compressors). Queryable homomorphic techniques keep data and structure together to combine compression with traditional XML parsing and querying abilities, while non-homomorphic compressors draw upon concepts from XML structural summarisation to split data and structure, while improving both querying and compression. However, despite the advances of these systems in terms of compression and querying, none of the compressors described in this chapter are designed to handle data structures that have been split into parts for sharing.

As noted at the start of the chapter, these methods are representative of a broad field and are not presented as an exhaustive list. The systems listed are those commonly referenced in the literature or which propose interesting or unusual ideas.

The next chapter will review in more detail the technologies that will be made use of later in this thesis.

3. RELEVANT TECHNOLOGIES

The previous chapter provides a general overview of research related to the experiments reported in Chapter 4. This chapter introduces in detail the technologies that are used or built upon in the experiments set out in the next chapter. In the work to be described, the structural aspects of data are dealt with using summarisation provided by the NSIndex [GTW07] system and the data value compression is based upon prior work on the HiBase [CMW98] relational compression system. Further description is also given of XGrind [TH02], a queryable XML compressor used for comparison purposes in the initial set of experiments.

3.1 NSIndex

NSIndex is a data structure designed with the restrictions associated with mobile devices in mind. It comprises of a structural summarisation of the XML combined with a numbering scheme linking to the data values. The original NSGraph [WGJN06b, WGJN06a] was a main memory-based system that combined a fast index with a compact array-based representation of the XML data. This was subsequently extended to become NSIndex, which retains data values as part of its internal representation of the XML structure (rather than in separate arrays) and can be written to disk.

The bisimilarity-based structural summarisation used in NSIndex 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. A bisimilarity-based partitioning of the XML takes the surrounding structure into account, meaning that similar data items will fall into the same partitions, as opposed to slicing the data based on a physical attribute such as the level at which it sits in the XML

tree. Additionally, it is reasonable to assume that where data of a similar kind is grouped together, there is a likelihood that the actual data values in each group will contain duplicates. This means that there is potential to extend NSIndex, which has bisimilarity at its core, with a system of dictionary-based compression to exploit duplication.

3.1.1 NSIndex Compression

The partitioning used is based around the Forward and Backward Index (as discussed in Chapter 2.1) 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 NSIndex structure is supplemented by a numbering scheme, based on the work of Dietz [Die82], which maintains the ordering of the data throughout.

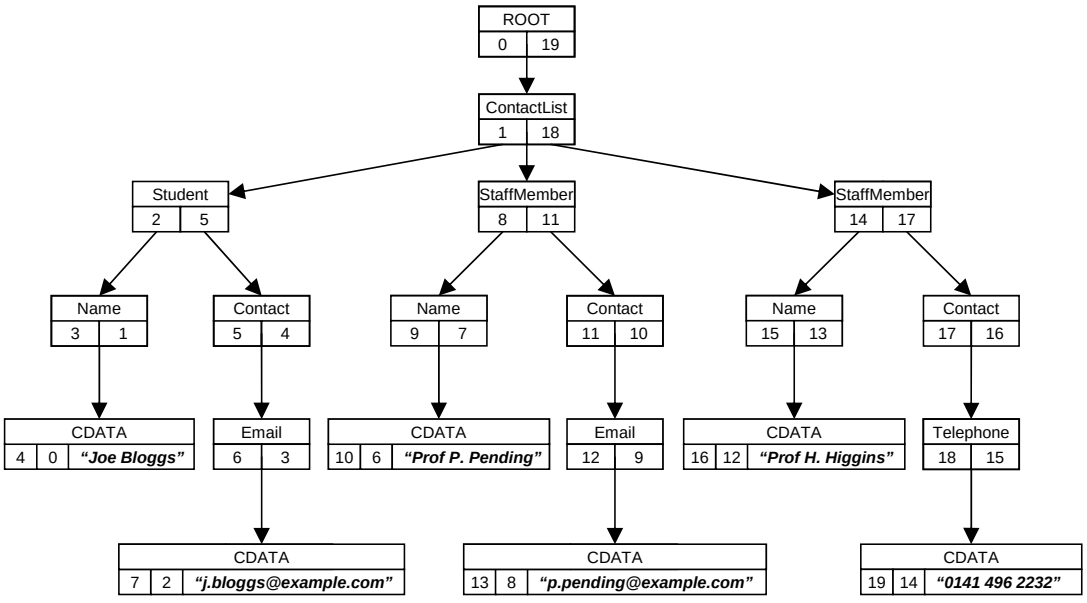

Fig. 3.1: NSIndex Using Forwards and Backwards Bisimilarity

Figure 3.1 shows a representation of the example XML introduced in Listing 2.1 in the NSIndex format. The structure is set out in terms of vertices (the large

boxes) and edges (the connecting lines). Each vertex may hold either structural information or data values.

In either case, vertices are comprised of NSEntries each representing a datagraph node as 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 depicted in the diagram as a set of two numbers as supplied by the numbering scheme. The first is the entry's 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. The second is the entry's 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.

Note that a special vertex `ROOT` appears at the top of the diagram and holds both the first pre-order number (`0`) and the last post-order number (`19`). A `ROOT` vertex appears in every NSIndex to give a clear starting point for the structure - the first tag of the XML document is always directly connected to `ROOT`.

Also recorded in each vertex entry, though not shown in the diagram, is the level at which the entry appears within the NSIndex structure and the size of the entry, including any sub-trees which appear beneath it in the structure. For example, the `Contact` entry `(17,16)` shown at the right-hand side of Figure 3.1 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,"0141 496 2232"`. 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 NSDataEntry (`19,14,"0141 496 2232"`).

3.1.2 Program Operation

The NSIndex architecture used to produce this structure is shown in Figure 3.2. The XML document is read in through a SAX Parser and a datagraph constructed. This is then used to produce the F&B-Index-based structural summarisation using one of four methods: 1) group by label only, ancestor and de-

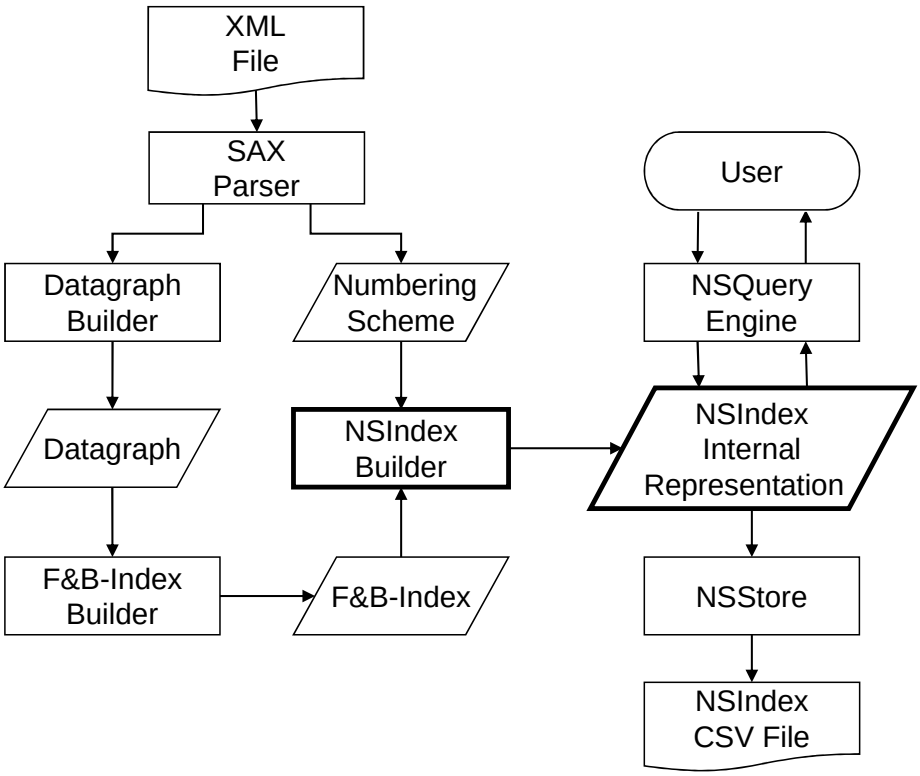

Fig. 3.2: NSIndex System Architecture

scendant nodes are not examined. 2) full backwards bisimilarity, repeatedly apply bisimilarity to incoming paths until a stable grouping is reached. 3) full forwards bisimilarity, find stable grouping by examining outgoing paths. 4) F&B-Index, using both forwards and backwards bisimilarity. At all points throughout this process an ordering (i.e. the pre-order number) is kept for each XML element. This is now used, along with the bisimilarity information from the F&B-Index, to construct the NSIndex.

Following this, a queryable NSIndex structure will exist in the computer's memory. Querying is performed as a two-stage process. First, all of the vertices within the NSIndex structure that hold the correct type of entry for each part of the query are identified. Secondly, the relationships between the individual entries are verified using the numbering scheme information. For example, a query to check the example data for the existence of `ContactList/Student/Contact/Email` would select the `ContactList` vertex, the `Student` vertex, all three `Contact` vertices and both `Email` vertices. It can then be identified that the query is answered by the sequence of data entries: `ContactList (1,18)`, `Student (2,5)`, `Contact (5,4)`, `Email (6,3)` as the numbering scheme shows that connections exist between these data entries (as indicated in this case by the sequence of pre-order values `1, 2, 3, 4, 5, 6`). The additional `Contact` and `Email` entries selected in the first stage are discarded as the numbering scheme shows they do not connect to a `Student` entry.

The NSIndex system is extended by the NSStore module, which allows the NSIndex structure to be written to disk. NSStore traverses the NSIndex structure outputting one complete vertex at a time starting at ROOT. The file is created in comma separated value (CSV) format as shown by the example in Listing 3.1. 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 sequence of four values: pre-order, post-order, level and size[1], with the data value additionally being recorded for data entries.

[1] NSIndex adds 1 to the expected size of any entry with descendants.

```
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,''Joe Bloggs''
Contact
5,4,4,4
Email
6,3,5,3
CDATA
7,2,5,1,''j.bloggs@example.com''
.
.
.
```

Listing 3.1: NSIndex File Format

3.1.3 Section Summary

The use of bisimilarity-based data partitioning makes NSIndex a suitable platform on which to explore the use of dictionary-based data storage and to investigate the effects of different combinations of bisimilarity methods upon the data that is stored. As part of the experimental work reported in Chapter 4, the existing codebase was extended to incorporate data value compression along with the necessary changes to allow continued access to the stored data. The adapted NSIndex system was then used in the experiments set out in Chapter 4.

3.2 HiBase

Partitioning provides a means of subdividing XML data, which is related to the concept of domain[2] in the relational database world. The concept of domain is exploited by compressed relational systems, which apply data dictionaries to encode data values, for example HiBase [CMW98], Chen et al. [CGK01] and C-Store [AMF06]. However, dictionary compression is only one of a number of methods used in C-Store and is used as part of a hybrid system in the work by Chen et al. As HiBase makes exclusive use of dictionary compression, it is consequently selected as a starting point for the work currently described.

[2] The set of potential data values an element may contain.

The HiBase system was developed to permit traditional relational databases, ordinarily stored on disk, to be held within main memory and take advantage of the reduced access times that follow from this.

Compression is applied on a per column basis, with each column containing all the values for one attribute of the relational table. As each column contains data of a particular domain, HiBase employs a dictionary-based compression system on a one dictionary per column basis to take advantage of repetition within the data values.

3.2.1 HiBase Compression/Operation

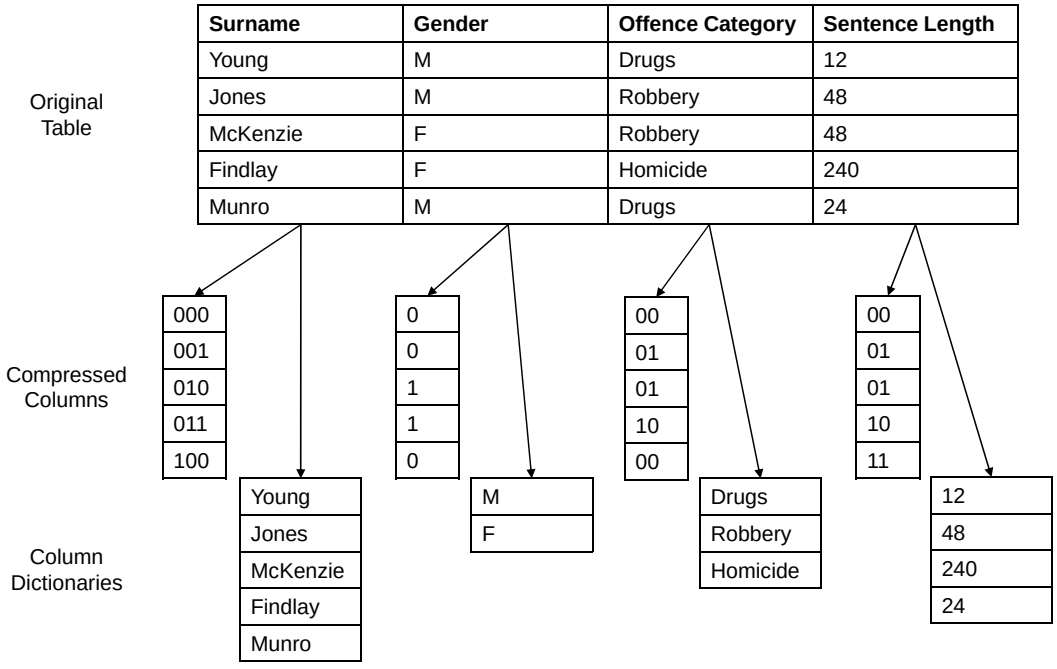

Fig. 3.3: HiBase Compressed Columns and Dictionaries

As noted above and shown in Figure 3.3, HiBase applies compression on a per column basis. A dictionary of unique terms is drawn up for each column of the database table. These dictionaries are then used to produce compressed representations of each column. The tokens (binary elements that indicate dictionary terms) used within each individual column are of uniform length to allow straightforward addressing of the values, with each column using the fewest bits

possible to cover the associated dictionary. For example, the two distinct values contained in the `Gender` column of Figure 3.3 can be represented using a single bit token, the `Offence Category` and `Sentence Length` columns (with three and four distinct values respectively) each use a two-bit token, while `Surname` requires a three-bit token to cover five distinct values.

While each tokenised value in the compressed columns is of uniform length, the original data values held in the column dictionaries are of variable length. The HiBase dictionaries therefore consist of two parts: a string heap, a chunk of text containing all the unique data values one after another, and an offset table that notes where each value begins in the string heap.

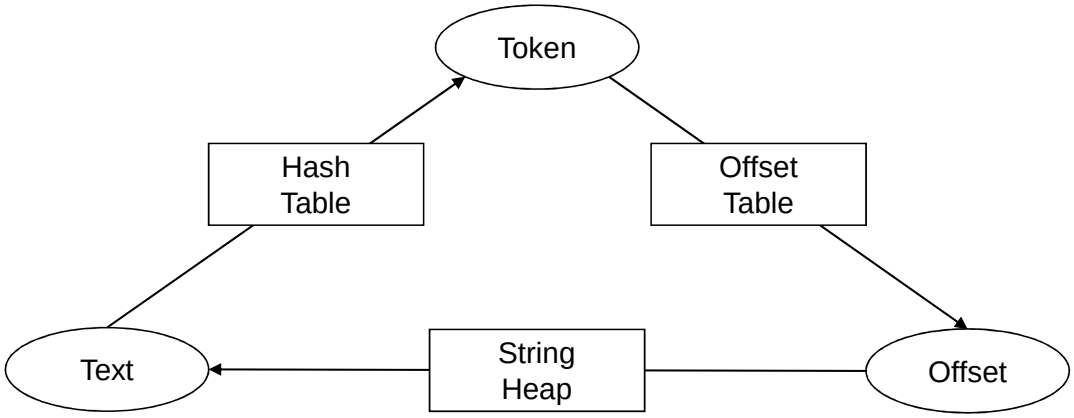

Fig. 3.4: HiBase Text/Token Conversion

Figure 3.4 shows that to convert a token to text the offset table is consulted to obtain the offset address, this in turn is used to retrieve the correct text value from the string heap. The system also maintains a hash table to allow text values to be converted back to tokens, thus making the process cyclical. This is used during the querying process: the query terms are themselves compressed and HiBase then makes use of a number of access support structures to quickly locate these terms within the compressed column structure. Only the matching results are retrieved and decompressed.

The authors report that data stored in the HiBase compressed database is between one-tenth and one-quarter of the size of same data under a conventional

relational database system.

3.2.2 Section Summary

The dictionary-based data compression used by HiBase is similar to the approach that it is proposed to add to NSIndex. The existing HiBase system is used in the first experiment of Chapter 4 to provide evidence of the compression achievable using such dictionary methods in the context of values contained in XML structures.

3.3 XGrind

An early queryable XML compressor, XGrind was proposed as a system that not only saves on storage space but, by maintaining the same layout as the original XML document, allows access to the compressed data without resort to full decompression. XGrind makes some use of dictionary-like symbol tables, but is largely built around Huffman encoding - most data is compressed as text on a per character basis. This is in contrast to the one token per data value style of compression used in the HiBase dictionaries.

Chapter 2 identified three compressors which make use of Huffman encoding - XGrind [TH02], XPRESS [MPC03] and XQueC [ABC$^+$04]. Of these, XQueC makes selective use of different compression methods depending on the query workload and XPRESS is designed to improve upon the query execution of XGrind, not its compression level. Given this, and its profile in the literature, XGrind is used in the first experiment of Chapter 4 to provide a comparison between character-based compression (XGrind) and dictionary-based compression (HiBase).

3.3.1 XGrind Compression/Operation

The basic description of the compression used by XGrind was given in Section 2.2.2, the following gives more detail on the flow through the XGrind system as shown in Figure 3.5.

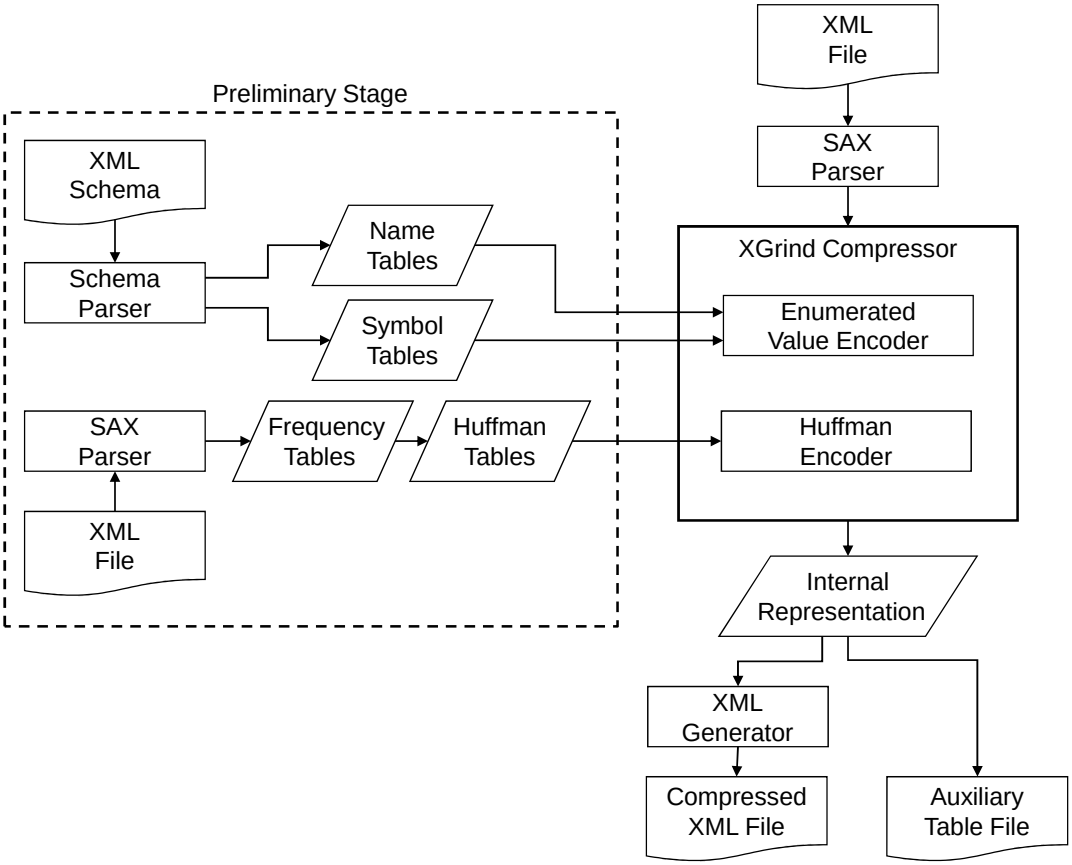

Fig. 3.5: XGrind Operation

The preliminary stage sees XGrind examining the XML Schema (if present) to identify any enumerated attribute types. If any are found, the system builds a symbol table to provide a token for each possible attribute value. The system additionally creates tables for element and attribute names. A first pass is then made over the XML document with frequency information being recorded for the individual characters that make up the data values contained in each non-enumerated attribute and element. These frequency tables are used to produce Huffman coding tables, with the most frequently used characters in each frequency table receiving the shortest codes in the corresponding Huffman table.

With this preliminary stage complete, the actual XGrind compression takes place during the second pass through the XML document. The enumerated value encoder uses the element and attribute name tables to encode start tags and attribute names (in the formats "`T1, T2, ...`" and "`A1, A2, ...`" respectively) and employs the symbol tables created earlier to tokenise any enumerated attribute values. The remaining textual data (the non-enumerated data values) are dealt with by the system's Huffman compressor using the coding tables produced from the character frequency information in the preliminary stage.

The resulting internal XGrind representation of the data is stored on disk in two parts. An XML generator is used to produce the main compressed XML document (saved as a .xgr file) while the Huffman, symbol and element/attribute name tables necessary for decompression are saved as a separate auxiliary (.met) file.

The XGrind query system, having previously loaded the appropriate auxiliary tables, compresses both the path and values involved in the query. The compressed XML document is then parsed to identify those parts of the document that match the query path, the encoded data values stored at those locations are then compared to the encoded query values. Only the results that answer the query are fully decompressed to return to the user.

3.3.2 Section Summary

As one of the earliest queryable XML compressors, XGrind is consequently one of the best known and most referenced in the literature for comparison purposes. In the first experiment of Chapter 4 the XGrind system is used to provide a comparison with dictionary-based methods as typified by HiBase.

3.4 Summary

The technologies described above are employed during the experimental work of this thesis. HiBase is a useful example of the kind of dictionary-based compression proposed to extend the structural summarisation provided by NSIndex, while a comparison with XGrind provides a useful insight into the way that dictionary-based compressors perform in comparison with Huffman-based compressors. The following chapter now gives details of the experimental work.

4. EXPERIMENTAL WORK

The purpose of this thesis is to report experiments that evaluate the potential for compression of data values in compressed XML structures. This chapter briefly introduces the data sets used in this evaluation and describes the experiments conducted using them.

4.1 Overview

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.

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. Data partitioning is achieved through the use of bisimilarity to group similar items of data. As direct access to individual values removes the need to decompress more than the required values for any given query, the choice must be made between dictionary-based compression and character-based compression. To this end the first section of experimental work (Section 4.3) compares the dictionary method against Huffman encoding using both real-world and benchmark data. The second set of

experiments (Section 4.4) then makes use of the chosen compression method in evaluating the effects of four different bisimilarity-based partitioning schemes on compressed data sizes and the number of partitions created.

Having selected the type of bisimilarity and the method of data value compression to be used, Section 4.5 notes the manner in which data value compression was added to the bisimilarity-based NSIndex system. Section 4.6 then describes the changes made to the querying system to handle compressed values and how the query strategy was amended in order to reduce the number of data segments processed in answering a query.

Finally, with a view to managing the volume of dictionaries produced, Section 4.7 sets out a methodical set of comparisons designed to identify and remove duplication within the set of dictionaries.

4.2 Data Sets

The experiments described later in this chapter make use of a number of data sets selected to represent a range of both real world and benchmark data. These data sets are categorised according to the origin of their data and the regularity of their structure.

With regard to data origin, each data set is described as either "real world" or "benchmark", this distinguishes between generated benchmark data sets and those taken from real world sources. In terms of structure, each data set either has a rigid pattern of elements and sub-elements ("regular") or takes a more flexible approach ("irregular").

	Regular	Irregular
Benchmark	Orders Modified Orders	XMark
Real World	Legal	NASA Medline Dream Rat Human

Tab. 4.1: Categorisation of Test Data Sets

Table 4.1 summarises the categorisation of the data sets. Of the three benchmark data sets, Orders and Modified Orders have regular structure with XMark being irregular, while for the real world data sets only Legal has a regular structure, the remaining data sets (NASA, Medline, Dream, Rat and Human) each having irregular structure. Additional detail on the data sets can be found in Appendix A.

4.3 Preliminary Work

To facilitate efficient access to data within a compressed semi-structured system, it must be possible to extract individual data values from the system without the need to first decompress the entire structure. This implies that the stored values must be encoded using either dictionary based methods (at a per value level) or using text compression methods (at a per character level).

The two technologies selected in Chapter 3 provided examples of each of these methods. HiBase (Section 3.2) stores data values using dictionary tables while XGrind (Section 3.3) employs Huffman coding.

To form a comparison of these compression methods as implemented by the exemplar programs, test data series were compressed using each system. Since HiBase is a complete working system with additional support data structures to speed up querying, a theoretical compressed size was also calculated. This would reflect the actual space required to store the compressed data values and associated dictionaries, without including any of the overheads introduced by HiBase.

The theoretical size is calculated as shown in Calculation 1.

Token Size = $\log_2$(no of unique values in group)
Dictionary Size = (token size * no of entries) + Σ(uncompressed entry size)
Compressed Data Size = no of records * Σ(size of tokens in record)
Total Theoretical Size = compressed data size + Σ(dictionary size)

Calculation 1: Theoretical Size

For this initial work, the Orders data series and the Legal data series were

selected as representative of benchmark and real world data respectively. To see the effect of the generated text values in Orders, the Modified Orders data series was also used. These data sets have been selected as their regular structure allows simple conversion to the comma separated format required by the column-based HiBase without the need to pad the data with null values as would be necessary with an irregular data structure.

To summarise, each test data set was processed using the three methods shown in Table 4.2:

HiBase	The XML markup is stripped and the data values passed to Hibase as columns in CSV format. Compressed data size is taken to be the overall memory requirement reported by Hibase.
XGrind	The XML file is processed by XGrind. Compressed data size is taken to be that of the two files generated by the program.
Theoretical	The XML markup is stripped and text files of data values produced. Compressed data size is calculated using values taken from these files.

Tab. 4.2: Summary of Processing Methods for Preliminary Work

The results of this experimental section provide indications as to whether minimal-token dictionaries are a suitable option for handling data values in the extended NSIndex system.

4.4 Evaluation of Partitioning Methods

The structural summarisation produced by NSIndex is dependent on the type of bisimilarity used during the partitioning process. Earlier work [GTW07] has shown that varying the bisimilarity used has an effect on the number of edges and vertices that constitute the NSIndex structure. It follows from this that as the structure changes, the groupings of data values at the bottom of the structure will also change. This section of work evaluates the effect of changing the partitioning scheme on the number and size of data dictionaries required.

A selection of data sets from each of the categories described in Section 4.2 are used in this experiment. From the randomly generated benchmark data sets, the

irregularly structured XMark dataset (10Mb and 30Mb sizes, hereafter XMark-10 and XMark-30) and the regularly structured Orders and Modified Orders (1000 and 15000 orders: Orders-1, Orders-15, ModifiedOrders-1 and ModifiedOrders-15) are used. The real world, regularly structured, Legal data set is used (1000 and 13000 convictions: Legal-1 and Legal-13) as are the five real world, irregular data sets: Dream, Medline, NASA, Rat and Human.

A version of the NSIndex program with the ability to output a summarised structure to a comma-separated file was used to process each data set using each of the four available bisimilarity options as outlined in Section 3.1.2: no bisimilarity (data entries grouped by label only), full backwards bisimilarity, full forwards bisimilarity and full F&B-Index (using both varieties of bisimilarity).

The data values from each data vertex were then extracted from the CSV file using a simple parser. In their initial form as extracted from the CSV file, the set of uncompressed data vertices produced for each partitioning scheme will be the same size since, regardless of their distribution, the complete set of data values will appear across each set of uncompressed vertices.

However, changing the type of bisimilarity used will have an effect on the compressed size. Changing the partitioning has an effect on the number of data vertices that the data values are distributed between at the bottom of the NSIndex structure. 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 4.1 using the data values `A`, `A`, `B`, `B`, `B`. If, as shown on the left of the diagram, 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 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`.

However, if as on the right of Figure 4.1 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,

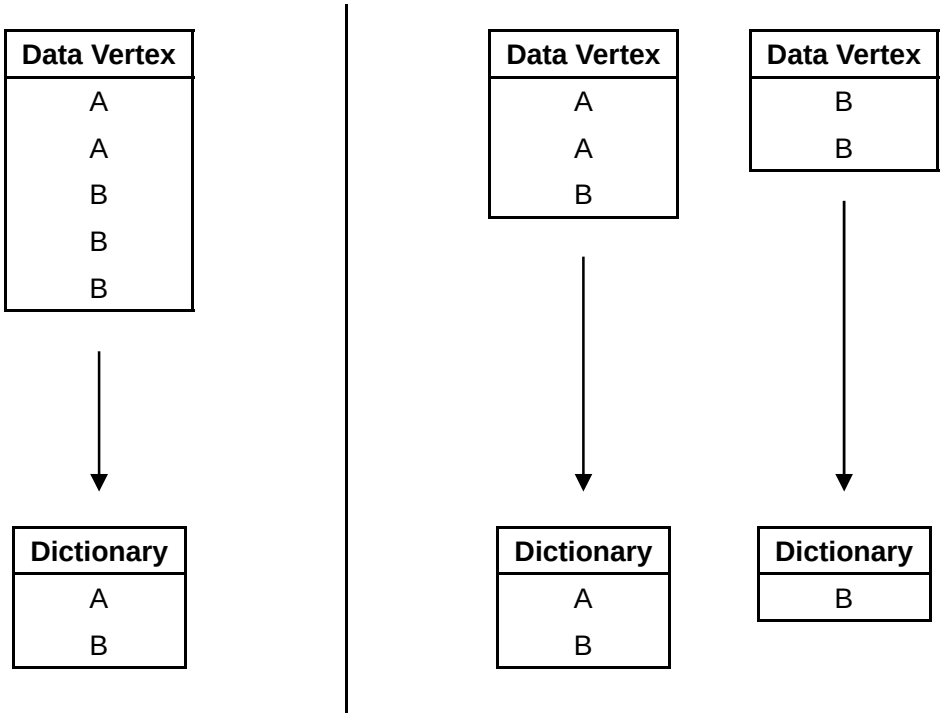

Fig. 4.1: Effect of Data Value Distribution

increasing the overall dictionary size by 50%. This could have considerable effect on the overall size of dictionaries, especially when longer data values are involved.

By processing these data vertex files, a set of associated data dictionaries containing only the unique entries from each vertex can be created. This process was applied to each of the four differently-partitioned sets of data vertices for each data set. In each case, the size of the compressed data was calculated in the same way as in Section 4.3 using the values taken from the processed files.

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. These results are shown in Chapter 5.2.

4.5 Integration of Data Value Compression

A number of changes and additions to the existing code were required to implement data value compression within NSIndex. The following two subsections note the implementation of dictionary creation and data encoding and a method for loading previously created NSIndex structures back into the system.

4.5.1 Dictionary Creation and Data Encoding

During the partitioning process, each data vertex in the modified NSIndex system maintains a list of the unique data values it contains (in addition to the full (pre, post, level, size, value) details held for each data entry). As the NSIndex structure is written out to file, these lists become the data dictionaries - each one is written out to a separate file as it is encountered during the traversal of the structure and is allocated the next available dictionary number. The dictionary values are sorted before they are output so that the tokens generated are order preserving (i.e. the ordered list of tokens will appear in the same order as the ordered list of values they represent).

Once the dictionary for a data vertex has been written to file, the data vertex itself can be added to the NSIndex CSV file. The dictionary number forms part of the dictionary filename and is recorded as part of the data vertex header line. The data values contained within the individual data entries are encoded according to the dictionary associated with that data vertex as they are written to the CSV file. An updated version of the CSV file shown in Listing 3.1 incorporating dictionary references and encoded data values is given in Listing 4.1.

4.5.2 File Loading

While the unmodified NSIndex had the function of outputting the structure to a CSV file, there was no equivalent method of loading this file back into NSIndex for querying. A parser was written to process the CSV file and read the stored NSIndex strucure back into memory.

The role of each line of the file (Listing 4.1) can be derived from the number

```
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 4.1: NSIndex Compressed File Format

of tokens on that line: a single token on a line indicates the start of a structural vertex, while four tokens represent an element stored in a structural vertex. More than four tokens indicate an element stored in a data vertex. The fifth token is the data value but there may be additional tokens if that value contains commas. In such a case these additional tokens must be concatenated to the fifth. A special case is any line starting with the token "CDATA", this marks the start of a data vertex with the remainder of the line being the associated dictionary filename. Algorithm 2 gives a more complete overview of the process.

When the NSIndex structure is loaded from the 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 by NSIndex, which permits multiple vertices to refer to the same dictionary (as described later in Section 4.7).

4.6 Querying

None of the pre-existing querying strategies for NSIndex were designed with compressed data values in mind. The introduction of compressed data values meant these querying strategies could no longer access the data values in the manner previously used. In attempting to rectify this it became apparent that simply

```
Read first line.
while Line not empty do
    if first token on line is "CDATA" then
        Put previous vertex on stack.
        Get dictionary details.
        if dictionary not loaded then
            Load dictionary.
        end
        Make new data vertex and add it to structure.
        Add edge (pop from vertex stack until parent is found).
    else if only one token on line then
        if previous vertex was structural then
            Put on vertex stack.
        end
        Make new structural vertex and add it to structure.
        if this is not root of structure then
            Add edge (pop from vertex stack until parent is found).
        end
    else if four tokens on line then
        Add entry to current structural vertex.
    else if more than four tokens on line then
        Add entry to current data vertex.
    end
    Read next line.
end
```

Algorithm 2: File Loading

adding the facility to decompress the values was not a good solution. The nature of the existing query processing strategies meant that this would lead to a large number of dictionary decode actions and consequent poor query performance. For example, the top-down query strategy selected 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 on the left of Table 4.3 based on the query `/A/B/"Data"`.

NSIndex Native Approach	Compressed NSIndex 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

Tab. 4.3: Query Example

The right-hand side of Table 4.3 shows an adaptation of the old 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 native 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. The matchDescendants method, which sits at the centre of the compressed query strategy, is explained in Algorithm 3.

```
input: Candidates vertices (those which match the start of the query).
if Query term is a leaf of the query tree then
    if Query term is a data value then
        Return matching data entries.
    else
        Return all data value results.
    end
else
    Note individual data entries from candidate vertices (used later).
    Find descendants of candidates which match next query term.
    Recursive call to matchDescendants method, passing current
    descendant list as the candidate list.

    (At this point have reached bottom of query and working backwards.)
    (Now looking at individual data entries, not vertices.)

    Retain data entries from candidates where a descendant has been
    returned.
end
Return final result.
```

Algorithm 3: Compressed Query Strategy - matchDescendants method

4.7 Dictionary Thinning

Whichever partitioning method is chosen, there is potential for a large number of data dictionaries to be created by NSIndex. 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.

By the application of some additional processing, this redundancy in the data dictionaries can be removed. This 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 thin 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.

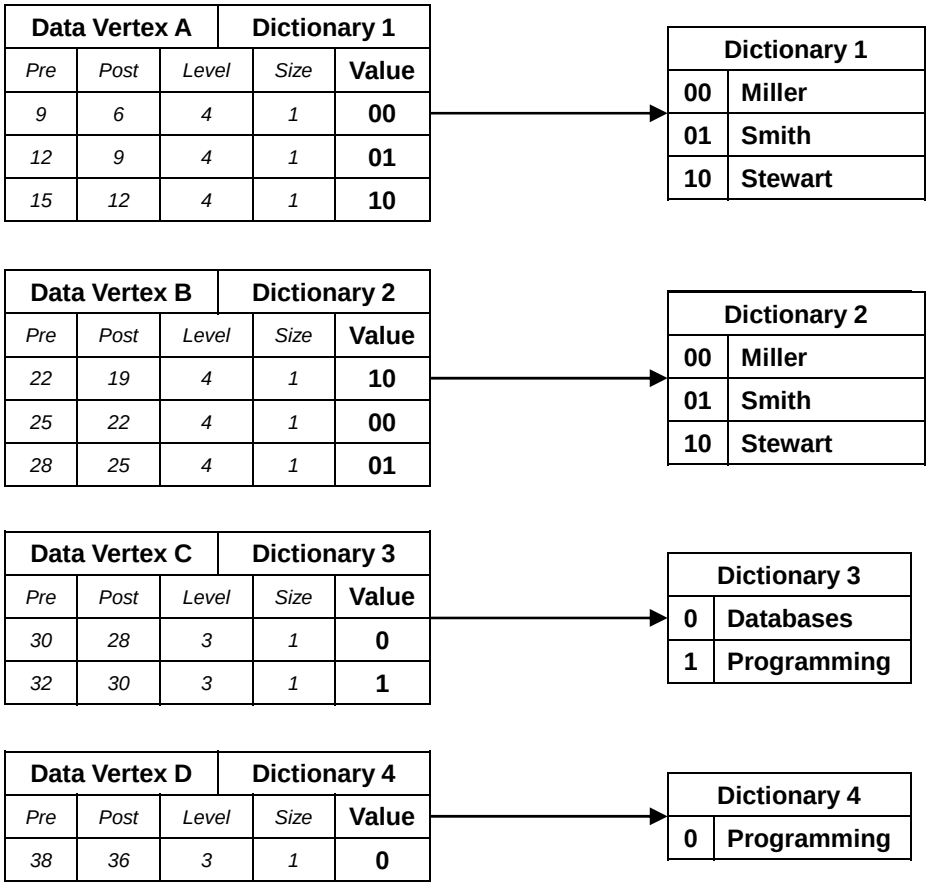

Fig. 4.2: Data Vertices and Dictionaries Before Thinning

An example of these processes is shown in Figures 4.2 and 4.3. The compressed NSIndex data vertices and dictionaries before the thinning process are illustrated in Figure 4.2 with each of the four example data vertices having an associated dictionary. Note that Dictionaries 1 and 2 are duplicates of each other and that the values stored in Dictionary 4 are a subset of those in Dictionary 3. As depicted in Figure 4.3, the thinning process disposes of Dictionary 2 and points Data Vertex B at Dictionary 1 (no change to the compressed data values is required as these

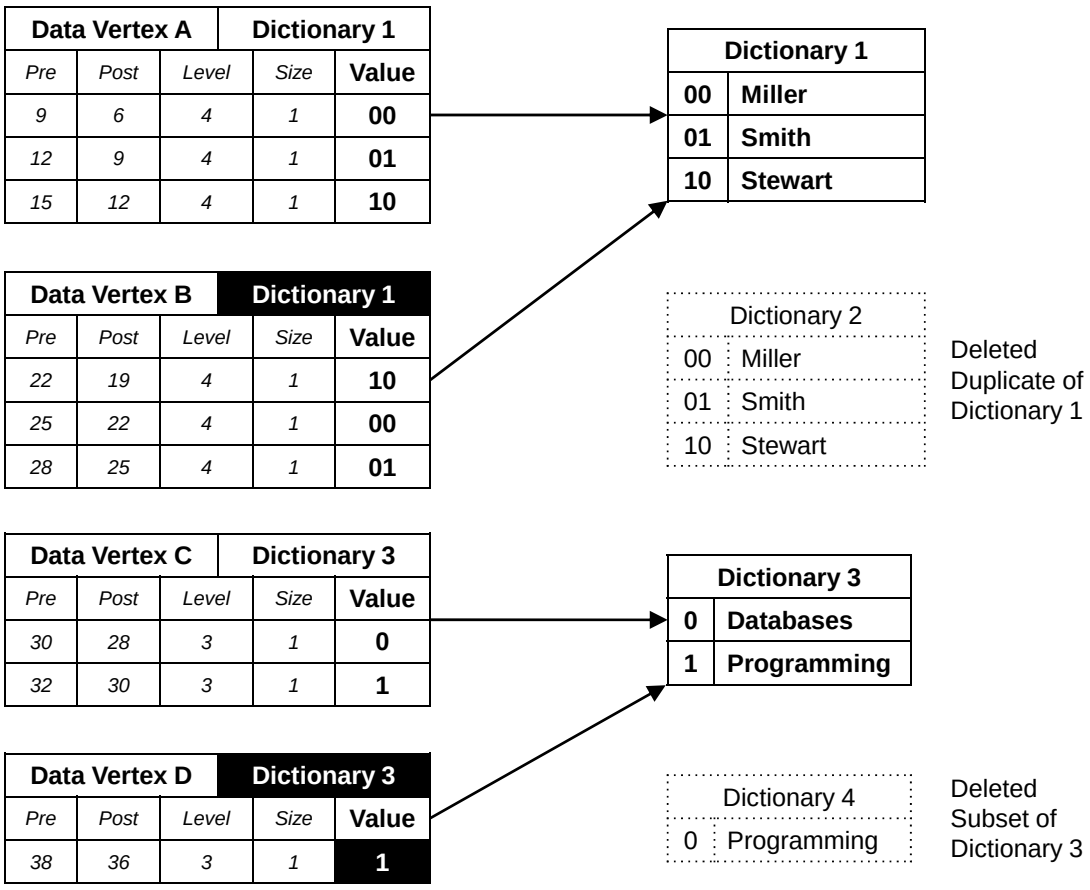

Fig. 4.3: Data Vertices and Dictionaries After Thinning (updated values shaded)

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 (equivalent to the mathematical "handshaking" problem[1] - *n(n-1)/2* comparisons) and for the existence of a subset (a further *n(n-1)* comparisons).

Using advance knowledge of dictionary file sizes captured as the dictionaries were produced, this can be reduced to the equivalent of the handshaking problem. Knowing if the file sizes are identical decides whether to test for 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.

This present work considers only subsets that appear between dictionaries of the same token size - this avoids affecting the size of the compressed data values where the token size would be increased. With respect to file comparisons this means that instead of comparing a single large set of dictionaries, a number of smaller sets of dictionaries are compared, reducing the overall number of comparisons. To this end, the token sizes are also recorded as the dictionaries are produced.

It should be noted that the above refers to the maximum number of comparisons for a set of dictionaries where no duplicates or subsets exist. Where a duplicate or subset is found, the redundant dictionary is removed and takes no part in any further comparison.

Listing 4.2 shows the information on token size and file size captured as the dictionaries shown in Figure 4.2 are produced. The list is sorted in descending order first by token size (to find the groups to be compared) and then by file size such that potential duplicate dictionaries will appear next to each other (having identical file sizes) and, importantly, possible subset dictionaries will

[1] Where n people are in a room, how many handshakes are required in total for each person to greet every other person?

always appear after their potential superset dictionary (the subset dictionary necessarily being smaller).

```
Dictionary.1.dic    2    22
Dictionary.2.dic    2    22
Dictionary.3.dic    1    22
Dictionary.4.dic    1    11
```

Listing 4.2: Dictionary List

Algorithm 4 shows the overall dictionary thinning process with Algorithm 5 giving a more detailed account of the test for subset dictionaries. The method of creating the translation tables required to update the CSV file when a subset dictionary is found is shown in Algorithm 6.

```
repeat
    Read group of dictionaries with same token size from list.
    for each dictionary (A) in list do
        for each dictionary (B) lower in the list do
            if file size A == file size B then
                if B is duplicate of A then
                    Note to change references from B to A.
                    Mark B for deletion.
                    Remove B from list (no further comparisons).
                end
            else
                if B is subset of A then
                    Note to change references from B to A.
                    Note translation table for tokens.
                    Mark B for deletion.
                    Remove B from list (no further comparisons).
                end
            end
        end
    end
until End of dictionary list.
```

Algorithm 4: Dictionary Thinning Algorithm

Once the list of dictionary reference changes is sorted into ascending order by old dictionary number, the changes can then be applied during a single pass through the NSIndex CSV file. Where the old dictionary was a subset of the

```
repeat
    Compare item b in dictionary B to item a in dictionary A.
    if a == b then
        Increment a and b.
    else if b > a then
        Increment a.
    else if b < a then
        a = terms in dictionary A (force loop exit).
    end
until a >= terms in dictionary A OR b >= terms in dictionary B
if b == terms in dictionary B then
    Return true (all values found in dictionary A).
else
    Return false.
end
```

Algorithm 5: Subset Test

```
Read first subset dictionary value.
while subset dictionary value is not null do
    repeat
        Read next superset dictionary value.
    until matching values are found
    Add relevant superset dictionary token to translation table.
    Read next subset dictionary value.
end
```

Algorithm 6: Building Translation Table

replacement one, the appropriate translation table can be applied to the tokens as part of this process.

This dictionary thinning process takes no account of the location of the data values within the structure. For example, if a dictionary containing place names happens to contain the same data values as one containing surnames, one of these dictionaries will be removed as a duplicate and both data vertices will use the same dictionary. There is no limit to the number of data vertices that may share a dictionary.

The thinning process was applied to NSIndex summarised versions of the data sets described in Section 4.2. Informed by the results of earlier work (Section 4.4), these summarisations were created using the full F&B-Index partitioning method. The effects on dictionary numbers and file sizes are shown in Section 5.4.

4.8 Summary

This chapter has set out the main phases of experimental work: the preliminary investigation into the viability of using minimal-token dictionaries, the examination of the effects of differing combinations of bisimilarity and the trial of methods to thin out the dictionaries. In addition, the experimental data sets have been described and categorised and the necessary additions to the NSIndex code have been explained. The next chapter gives the results of the experiments detailed above, with discussion of these following in Chapter 5.

5. RESULTS & DISCUSSION

This thesis has described experiments designed to test the potential for bisimilarity-based partitioning and dictionary compression methods to be combined in a queryable storage model for semi-structured data. These investigated the potential for dictionary-based compression, evaluated differing approaches to bisimilarity-based partitioning, integrated compression with structural querying and explored the potential for rationalising the number of dictionaries produced. This chapter now considers the results of these experiments and the implications that they have for the future sharing of independent segments of data.

The chapter begins by looking at the results of each experiment in turn. Section 5.1 presents the results of the comparison of dictionary methods against Huffman-type text compression, with the results of the evaluation of partitioning methods following in Section 5.2. The testing of the modified NSIndex query system is shown in Section 5.3, with dictionary thinning results given in Section 5.4.

The discussion then turns to what may be taken from the experiments as a whole, with these findings then linked back to the hypothesis and research questions laid out in Section 1.2. Finally, the limitations of this thesis are discussed along with avenues for future work that arise from it.

5.1 Preliminary Work

This first experiment [GTW07], as introduced in Section 4.3, sought to compare the effectiveness of dictionary-based compression, similar to that found in the HiBase relational system, against Huffman-style text compression as used by the XGrind XML compressor. These results were complemented by a set of Theoretical results, based on calculation of what the minimum storage requirements for

the dictionaries and compressed data values would be.

This section of work was carried out using three data series as per Section 4.3: Orders, Modified Orders and Legal. Figures 5.1, 5.2 and 5.3 illustrate the results for each of these series in turn.

For purposes of comparison, two additional lines are shown on the graph. The XML line shows the size of the uncompressed XML files, while the Raw Data line shows the size of the uncompressed data values with all structural formatting removed.

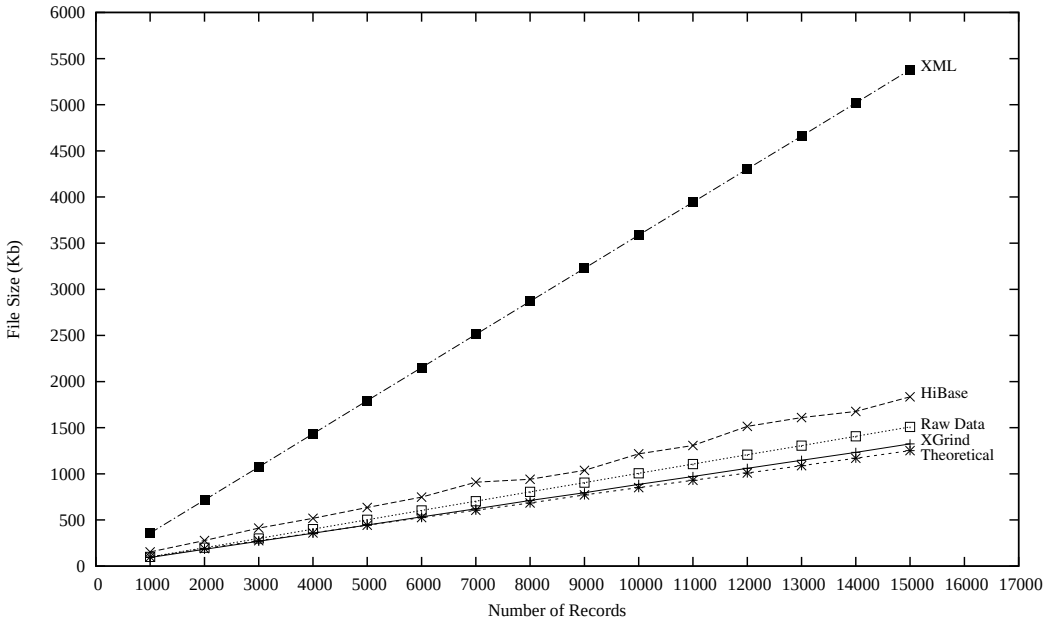

Fig. 5.1: Orders Data Set

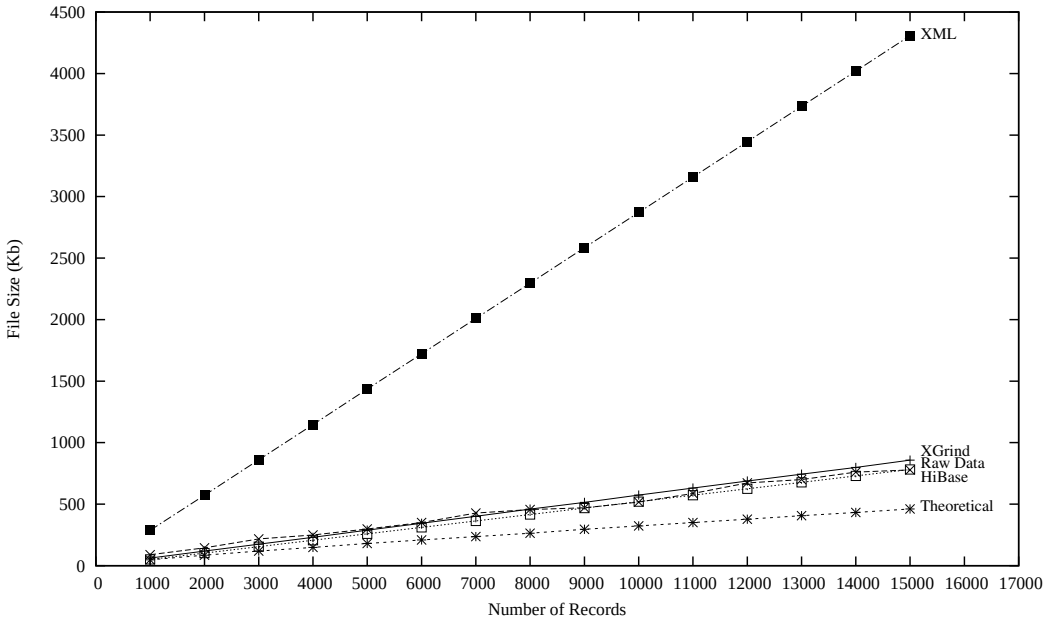

Fig. 5.2: Modified Orders Data Set

Leaving aside the Theoretical sizes for the moment, Figures 5.1, 5.2 and 5.3 show notable differences in the effectiveness of the HiBase and XGrind compression systems across the three data series. In Figure 5.1 it is shown that, while each of the compressed representations give a reduction in size over the original Orders XML files, HiBase produces the largest of these (slightly above the Raw Data line) with XGrind's compressed files being considerably smaller (and below the Raw Data line). By comparison, the gap between the two compression systems is much closer for the Modified Orders data series (Figure 5.2), where there are only marginal differences in the compressed sizes. File sizes for HiBase and XGrind are close to, but slightly above, the Raw Data line. XGrind file sizes are marginally smaller for files up to 8000 records, but from that point HiBase produces smaller files than XGrind.

As Modified Orders differs from Orders only in the removal of the large generated text element, it follows that this is a result of the abilities of the two systems to cope with such non-repetitive data values. HiBase requires repetition of data values to make space savings using its dictionary-based technique. The problem is compounded in the Orders data by the length of the large text data values -

each of which must be stored uncompressed in the dictionary for that element. XGrind's Huffman coding technique is not affected by the lack of repetition in the data values, it operates on a per-character basis, and does not store uncompressed versions of the large data values.

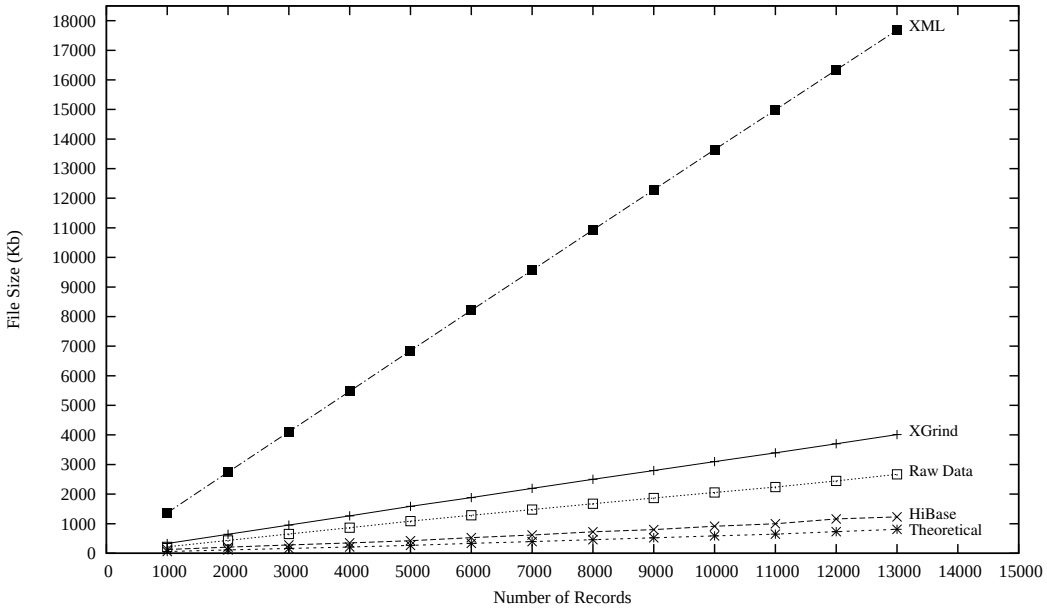

Fig. 5.3: Legal Data Set

The situation is reversed when looking at the Legal data series (Figure 5.3), where the graph lines are more widely spread. XGrind files sizes are clearly above the Raw Data line, with results for HiBase well below. It is shown that HiBase offers considerably greater compression than that of XGrind. The higher level of data value repetition within the Legal data is exploited by the dictionary-based compression of HiBase resulting in a smaller compressed representation. As XGrind's compression is blind to this repetition at the data value level, a gap forms between the two systems.

These preliminary results show good levels of compression for HiBase over the real world data (Legal) and results that are comparable with XGrind for benchmark data without large generated text elements (Modified Orders). Although HiBase does not perform so well with respect to such random elements, as shown

with the Orders data, there is still a considerable level of compression compared with the original XML file.

Returning to the Theoretical compressed data sizes, as was expected these are lower than the sizes produced by HiBase in all cases - due to the overheads present in the full HiBase system (as noted in Section 4.3).

Comparing the benchmark data series to the real world Legal series, it can be seen that where there is greater repetition of data values, the effect of the HiBase overhead is lessened. Fewer unique data values leading not only to smaller dictionaries, but to smaller access support structures as well.

For each of the data series, including the Orders series for which HiBase provided less effective compression than XGrind, the Theoretical compression consistently produced the smallest compressed size. This indicates that dictionary-based methods form a reasonable basis for the incorporation of data value compression into semi-structured data storage.

5.2 Evaluation of Partitioning Schemes

The second experiment [TFGW09], as described in Section 4.4, explored the effects of differing types of bisimilarity upon the test data sets. Changes in the number of data vertices produced by the partitioning methods and in the sizes of the compressed data and associated dictionaries were measured. The next two subsections discuss each of these in turn, with the following subsection bringing the two together.

5.2.1 Effect on Number of Data Vertices

Table 5.1 displays the results produced by the various bisimilarity strategies applied to the sample data sets. These results are summarised in Figure 5.4 where, as the data sets vary greatly in the number of data vertices produced, a logarithmic scale is employed - this causes the omission of the "No bisimilarity" and "Forwards bisimilarity only" bars from the graph.

	No Bisimilarity	Forwards Bisimilarity	Backwards Bisimilarity	Full Bisimilarity
XMark-10	1	1	405	46311
XMark-30	1	1	444	129186
Orders-15	1	1	10	10
ModifiedOrders-15	1	1	9	9
Legal-1	1	1	38	346
Legal-13	1	1	40	1533
Dream	1	1	15	100
Medline	1	1	76	31539
NASA	1	1	88	34884
Rat	1	1	77	161
Human	1	1	86	170

Tab. 5.1: Effects of Bisimilarity Options on Number of Data Vertices

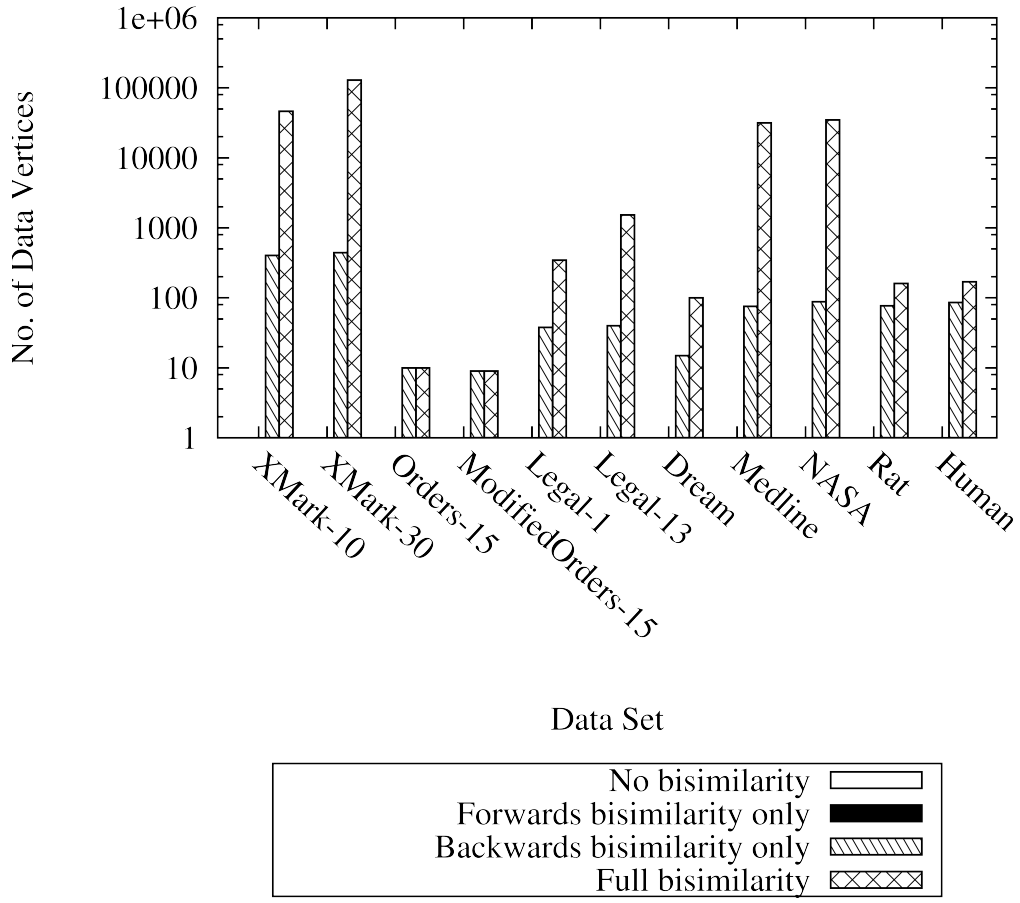

Fig. 5.4: Effect of Bisimilarity Options on Number of Data Vertices

It can be seen that for each data set processed using the no bisimilarity option, only one data vertex is produced. This is expected behaviour. As previously noted, the NSIndex program groups the datagraph nodes by their label and, just as all nodes of type `StaffMember` or `Student` would be grouped together, so too are the `CDATA` nodes (which contain all the data values) 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[1]. 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

[1] The number of structural vertices does increase, see Appendix B.

to examine.

Working in the opposite direction, the use of full backwards bisimilarity results in a notable 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.

Across the data sets there is great variety in the numbers of data vertices produced using backwards bisimilarity. This is a result of 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.

The full complexity of the XML structure is taken into account when the final NSIndex partitioning method is employed. 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.

For example, when using forwards bisimilarity an author with two (descendant) books is deemed different from an author with three books - however all the books would still be grouped together (each book having the same descendants). When both forwards and backwards bisimilarity are employed, an examination of the incoming paths of the book nodes would reveal these two different types of author node and the book nodes would be split accordingly (books by a two-book author and books by a three-book author).

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 (Table 5.1, Figure 5.4), where the irregular structure of the data set leads to a large number of data vertices when forwards and backwards bisimilarity is employed. This 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.

5.2.2 Effect on Compressed Data and Dictionary Sizes

The data sizes given in this section, and recorded in Table 5.2 and Figure 5.5, are each presented as a percentage compared to the total size of the uncompressed data values within that data set.[2] The compressed size is taken to be the size of the data dictionaries generated by NSIndex plus the size of the compressed data values (as tokenised using the dictionaries).

	No Bisimilarity	Forwards Bisimilarity	Backwards Bisimilarity	Full Bisimilarity
XMark-10	97%	97%	96%	99%
XMark-30	88%	88%	94%	99%
Orders-15	76%	76%	68%	68%
ModifiedOrders-15	54%	54%	43%	43%
Legal-1	29%	29%	20%	23%
Legal-13	35%	35%	20%	22%
Dream	99%	99%	99%	99%
Medline	73%	73%	67%	77%
NASA	64%	64%	64%	72%
Rat	57%	57%	56%	54%
Human	57%	57%	46%	46%

Tab. 5.2: Effects of Bisimilarity Options on Compressed Sizes

[2] The actual data sizes for this experiment are given in Appendix B.

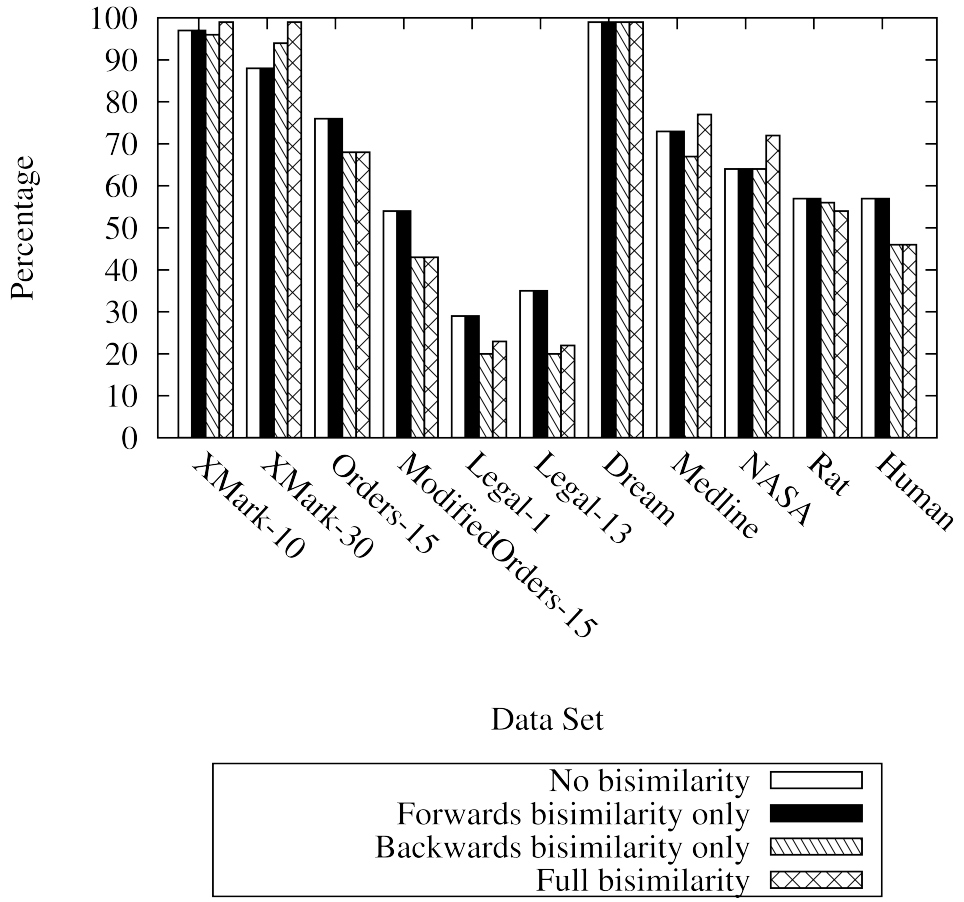

Fig. 5.5: Effects of Bisimilarity Options on Compressed Sizes

Table 5.2 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. With only one dictionary for each data set, corresponding to the single data vertex, the differences in compression levels are purely down to the number of unique data values found within each data set as a whole. For example, the Legal-1 data set with a high level of data value repetition is reduced to 29% of its uncompressed size, while the Dream data set is only lowered to 99% of its uncompressed size (there being little repetition among the lines and stage directions of the Shakespeare play).

When considering the effects of backwards bisimilarity on compressed data

and dictionary sizes, it is important to note that it is the distribution of the data values across the data vertices that makes a difference. As this changes, so too does the repetition of data values within any given data vertex and it is this repetition that the dictionary compression scheme relies upon to operate effectively.

This explains the differing levels of change in compressed size visible across the data sets. For most of the data sets used, the new distribution of data values caused by the move to backwards bisimilarity separates the data values out in such a way that it permits the dictionary-based compression scheme to use smaller tokens, i.e. where the new set of data vertices have a greater level of repetition of data values per vertex. The effect is only notable where this type of data value separation occurs - if the repartitioning does not separate the values in this way, the process can have negligible effect - for example, the Dream, NASA and Rat data sets where sizes were reduced by less than 2%.

It is noted that in the case of XMark-30 the introduction of backwards bisimilarity has had a negative effect and the compressed size has increased 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 - as such the dictionary-based scheme cannot operate as effectively and the compressed size rises.

As shown earlier, the combined use of full forwards and backwards bisimilarity adds an extra discriminator to the partitioning process and 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.

5.2.3 Selection of Partitioning Method

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 data vertices produced by the full forwards and backwards bisimilarity method makes it the most reasonable compromise. Accordingly the experiments described in Sections 4.6 and 4.7 and discussed in Sections 5.3 and 5.4 use this partitioning method to obtain their initial data.

5.3 Querying

The work on querying the compressed structure, set out in Section 4.6, 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.

Table 5.3 lists six queries performed over three of the test data sets - Legal-1, Orders-1 and XMark10. Each query is executed using both the unmodified NSIndex system (no data value compression, original query strategy) and the extended NSIndex system (using dictionary compression and a query strategy which takes advantage of the structure).

It is shown that for all six queries the same number of results are returned for each strategy and thus the ability to access the data values has not been affected by the introduction of data value compression. Therefore it is of interest how many vertices each strategy must utilise to achieve these results.

Data Set	Query	Strategy	Results	Structural Vertices	Data Vertices
Legal-1	/sis/pc_age=“21”	unmodified	33	20	346
		modified	33	20	10
Legal-1	/sis/[pc_judge=“Lady Cosgrove” & pc_category=“Homicide”]	unmodifed	3	30	346
		modifed	3	30	20
Orders-1	/T/O_CUSTKEY=“370”	unmodifed	4	2	10
		modifed	4	2	1
Orders-1	/T/[O_ORDERSTATUS=“O” & O_ORDER-PRIORITY=“1-URGENT”]	unmodifed	95	3	10
		modifed	95	3	2
XMark10	/regions/africa/item	unmodifed	55	2382	0
		modifed	55	51	0
XMark10	/regions/*/item	unmodifed	2175	76653	46311
		modifed	2175	2227	0

Tab. 5.3: Test Queries

For the first four test queries 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. However, the number of data vertices required differs. Due to the way in which the unmodified NSIndex query strategy processes the query, each data vertex in the structure is accessed to check for matching data values. By contrast, as the modified query strategy only accesses the data vertices which satisfy the structural part of the query, reducing the pool of potential matches at each stage, a much smaller number of data vertices are required. In the case of the first query 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 unmodified strategy must access.

Looking at a data set with a more complex structure, XMark10, the effects of the modified query strategy upon the structural part of the query may be observed. Query 5 seeks to find all the `item` elements listed under `africa`. In the case of the unmodified strategy this means accessing the vertices relating to `regions` and `africa` and all of the `item` vertices held across the entire structure. By comparison the modified strategy also accesses `regions` and `africa` but is able to confine its search to only those vertices which appear below `africa` when looking for `item`, answering the query while accessing considerably fewer vertices. As a purely structural query, neither query method accesses any data vertices.

The effect is even more pronounced in Query 6 which includes a wildcard character ("*"). While the structure-minded modified strategy is able to follow the structure and check only the vertices below `regions`, the unmodified strategy interprets the wildcard as being any vertex within the entire structure. This results in not only every structural vertex being accessed but also every data vertex as, despite the query being purely structural, the unmodified system does not fully differentiate between structural and data vertices when dealing with wildcards.

These results show that the addition of data value compression has not compromised access to the data values and that the use of a query strategy that takes account of the structure of the compressed data accesses less of the structure to

satisfy a query than the previous query strategy. This has the effect of fewer potential matches having to be evaluated at each stage but, more importantly in terms of sharing segments of data, also means that fewer segments of data must be transferred to permit query execution.

5.4 Dictionary Thinning

This final experiment [TFGW10], described in Section 4.7, aimed to reduce the redundancy inherent across the set of dictionaries produced for each data set by thinning out unnecessarily repeated dictionaries and leaving a smaller, useful set in place. The three measured effects of dictionary thinning are discussed in separate subsections: the effect on the number of dictionaries, the effect on dictionary size and the effect on dictionary size on disk. In all cases this thinning process was applied to an NSIndex version of the data set created using the full forwards and backwards bisimilarity method.

5.4.1 Effect on Number of Dictionaries

The effect of dictionary thinning on the number of dictionaries required is shown in Table 5.4. The results largely fall into three groups, which are associated with the data set categories defined in Section 4.2: the two regularly-structured benchmark data sets (Orders-15 and ModifiedOrders-15) are both unaffected by the dictionary thinning process, while the two irregular XMark benchmark data sets show a 32% (XMark-l0) and 39% (XMark-30) reduction in dictionary numbers. All but one of the real world, irregular data sets show between a 56% and 71% reduction (the exception is the Dream data set, discussed below).

	No. of dictionaries before thinning	No. of dictionaries after thinning	Reduction
XMark-10	46311	31368	32%
XMark-30	129186	78988	39%
Orders-15	10	10	0%
ModifiedOrders-15	9	9	0%
Legal-1	346	137	60%
Legal-13	1535	504	67%
Dream	100	79	21%
Medline	31539	9221	71%
NASA	34889	15445	56%
Rat	161	60	63%
Human	170	61	64%

Tab. 5.4: Consequences of Thinning on Number of Dictionaries

Thinning works by identifying duplicate and subset dictionaries without regard to the kind of data represented by those dictionaries. The variable effect of dictionary thinning across the data sets is again due to data value distribution. However, by contrast to data partitioning where it is repetition of values within individual data vertices that is important, for dictionary thinning it is the repetition of data values across the data dictionaries that determines how successful the process will be.

The difference in the effect of dictionary thinning between the real world and benchmark data sets is a result of this. As benchmark data sets have their data values generated, they are less likely to produce repeated data values, and consequently less likely to produce duplicate or subset dictionaries.

This also applies in part to the Dream data set. Although categorised as a real world data set, a large part of it acts more like benchmark data in that the data values provided by the spoken words and stage directions of "A Midsummer Night's Dream" have more in common with generated text than with data values from a real world database.

As the structure of the Orders-15 and ModifiedOrders-15 data sets did not

lend itself to partitioning, the resultant NSGraph representations consist of only 10 and 9 dictionaries respectively - one for each of the elements held in an order record. There are no duplicate or subset dictionaries for these data sets and dictionary thinning has no effect on the number of dictionaries or either of the file size measurements discussed in Sections 5.4.2 and 5.4.3.

5.4.2 Effect on Dictionary Size

Changes to the total dictionary file sizes for each data set are shown in Table 5.5. These are the logical file sizes and represent the space actually required by the NSIndex representation of the dictionaries, not the amount allocated by the file system. It can be seen that the reductions in file sizes caused by dictionary thinning do not correspond to the reductions in dictionary numbers. For example, the Rat data set has 63% of its dictionaries removed and reduces in size by 31%, but the Human data set dispenses with 64% of its dictionaries and only decreases its dictionary size by 11%.

	Size before thinning (bytes)	Size after thinning (bytes)	Reduction
XMark-10	8029213	7759082	3%
XMark-30	23719667	22500517	5%
Orders-15	1065251	1065251	0%
ModifiedOrders-15	312682	312682	0%
Legal-1	46908	43387	8%
Legal-13	574132	503894	12%
Dream	92414	92011	0%
Medline	5448004	5222699	4%
NASA	8999070	8371988	7%
Rat	1585693	1091384	31%
Human	1293906	1156494	11%

Tab. 5.5: Consequences of Thinning on Logical Dictionary Size

These differences are due to the variable size of the individual dictionaries

involved. Removing a few large dictionaries can have a greater effect on the overall dictionary size than removing many small dictionaries. In the case of the Rat data set, eight of the dictionaries removed were over 21Kb in size (including three at over 99Kb) whereas with the Human data set all dictionaries removed were under 17Kb, with only ten above 1Kb.

5.4.3 Effect on Dictionary Size on Disk

	Size on disk before thinning (bytes)	Size on disk after thinning (bytes)	Reduction
Xmark-10	190496788	129290260	32%
XMark-30	532369408	326762496	39%
Orders-15	1093632	1093632	0%
ModifiedOrders-15	339968	339968	0%
Legal-1	1429504	573440	60%
Legal-13	6684672	2420736	64%
Dream	471040	385024	18%
Medline	132198400	40787968	69%
NASA	146227200	66531328	55%
Rat	2170880	1294336	40%
Human	1929216	1368064	29%

Tab. 5.6: Consequences of Thinning on Dictionary "Size on Disk"

As the NTFS file system used on the test system allocates disk space in 4Kb units, the actual disk space required by the dictionary files is larger than the logical sizes discussed above. The effect dictionary thinning has on the "size on disk" for each data set is shown in Table 5.6, however the relationship between these sizes and the logical sizes is best shown in Figure 5.6.

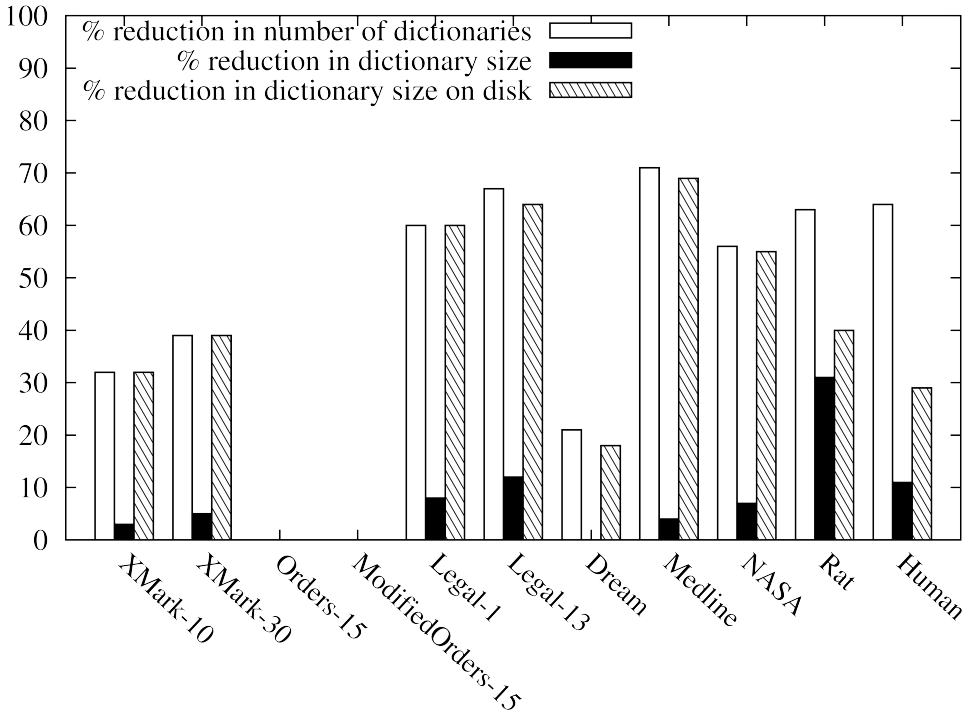

Fig. 5.6: Summary of Dictionary Thinning Effects

Looking at the Medline data set, 71% of its dictionaries have been removed by the thinning 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. 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.

The dictionary thinning process helps to eliminate some of the duplication of data within the NSIndex dictionaries. Although the effects on logical space occupied by the dictionaries appear modest, it is the "size on disk" that is more important in terms of occupying storage space and for this the results are greater. In any case, there is a potentially useful reduction in the number of dictionary files that require to be managed.

5.5 General Discussion

This work has sought to extend the NSIndex system to provide a rational method of storing data values as part of the structural summarisation process. While a level of compression was required to make efficient use of storage space, it was a key consideration that the data values must be accessible without the need to decompress the entire structure first.

Preliminary work showed the potential compression achievable with a dictionary-based minimal-size token scheme similar to that used in HiBase - especially over data with high repetition of data values. This, combined with the fact that decompression would only be performed upon the data values actually involved in a query, led to the selection of this as the basis of the NSIndex data value storage system.

An additional benefit of the dictionary-based scheme is that only those dictionaries actually involved with a particular query need to be accessed by the NSIndex system as demonstrated by Section 5.3. 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 the queryable non-homomorphic compressors ([ABC+04], [CN04], [NLWL06], [WLS07]) considered earlier which must be transferred as a complete single entity.

The consideration of differing types of bisimilarity in the partitioning process

also has a bearing upon this. This present work corroborates the findings of the earlier work [GTW07] with regard to the effects of varying bisimilarity on the number of vertices produced using a wider range of test data sets than before. This is extended through a comparison of the effects of differing bisimilarity methods on the size of the compressed data and dictionaries. Section 5.2.3 notes that the use of backwards bisimilarity is best for overall compressed size, but that full forwards and backwards bisimilarity may be a better compromise based on the number of dictionaries produced. With a view to sharing segments of data, the larger number of dictionaries can be considered preferable as it offers a finer level of granularity of data and therefore means less data extraneous to the user's query is transferred alongside the relevant data.

Likewise it is not simply the total size of the dictionaries that is at issue regarding dictionary thinning. 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 NSIndex structure. This increases the likelihood that additional user queries may be satisfied using dictionaries that have already been acquired, as discussed in Section 1.1. Where files are being requested over a data connection this has a double effect: no further bandwidth is used and no additional storage space is required.

5.5.1 Research Questions and Hypothesis

To return to the research questions set out in Section 1.2:

RQ1: **Are dictionary-based methods a reasonable choice for use in a compressed semi-structured data storage model?**

In order to maintain the ability to query the data structure, it was proposed that direct access to the individual compressed data values should be a priority. This meant a choice between compression at the level of the individual data values or at the level of individual characters of the data - each method facilitating data access without the requirement to first decompress an entire grouping of compressed values.

The first experiment set out in Section 4.3 was designed to evaluate these two compression types, i.e. to compare value-based dictionary compression against text-based compression. It was demonstrated that dictionary-based methods, as represented by HiBase, offered levels of compression in excess of those attained by the Huffman-type compression used by XGrind for the real world data sets. Although in the worst case the reverse was true for the benchmark data set incorporating large text elements, dictionary-based compression still shows a considerable reduction in size compared to the original XML file.

Given these results for the fully working dictionary-based system and that the theoretical dictionary compression sizes were the smallest across all data sets, it is clear that dictionary-based methods are indeed a reasonable choice for use in a compressed semi-structured data storage model.

RQ2: **What are the effects of varying the partitioning method on the storage of data values?**

It was noted that the number and sizes of the compressed segments produced by the partitioning would be of interest with a view to potentially sharing the data held within the compressed model. Section 4.4 described an experiment to evaluate the effects of different types of bisimilarity-based partitioning.

The experimental results showed that overall, the use of full backwards bisimilarity produced the smallest total compressed data and dictionary sizes. However it is argued that the full forwards and backwards bisimiliarity method of compression produces a set of compressed data vertices and dictionaries that is more beneficial for the sharing of data - the larger set of vertices meaning that each individual item is smaller and more specialised, leading to less wasted transfer bandwidth.

RQ3: **Can a querying strategy designed around the compressed structure allow queries to be answered with reduced access to the data structure?**

It was demonstrated that use of the query strategy that restricted its use of data vertices based on the structural part of the query resulted in a considerable reduction in the number of data vertices accessed, i.e. only the parts of the structure that might potentially hold a result are examined. In terms of sharing independent segments of data, this would equate to a substantial reduction in the volume of data (and dictionaries) that must be transferred to satisfy a query - the transfer of those parts not considered by the query strategy need never be requested.

RQ4: **With the data split into segments, how might the volume of dictionaries be managed?**

The experimental work described in Section 4.4 showed that partitioning using bisimilarity did indeed produce a large number of data vertices and consequently a high number of associated dictionaries. The work described in Section 4.7 considered the reduction of redundancy between these dictionaries by eliminating duplicate dictionaries and those whose values were a subset of another dictionary.

It was found that this thinning process could significantly reduce the number of dictionary files involved, especially for the majority of the real world data sets, potentially reducing the number of dictionary file transfers that may be required in a sharing environment. There is also a benefit in terms of storage requirements. Although the reductions in logical space requirements are quite modest for most data sets (due to the small sizes of the dictionaries removed), there are noticable reductions in the total disk space used once file system allocation methods are taken into consideration. Both the reduction in dictionary numbers and the savings in storage requirements are beneficial to the overall management of the dictionaries.

The original hypothesis presented in Chapter 1 claims that:

Independent sharing of data segments while maintaining direct query access is effectively facilitated by the combination of bisimilarity-based partitioning and dictionary compression methods.

The research questions explored in this thesis sought to examine the technical matters that derived from the hypothesis proposed, namely the validity and usefulness of dictionary-based compression and bisimilarity-based partitioning methods and whether a system based on these could be queried while accessing only a portion of the data structure. In light of the answers to these research questions, as informed by the results of the experiments described herein, it is reasonable to conclude that the hypothesis itself has been suitably tested and stands as a valid statement of the potential for compression in data sharing.

5.6 Limitations and Future Work

The partitioning methods used by the NSIndex system produce a large number of dictionary files for each data set. The dictionary compression added to the system in Section 4.5 removes the duplication of data values within data vertices and the dictionary thinning process described in Section 4.7 is able to remove some of the duplication of data values that occurs across the set of dictionaries for each data set. However there are still issues to be considered with respect to the storage of dictionaries, especially the smaller ones, as highlighted by the gap between logical file size and the allocated disk space noted in the work on dictionary thinning (Section 5.4).

The potential effects of combining small dictionaries needs to be examined. One possible method is to combine 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. Some balance point must be found between dictionary numbers and compressed data sizes that allows for less wastage of the allocated

disk space.

On a related note it would be worth considering whether there is a size of dictionary beyond which it makes sense to physically split the dictionary into parts for the purposes of sharing between users. These parts could potentially be reassembled as a single dictionary as they are needed or treated as number of smaller dictionaries with appropriate changes to the tokens used in the compressed data.

It would also be worth investigating whether a second level of compression may usefully be applied to the dictionaries (the first level of compression being the tokenisation within the data structure). This could be particularly useful 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.

In the current implementation of the NSIndex system, point queries with branching paths can be successfully processed. Since the addition of the code to deal with compressed data values and the associated dictionaries, it is only possible to perform relatively simple queries upon the data it holds. Algorithms to perform more complex queries or those involving value wildcards are present within the codebase but, as noted in Section 4.6, the pre-existing query strategies require adaptation to cope with compressed data values. The order-preserving nature of the data tokens (i.e. if value-A $<$ value-B then token-A $<$ token-B) permits the development of querying approaches that would support range queries directly over the compressed data tokens. Once querying is fully featured and the selective loading of dictionaries (see below) is implemented as part of this process, it would then be appropriate to consider query performance in greater detail. The current working query system serves to demonstrate that it remains possible to access, and properly decompress, the stored data.

One aspect of the NSIndex system touched upon above, is that the system only requires access to the dictionaries directly related to any user query it is given. At present, this on-demand system of loading dictionaries has not been implemented as it was not required for this set of experiments. It is conceivable

that, given the tree-like structure of NSIndex, this selective loading could be extended to the structural element of the NSIndex data representation. The structural summarisation could be split into sub-trees with these also loaded on-demand.

Finally, the current method of writing the structural component of NSIndex to file, as shown in Listing 3.1, is highly verbose. For this present work (which has been concerned with the sizes of data values and dictionaries only) this has been useful for debugging purposes, but the structural part of NSIndex could be stored in a more compact manner by switching to a less humanly-readable format.

5.7 Summary

This chapter has discussed the results of the experimental work as detailed earlier in this thesis. Having established the validity of dictionary-based compression in the context of a segmented semi-structured storage model and evaluated the effects of differing types of bisimilarity in the partitioning process, the reduced data access required by the structure-aware query strategy was noted. The potential benefits of thinning the dictionaries were then discussed before bringing the focus back to the research questions and hypothesis. A consideration of the limitations of the work and potential areas for futher consideration rounded off the chapter. The final chapter looks back and summarises the thesis as a whole.

6. CONCLUSION

The aim of this thesis 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.

The context was set by Chapter 1, which also outlined the hypothesis and research questions to be explored. The main hypothesis was that "Independent sharing of data segments while maintaining direct query access is effectively facilitated by the combination of bisimilarity-based partitioning and dictionary compression methods".

Chapter 2 then provided the background to this work giving some illustrative examples of the kinds of technologies that have previously been employed to process XML. Initially, indexing ([GW97]) and structural summarisation ([KSBG02], [BGK03], [KBNK02]) methods that sought to speed up querying were reviewed, along with non-queryable compression systems ([LS00], [Che01], [LW02], [SGS07]) suitable for archival purposes. Of the queryable compression methods it was noted that homomorphic compression ([TH02], [MPC03], [SS07]) which kept data and structure together allowed for traditional XML parsing techniques to be used. There followed a review of non-homomorphic methods ([ABC$^+$04], [CN04], [NLWL06], [WLS07]), which offered improved compression and querying by splitting data and structure - drawing upon ideas from structural summarisation techniques. However it was noted that each of these methods were designed only to compress data and must be transferred as a single unit, rendering them unsuitable for the sharing of segments of data.

Chapter 3 provided further details of the NSIndex, HiBase and XGrind systems used and extended during the experimental work of the thesis set out in Chapter 4.

The Preliminary Work sought to compare dictionary-based compression (as represented by HiBase) against text-based compression (as represented by XGrind) to find an appropriate method for the storage of data values, such that the values could be accessed individually. The results showed that the real world data sets were most compressed by HiBase and the benchmark data sets were most compressed by XGrind. However, over all the data sets the calculated theoretical dictionary compression consistently offered the greatest compression, thus validating the dictionary-based method as a reasonable choice for use.

Having identified bisimilarity as an appropriate method of segmenting the data, the second experiment, on the Evaluation of Partitioning Methods offered by NSIndex, considered varying combinations of forwards and backwards bisimilarity. Supporting and extending knowledge from the earlier work ([GTW07]), it was shown that backwards bisimilarity produced the smallest overall compressed sizes, although the finer granularity offered by the combination of forwards and backwards bisimilarity could provide benefits with respect to data transfer bandwidth.

The work on the Integration of Data Value Compression described the steps necessary to build dictionary-based compression into the NSIndex system. In addition to the construction of dictionaries and the subsequent encoding of data values, a method of loading NSIndex structures from file was implemented. This was followed by the description of a query strategy devised to 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 final experiment looked at Dictionary Thinning to reduce redundancy across the set of dictionaries associated with a compressed NSIndex structure. It was shown that by removing duplicate dictionaries, and by eliminating those that were a subset of another, the number of dictionaries could be reduced to a more manageable level with associated benefits in disk space requirements.

The experimental results were presented and discussed in Chapter 5 which then related these experimental findings back to the research questions and the

hypothesis quoted at the beginning of this chapter. The experimental data supports the validity of this hypothesis. This was followed by a consideration of the limitations of the thesis and the suggestion of future avenues of research.

The contribution of this thesis is the proposal 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.

BIBLIOGRAPHY

[ABC+04] Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, and Andrea Pugliese. Efficient Query Evaluation over Compressed XML Data. In *EDBT 2004* [BCP+04], pages 200–218.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[Ant97] Gennady Antoshenkov. Dictionary-Based Order-Preserving String Compression. In *VLDB '97* [JCD+97], pages 26–39.

[BCP+04] Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors. *Proceedings of 9th International Conference on Extending Database Technology: Advances in Database Technology (EDBT 2004), Heraklion, Crete, Greece, March 14-18, 2004*, volume 2992 of *Lecture Notes in Computer Science*. Springer, 2004.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path Queries on Compressed XML. In Johann Christoph Freytag, Peter C. Lock-

emann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003), September 9-12, 2003, Berlin, Germany*, pages 141–152. Morgan Kaufmann, 2003.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query Optimization In Compressed Database Systems. In *SIGMOD 2001*, pages 271–282, 2001.

[Che01] James Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *Data Compression Conference (DCC 2001), 27-29 March 2001, Snowbird, Utah*, pages 163–. IEEE Computer Society, 2001.

[CMW98] W. Paul Cockshott, Douglas R. McGregor, and John Wilson. High-Performance Operations Using a Compressed Database Architecture. *The Computer Journal*, 41(5):283–296, 1998.

[CN04] James Cheng and Wilfred Ng. XQzip: Querying Compressed XML Using Structural Indexing. In *EDBT 2004* [BCP+04], pages 219–236.

[Die82] Paul F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC 14), 5-7 May 1982, San Francisco, California, USA*, pages 122–127. ACM, 1982.

[Ens] Ensembl. *Genome Annotation Project*. `http://www.ensembl.org`.

[GTW07] Richard Gourlay, Brian Tripney, and John N. Wilson. Compressed Materialised Views of Semi-Structured Data. In *Workshops of the Twenty Fourth British National Conference on Databases, Glasgow, Scotland, 2nd-3rd July 2007*, pages 75–82. IEEE Computer Society, 2007.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB '97* [JCD+97], pages 436–445.

[Huf52] David Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[IEE02] IEEE Computer Society Press. *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[JCD+97] Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors. *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB '97), August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering Indexes for Branching Path Queries. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 133–144. ACM, 2002.

[KSBG02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *ICDE 2002* [IEE02], pages 129–140.

[LS00] Hartmut Liefke and Dan Suciu. XMILL: An Efficient Compressor for XML Data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 153–164. ACM, 2000.

[LW02] Mark Levene and Peter Wood. XML Structure Compression. In *International Workshop on Web Dynamics*, 2002.

[MPC03] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: A Queriable Compression for XML Data. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 122–133. ACM, 2003.

[NLC06] Wilfred Ng, Wai Yeung Lam, and James Cheng. Comparative Analysis of XML Compression Technologies. *World Wide Web*, 9(1):5–33, 2006.

[NLWL06] Wilfred Ng, Wai Yeung Lam, Peter T. Wood, and Mark Levene. XCQ: A queriable XML compression system. *Knowledge and Information Systems*, 10(4):421–452, 2006.

[Sak09] Sherif Sakr. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009.

[SGS07] P. Skibinski, S. Grabowski, and J. Swacha. Fast Transform for Effective XML Compression. In *CAD Systems in Microelectronics, 2007, 9th International Conference - The Experience of Designing and Applications of (CADSM '07)*, pages 323 –326, Feb. 2007.

[Sha] Jon Bosak. *Shakespeare In XML*. `http://www.cafeconleche.org/examples/shakespeare`.

[SS07] Przemyslaw Skibinski and Jakub Swacha. Combining Efficient XML Compression with Query Processing. In Yannis E. Ioannidis, Boris Novikov, and Boris Rachev, editors, *Advances in Databases and Information Systems, 11th East European Conference (ADBIS 2007), Varna, Bulgaria, September 29-October 3, 2007, Proceedings*, volume 4690 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2007.

[Sta] *StatCounter Global Statistics, Mobile v Desktop Usage, June 2010 to June 2012.* http://gs.statcounter.com/#mobile_vs_desktop-ww-monthly-201006-201206.

[TFGW09] Brian Tripney, Christopher Foley, Richard Gourlay, and John N. Wilson. Sharing large data collections between mobile peers. In Gabriele Kotsis, David Taniar, and Eric Pardede, editors, *The 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2009), 14-16 December 2009, Kuala Lumpur, Malaysia*, pages 321–325. ACM, 2009.

[TFGW10] Brian Tripney, Christopher Foley, Richard Gourlay, and John N. Wilson. Efficient data representation for XML in peer-based systems. *International Journal of Web Information Systems (IJWIS)*, 6(2):132–148, 2010.

[TH02] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *ICDE 2002* [IEE02], pages 225–234.

[TWH96] Cyrus Tata, John N. Wilson, and Neil Hutton. Representations of Knowledge and Discretionary Decision-Making by Decision-Support Systems: the Case of Judicial Sentencing. *Journal of Information, Law and Technology*, 1996(2), 1996.

[Uni] University of Washington. *XML Repository.* http://www.cs.washington.edu/research/xmldatasets.

[VVI04] T. Pliakas V. Vlahakis, A. Demiris and N. Ioannidis. Experiences in applying augmented reality techniques to adaptive, continuous guided tours. In *IFITT ENTER*, pages 26–28, January 2004.

[WGJN06a] J. N. Wilson, R. Gourlay, R. Japp, and M. Neumueller. Extracting Partition Statistics from Semistructured Data. In *17th International Workshop on Database and Expert Systems Applications (DEXA 2006)*, pages 497–506, Piscataway NJ, September 2006. IEEE.

[WGJN06b] John N. Wilson, Richard Gourlay, Robert Japp, and Mathias Neumüller. A Resource Efficient Hybrid Data Structure for Twig Queries. In Sihem Amer-Yahia, Zohra Bellahsene, Ela Hunt, Rainer Unland, and Jeffrey Xu Yu, editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2006.

[WLS07] Raymond K. Wong, Franky Lam, and William M. Shui. Querying and maintaining a compact XML storage. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 1073–1082. ACM, 2007.

[XMa] *XMark XML Benchmark Project.* `http://www.xml-benchmark.org`.

[XML] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation 10 February 1998, `http://www.w3.org/TR/1998/REC-xml-19980210`.

APPENDIX

A. DESCRIPTION OF DATA SETS

The subsections below provide short descriptions and examples for the data sets described in Chapter 4.

A.1 XMark Benchmark

The XMark XML Benchmark Project [XMa] provides a generator to create test data files based on the structure of an online auction website. It is designed to mimic the important features of XML documents. With no fixed pattern to its structure and text formed from words randomly selected from Shakespeare, the XMark data is categorised as benchmark in origin and irregular in structure. XMark data was generated in file sizes of 10Mb and 30Mb. An example is given in Listing A.1.

A.2 Orders

A subset of the TPC-H relational benchmark converted into XML, Orders is categorised as benchmark and regular. As shown in Listing A.2 it is modelled on order summary data and contains randomly generated values. A series of test XML files were created ranging from 1000-15000 orders (349Kb - 5.1Mb).

A.3 Modified Orders

To see the effect of the large text element <O_COMMENT> in the Orders data set, the Modified Orders set was created with this generated text element removed. Modified Orders is otherwise the same as Orders so remains benchmark-regular and is used in a series of file sizes. These range from 279Kb for 1000 orders to 4.1Mb for 15000 orders.

```
<item id=``item3''>
<location>United States</location>
<quantity>1</quantity>
<name>poisons </name>
<payment>Money order, Creditcard, Personal Check, Cash</payment>
<description>
<text>
jack
</text>
</description>
<shipping>Will ship internationally, Buyer pays fixed shipping charges, See
    description for charges</shipping>
<incategory category=``category5''/>
<mailbox>
<mail>
<from>Mitsuyuki Toussaint mailto:Toussaint@uiuc.edu</from>
<to>Cort Penn mailto:Penn@uic.edu</to>
<date>07/17/2000</date>
<text>
gentleman observe silver eagle battles bastardy shames brook mounted officers
    dean shrunk lowness dew sandy prologue armies suspicion eighty advance
    thankfulness albany ended experience halt doubted wert kingdom fiend
    directed pair perhaps <emph> happy lucky odds rend condemn </emph> cannot
    dispos perfect silence
</text>
</mail>
</mailbox>
</item>
```

Listing A.1: Example of XMark Data

```
<T>
  <O_ORDERKEY>35</O_ORDERKEY>
  <O_CUSTKEY>1276</O_CUSTKEY>
  <O_ORDERSTATUS>O</O_ORDERSTATUS>
  <O_TOTALPRICE>194641.93</O_TOTALPRICE>
  <O_ORDERDATE>1995-10-23</O_ORDERDATE>
  <O_ORDER-PRIORITY>4-NOT SPECIFIED</O_ORDER-PRIORITY>
  <O_CLERK>Clerk#000000259</O_CLERK>
  <O_SHIP-PRIORITY>0</O_SHIP-PRIORITY>
  <O_COMMENT>fluffily regular pinto beans </O_COMMENT>
</T>
```

Listing A.2: Example of Orders Data

```
<sis>
  <in_id>2531</in_id>
  <in_year>2001</in_year>

  <pc_age>25</pc_age>
  <pc_sex>M</pc_sex>
  <pc_plea>Guilty</pc_plea>
  <pc_category>Robbery</pc_category>
  <pc_classm>Aggravated Robbery</pc_classm>
  <pc_classp>Complete Robbery</pc_classp>
  <pc_senttype>Imprisonment</pc_senttype>
  <pc_sentmths>54</pc_sentmths>

  <pc_forenames>Joseph</pc_forenames>
  <pc_surname>Bloggs</pc_surname>
  <pc_dob>7/12/1976 12:00:00</pc_dob>
  <pc_collector>Eric</pc_collector>
  <pc_sentcat>103</pc_sentcat>
  <pc_sentname>4 yr 6 m</pc_sentname>
  <pc_judge>Lord Cullen</pc_judge>
  <pc_location>Glasgow</pc_location>
</sis>
```

Listing A.3: Example of Legal Data

A.4 Legal

The first of the real world data sets, Legal consists of records from a court sentencing system developed within the department [TWH96]. Having been taken from a relational system, the XML is regular in structure. A series of files containing between 1000 and 13000 convictions were created with file sizes between 1.3Mb and 16.8Mb. An example is given in Listing A.3.

A.5 Dream

Real world data with irregular structure, Dream is the text of Shakespeare's "A Midsummer Night's Dream" as encoded into XML by Jon Bosak [Sha] and has a file size of 146Kb. Listing A.4 shows how both spoken word and stage directions are annotated.

A.6 Medline

A 20Mb section of the US National Library of Medicine bibliographic database in XML format. Listing A.5 gives an example of the data contained in Medline. This data set is real world and irregular.

```
<SPEECH>
<SPEAKER>TITANIA</SPEAKER>
<LINE>First, rehearse your song by rote</LINE>
<LINE>To each word a warbling note:</LINE>
<LINE>Hand in hand, with fairy grace,</LINE>
<LINE>Will we sing, and bless this place.</LINE>
</SPEECH>

<STAGEDIR>Song and dance</STAGEDIR>
```

Listing A.4: Example of Dream Data

```
<MedlineCitation Owner=''NLM'' Status=''MEDLINE''>
  <PMID>1365841</PMID>
  <DateCreated>
    <Year>1995</Year>
    <Month>03</Month>
    <Day>24</Day>
  </DateCreated>
  .
  .
  .
  <Journal>
    <ISSN>0047-6374</ISSN>
    <JournalIssue>
      <Volume>66</Volume>
      <Issue>2</Issue>
      <PubDate>
        <Year>1992</Year>
        <Month>Nov</Month>
      </PubDate>
    </JournalIssue>
  </Journal>
  <ArticleTitle>Effect of aging on macrophage adherence to extracellular matrix
      proteins.</ArticleTitle>
  <Pagination>
    <MedlinePgn>149-58</MedlinePgn>
  </Pagination>
  <Abstract>
    <AbstractText>Fibronectin, ...</AbstractText>
  </Abstract>
  .
  .
  .
</MedlineCitation>
```

Listing A.5: Example of Medline Data

```
<dataset subject=``astronomy'' xmlns:xlink=``http://www.w3.org/XML/XLink/0.9''>
  <title>Redshift distribution of galaxies in the southern Milky way region 210{
      deg}&lt;l&lt;360{deg} and |b|&lt;15{deg}.</title>
  <altname type=``ADC''>J/ApJS/107/521</altname>
  <altname type=``CDS''>J/ApJS/107/521</altname>
  <altname type=``brief''>Galaxies redshifts, 210&lt;l&lt;360, |b|&lt;15</
      altname>
  <reference>
   <source>
    <journal>
     <title>Redshift distribution of galaxies in the southern Milky way region
         210{deg}&lt;l&lt;360{deg} and |b|&lt;15{deg}.</title>
     <author>
      <initial>N</initial>
      <lastName>Visvanathan</lastName>
     </author>
     <author>
      <initial>T</initial>
      <lastName>Yamada</lastName>
     </author>
     <name>Astrophys. J. Suppl. Ser.</name>
     <volume>107</volume>
     <pageno>521</pageno>
     <date>
      <year>1996</year>
     </date>
     <bibcode>1996ApJS..107..521V</bibcode>
    </journal>
   </source>
   .
   .
   .
  </reference>
  .
  .
  .
 </dataset>
```

Listing A.6: Example of NASA Data

A.7 NASA

A section of metadata from an XML bibliographic project undertaken by NASA's Astronomical Data Centre. The centre closed in 2002 but the XML file is still available as part of the University of Washington's XML Repository [Uni]. The data is both real world and irregular in structure and has a file size of 23.6Mb. An example is given in Listing A.6.

A.8 Rat

A 25Mb section of rat genome data from the Ensembl genome annotation project [Ens]. This data is categorised as real world and regular. An example is given in

```
<ensembl-entry>
  .
  .
  .
  <external_db_refs>
    <external_db_ref>
      <status>KNOWN</status>
      <external_db_id>2200</external_db_id>
      <release>1</release>
      <object_xref_id>4052</object_xref_id>
      <db_name>SWISSPROT</db_name>
      <display_label>FGFA_RAT</display_label>
      <ensembl_id>123108</ensembl_id>
    </external_db_ref>
    .
    .
    .
  </external_db_refs>
  <gene>
    <gene_analysis_id>26</gene_analysis_id>
    <gene_display_xref_id>10775</gene_display_xref_id>
    <gene_description>
      <gene_description_description>FIBROBLAST GROWTH FACTOR-10 PRECURSOR (FGF
          -10). [Source:SWISSPROT;Acc:P70492]</gene_description_description>
      <gene_description_gene_id>96737</gene_description_gene_id>
    </gene_description>
    <gene_gene_id>96737</gene_gene_id>
    <gene_seq_region_id>137912</gene_seq_region_id>
    <gene_seq_region_start>50795032</gene_seq_region_start>
    <gene_seq_region_strand>1</gene_seq_region_strand>
    <gene_seq_region_end>50868552</gene_seq_region_end>
    <gene_type>ensembl</gene_type>
  </gene>
</ensembl-entry>
```

Listing A.7: Example of Rat Data

Listing A.7.

A.9 Human

A 25Mb section of human genome data from the Ensembl genome annotation project, categorised as real world and regular.

B. ADDITIONAL DATA

The following tables provide additional results which may be of interest along with those presented in Chapter 5.

B.1 Effects of Bisimilarity on Number of Vertices

Table B.1 shows the effects of using different combinations of bisimilarity upon the total number of vertices in the NSIndex structural summarisation. This includes both structural and data vertices and shows that, although forwards bisimilarity has no effect upon the number of data vertices (as shown in Table 5.1), it does have an effect higher up the NSIndex structure (upon the purely structural vertices).

	DataGraph	No Bisimilarity	Forwards Bisimilarity	Backwards Bisimilarity	Full Bisimilarity
XMark-10	319741	78	7091	933	122964
XMark-30	1010413	79	15258	993	318184
Orders-15	285004	14	14	23	23
ModifiedOrders-15	255004	13	13	21	21
Legal-1	70921	43	54	80	707
Legal-13	920678	44	101	83	3151
Dream	6203	18	33	38	244
Medline	997830	86	1112	188	77255
NASA	951681	76	986	217	90403
Rat	1114008	104	113	180	366
Human	1148835	115	124	200	386

Tab. B.1: Effects of Bisimilarity Options on Total Number of Vertices

B.2 Effects of Bisimilarity on Compressed File Sizes

Table B.2 shows the actual file sizes (in bytes) used to calculate the percentages shown in Table 5.2. As with Table 5.2 presented in the main text, each compressed size consists of both the compressed data values and the associated dictionaries.

	Uncompressed Size	No Bisimilarity	Forwards Bisimilarity	Backwards Bisimilarity	Full Bisimilarity
XMark-10	8113724	7852555	7852555	7822838	8071142
XMark-30	24103388	21128660	21128660	22590010	23923491
Orders-15	1776859	1333946	1333946	1215252	1215252
ModifiedOrders-15	1021781	551337	551337	436433	436433
Legal-1	286575	81812	81812	58278	65036
Legal-13	3574826	1244968	1244968	730736	796198
Dream	96235	95588	95588	95133	94846
Medline	7330615	5346378	5346378	4897319	5629049
NASA	12786475	8193973	8193973	8169505	9236373
Rat	3763682	2147705	2147705	2095163	2029221
Human	3949624	2249865	2249865	1820621	1801408

Tab. B.2: Effects of Bisimilarity Options on Dictionary Sizes