A Fast and Versatile Path Index for Querying Semi-Structured Data

Michael Barg Raymond K. Wong

School of Computer Science and Engineering

University of New South Wales

Sydney, NSW 2052, Australia

{mbarg,wong}@cse.unsw.edu.au

Abstract

The richness of semi-structured data allows data of varied and inconsistent structures to be stored in a single database. Such data can be represented as a graph, and queries can be constructed using *path expressions*, which describe traversals through the graph.

Instead of providing optimal performance for a limited range of path expressions, we propose a mechanism which is shown to have consistent and high performance for path expressions of any complexity, including those with descendant operators (path wildcards). We further detail mechanisms which employ our index to perform more complex processing, such as evaluating both path expressions containing links and entire (sub) queries containing path based predicates. Performance is shown to be independent of the number of terms in the path expression, even where these contain wildcards. Experiments show that our index is faster than conventional methods by up to two orders of magnitude for certain query types, is small, and scales well.

1 Introduction

The richness of semi-structured data allows data of varied and inconsistent structures to be stored in a single database. For example, an online catalog may integrate heterogeneous data adhering to multiple schema from multiple sites.

Semi-structured data can be represented as a graph, and queries can be constructed using *path expressions* [1], which describe traversals through the graph. As the graph may be constructed from widely varying heterogeneous data, it is not always possible to precisely specify the desired paths. For this reason, path expression may contain constructs such as *descendant operators* (path wildcards) which enable queries to be posed over multiple or imperfectly understood data sources.

A large number of path indexes have been proposed to evaluate path expressions. Such indexes, however, typically trade off performance with the expressiveness of the path expressions they can process [14, 13, 8].

We propose a path indexing mechanism which is shown to have consistent and high performance for path expressions of any complexity, including those with descendant operators. Evaluating queries without wildcards tends to be independent of the number of query terms in practice. Our mechanism always allows path expressions to be evaluated with a single index access.

Recent work on indexing semi-structured data [3, 8, 2], has proposed encoding various aspects of the data. Such methods generally have the advantage that the encoding size is insensitive to key length. We extend this notion, encoding multiple aspects of a data set in a very small space (a typical encoding is no more than 8 bytes). The encodings themselves are stored in a special data structure that enables comparison of arbitrary length paths in near constant time, using a variety of bitwise operations.

Traditional path indexes tend to focus solely on locating a set of nodes. In contrast, our index supports efficient node filtering based on the location within the graph of the data. This allows us to utilise our index for more complex processing, such as evaluating path expressions with links, entire (sub) queries with path-based predicates, and ancestor operators.

The remainder of our paper is organised as follows: Section 2 gives an overview of our approach. Section 3 describes our encoding schemes. Section 4 describes the data structure used to store the encodings and the algorithms that enable constant time comparisons. Section 5 describes the physical index structure. Section 6 details the algorithms for path expression evaluation. Section 7 describes modifications which allow us to evaluate path expressions with links, and section 8 discusses ways in which our index can support wider query processing. Section 9 presents our results. Finally, we present related work in section 10 and conclude in section 11.

2 Overview

The main approach to path indexing for semi-structured data [8, 13] has been to materialise and index the data schema. Conceptually, our approach employs a novel mechanism to encode each materialised schema path (MSP) in a manner that both minimises space and allows constant time comparison with other encoded MSPs.

2.1 Focused Searches

In order to dynamically process queries which contain wildcards, it is necessary to search a portion of the data (or a representation of it). Any such search has a number of different physical execution plans. The optimal plan depends on the values in the database, the shape of the data and the available indexes.

To ensure the optimal plan, we utilise a multi-phase method called a focused search. This seeks to choose ”foci” which reduce the search space, and only search within this reduced search space.

The first phase is to locate any query term(s) which significantly reduce the search space, and choose these as a focus. (Choosing a focus is discussed in detail in section 6). Focused searching requires that we always select the last query term as the last focus.

The second phase is to search only paths from the root to nodes which match the focus. Paths are only matched against the path expression up to the focus. If the focus has been chosen well, the new search space can reduced by many orders of magnitude.

If we have not matched the entire path expression, we repeat the first two phases. Subsequent comparisons only search paths from previous matches to the next focus. For example, suppose we are evaluating the query `//A//B//"5"` (ie. A query in soda which finds all character data with value `"5"` with ancestor element `B`, which has an ancestor `A`). Figure 1 shows some atomic values for 2 sample databases, and shows the foci most appropriate for each physical query plan.

In database (a), the first focus is `B`. The focused search first obtains all paths which terminate at `B`. The initial portion of the query (`//A//B`) is then evaluated against these paths. The focused search then evaluates the remainder of the query (`//"5"`), considering only paths from previous matches (node `B`) to descendants which match the current focus (`"5"`).

Database (b) has only a single focus. Here the entire path expression is evaluated against paths from the root to this focus (`"5"`).

Whilst we employ focused searching for path expression evaluation, our index does not impose any order on the overall query processing mechanism. Our index can be applied equally well to traditional top down, bottom

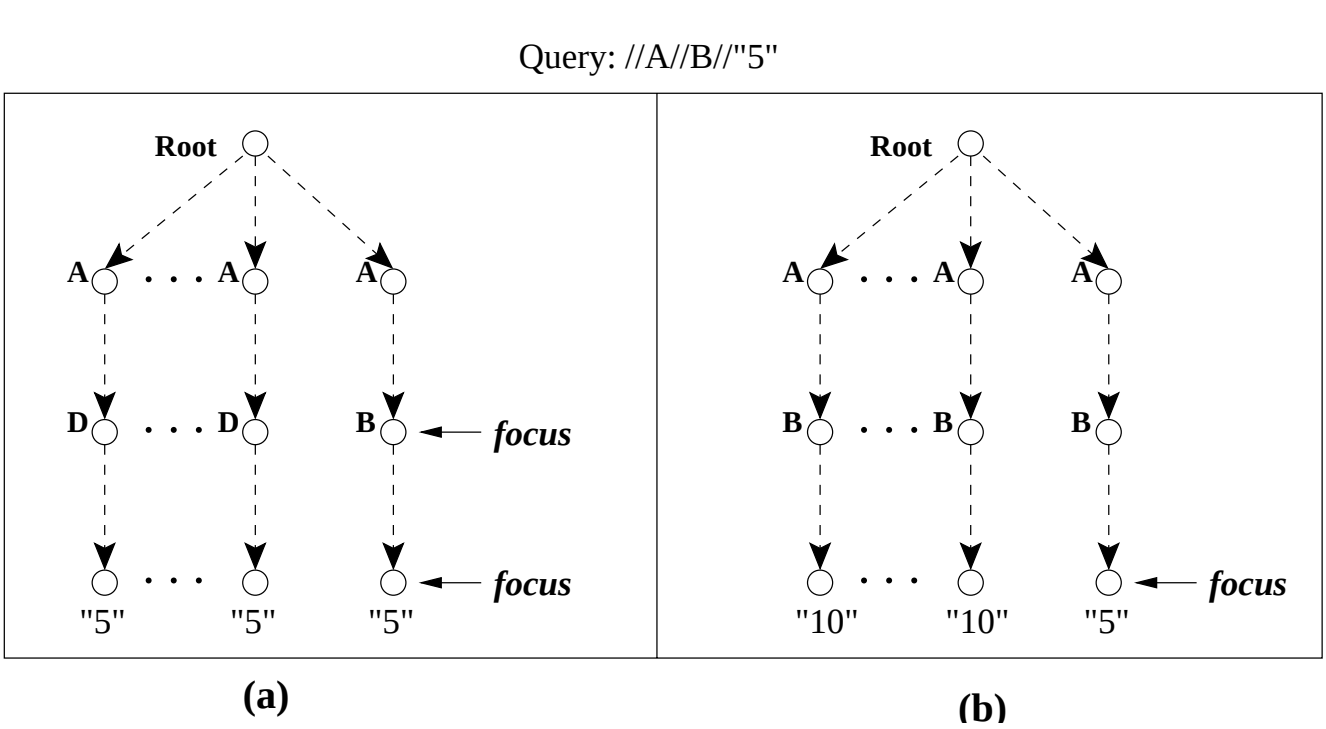

Figure 1: Different Databases and Appropriate Foci

up, or hybrid traversals (a combination of top down and bottom up, described in detail in [14]).

2.2 Index Overview

We exploit the fact that traditional indexes are accessed at some point during query processing, by using such indexes as entry points into our path index. This approach has the advantage of providing multiple entry points into our path index when a term is indexed in more than one traditional index (for example, the value ”1/5” might be indexed as both a floating point number and a string).

Queries are parsed to determine query foci, which are then located using traditional indexes. These index entries contain pointers to inverted lists containing < encoded MSP, `address` > elements, where each encoded MSP terminates at the given focus. The path expression is then evaluated against these encoded MSPs.

The `address` element of each tuple indicates the set of nodes which satisfy the given MSP. Each such set of nodes is represented by a list of <path encoding, OID> tuples. The OID is the unique element ID which enables the DBMS to look up the actual data. The path encoding is an encoding which specifies the physical location of the node within the graph of the data. This allows us to efficiently filter intermediate sets of nodes based on the physical location of the node within the graph of the data.

2.2.1 Comparison Performance

Both the list of MSP elements and the list of individual nodes are ordered in such a way which allows log(m) access, for a list with m elements. This can greatly reduce the number of elements which must be evaluated against a path expression.

The encodings themselves are designed to allow efficient comparison. Where an exact pattern is known, comparison is near constant time. This exact pattern may represent the entire pattern to be matched (eg. a query with no wildcards) or the initial segment of an encoding (eg. a node with a specific ancestor node). Even where the query does contain wildcard, comparison still tends toward constant time in practice (see sections 6.1 and 9.3). This means that query processing tends to be independent of the number of query terms in practice.

Another feature of our index is its small size, which is typically small enough to fit entirely in memory. For generality, however, the remainder of this paper assumes that the index is always stored on disk.

3 Encoding Schemes

In order to efficiently implement our focused searching mechanism, it is necessary to represent two different aspects of the data - the MSP itself, and the location of a particular node within the graph of the data.

3.1 Representing MSPs

The most basic information we must represent is an MSP. To minimise space, we map each unique, non-leaf label to the lowest unused positive integer. We refer to this number as a label identifier. This mapping has the advantage of representing a string (typically many bytes long) by a single number.

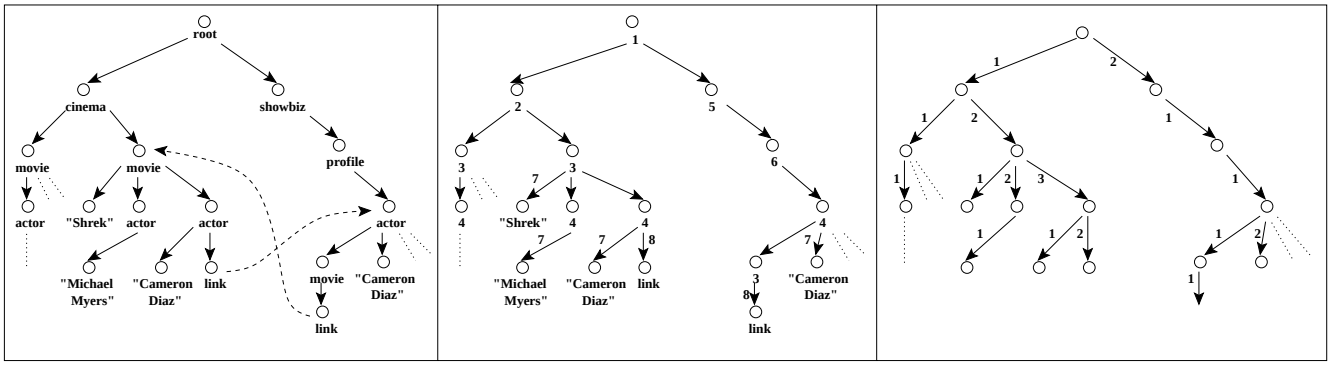

Figure 2: XML data with label and edge identifiers

Figure 2(a) shows some data from a semi-structured database. Figure 2(b) shows the label identifiers for this data. For example, we can see that the labels `root`, `movie` and `actor` have been assigned the label identifiers 1, 2 and 3 respectively.

An MSP is therefore represented by the sequence of label identifiers which represent the labels in the MSP from the root to the second last label. As MSPs are stored in an inverted list, the head of the inverted list implicitly defines the last element of the MSP. For example, the MSP `/cinema/movie/actor` is represented by the label identifier sequence 1-2-3, contained in the inverted list headed by `actor`.

Label identifiers are *not* required for leaf nodes, which will always be the head of an inverted list. This has the important consequence of reducing the number of different label identifiers by many orders of magnitude, as non-leaf nodes (which correspond to schema elements), tend to be restricted to a relatively limited range of values.

Our data structure exploits the low numerical value of the label identifiers, by only utilising twice the minimum space required to represent the numbers. For example, as the number ”1” is represented by 1 bit, and the number ”2” by 2 bits, the encoding ”1-1-2” is represented in only 8 bits (2×4 bits). This provides great space saving over methods which typically use a 4 byte integer for each number (thus requiring 12 bytes instead of 1 to represent the previous encoding).

3.1.1 Representing Node Types

Semi-structured data may include different kinds of nodes. We indicate different element ”types” by assigning a particular label identifier for different element types. The most common element type is assumed as the default if no other element type is specified.

For example, with XML data the most common element type is the normal element, and so this type is assumed if no other type is specified. In figure 2(b), character data and links are indicated by the label identifiers 7 and 8 respectively. These label identifiers can be seen on the links leading to the relevant nodes. The MSP which specifies `/cinema/movie/"Shrek"` is given by the sequence of label identifiers 1-2-3-7, stored in an inverted list headed by `"Shrek"`.

3.2 Representing Individual Nodes

We utilise a similar method to represent the physical location of a node within the graph of the data set. Each edge in the database is assigned the smallest unused positive number which is unique *only amongst edges originating from a given node*. This means that two edges can be assigned the same number as long as they originate from

different nodes. This number is referred to as an edge identifier.

Individual paths are specified as a sequence of edge identifiers, and nodes are identified as being the terminus of a given path. We refer to the sequence of edge identifiers as the *path encoding*.

Figure 2(c) shows the edge identifiers assigned to the data from figure 2(a). For example, the node `"Shrek"` is indicated by the sequence of edge identifiers 1-2-1. Note that this sequence of edge identifiers both uniquely identifies the node itself and the path from the root to the node.

This method of identifying nodes does not preserve the sequential ordering of tags within the XML document. We have developed a mechanism for preserving sequential order, but space limitations prevent us from presenting those techniques here.

This encoding has the important property that all ancestor nodes are included in the path encoding for each node. In the above example, the encoding for node `"Shrek"` indicates that its parent (the second `movie` child of `cinema`) has the path encoding `1-2` and its grandparent (`cinema`) has the path encoding `1`. This property is especially important for locating nodes with a particular ancestor.

4 Compressed Arrays

In order to efficiently implement our encoding schemes, we utilise a data structure called a *compressed array*. Compressed arrays, first proposed in [3], are optimized for comparing and examining sequences of numbers. These data structures have the advantage of substantially reduced space requirements for storing sequences of numbers, whilst allowing arbitrary portions of the sequence to be directly examined and compared utilising the compressed encoding.

4.1 Compressed Array Data Structure

Conceptually, a compressed array can be thought of as a pair of parallel bit patterns. One pattern, the *identifier pattern*, contains the bit patterns necessary for representing numbers. The bit patterns which represent two numbers frequently follow directly on from one another, with no space between the most significant bit of one number and the least significant bit of the second.

The second bit pattern, the *boundary pattern*, contains a set bit which denotes the boundary (or most significant bit) of a particular cell in the corresponding identifier pattern.

Figure 3 shows a compressed array storing the sequence of integers (from right to left) 1-3-1.

Each cell of a compressed array contains a single number. The bits relevant for a given cell are indicated by set bits in the boundary pattern, as seen in figure 3.

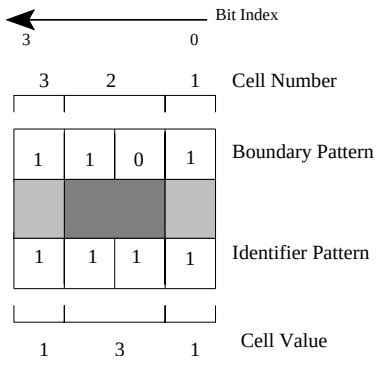

Figure 3: Example of a Compressed Array

We assume that compressed array ca has identifier pattern $ca.ident$ and boundary pattern $ca.bound$.

4.2 Comparing sequences

The most basic, non-trivial comparison is to determine whether one sequence is contained within another. Looking ahead a little, we see that this operation is utilised during our modified regular expression matching (discussed in detail in section 6.1).

The sequence in compressed array ca_1 exists within compressed array ca_2, between indices i and j, iff:

$ca_1.bound$ = ($ca_2.bound$ AND le_lssb(1 << j)) >> i

$ca_1.ident$ = ($ca_2.ident$ AND le_lssb(1 << j)) >> i,

where $j = i$+MaxIdx($ca_1.bound$)

The function le_lssb(num) obtains a mask with all bits less than or equal to the least significant set (*lssb*) bit of num set, and all others unset. This is easily implemented by:

le_lssb(num) = NOT(num XOR NOT(num - 1))

MaxIdx(num) is a non-iterative function which returns the index of the most significant set bit in num.

The method itself is constant time, with each step comprised of a very fast series of bitwise operations. Implementation of this method, however, requires that it be included as the main body of a loop, which may repeat if the compressed array is too large.

In our implementation we stored each of the boundary pattern and identifier pattern in a 4 byte unsigned integer. It is worth noting, however, that even in our index of over 3,000,000 nodes, each compressed array was stored in only two integers (one for each of the boundary and identifier patterns). Thus, implementation of this method tends toward constant time in practice.

This has important consequences for queries with no wildcards, which can be represented in a compressed array as a fixed sequence of label identifiers (discussed in section 6.1). This means that in practice, such queries can be be fully evaluated against a given MSP in constant time, which enables query evaluation time independent of the number of query terms for these queries.

4.3 Numerographic Encoding Comparison

Looking ahead again, we see that we employ a modified binary search which compares sequences based on numerographic order, rather than individual numeric values. Numerographic order is similar to lexicographical order, except that it only considers sequences of numbers, and comparisons are based on the value of each number, rather than the ASCII value of each digit. For example, the sequences $\{$1-2-7, 1-3, 1-3-2-4, 1-3-2-10$\}$ appear in numerographic order. Note that if lexicographic ordering was used, the order of the last two sequences would be reversed.

Rather than compare each sequence element one by one, the algorithm in figure 4 provides method for comparing entire sequences in constant time. As with all algorithms dealing with compressed arrays, this one makes heavy use of bitwise operations.

Conceptually, the above algorithm first checks if the two compressed arrays are equal (steps $[1]$ and $[2]$). If not, it finds the cell where the patterns diverge (step $[3]$) and obtains a mask to include this cell (steps $[4]$ and $[5]$). Finally, the numbers in the relevant cells are compared (steps $[6]$ through $[9]$).

The $FirstDiff$ mask obtained in step $[3]$ contains unset bits whilst ca_1 and ca_2 are the same (considering the array from least to most significant bits). We are only interested in the *lssb* of $FirstDiff$, which indicates the first bit position where the compressed arrays diverge. Whilst this *lssb* may occur in the middle of a cell, we wish to compare the value of the entire cell.

Steps $[4]$ and $[5]$ generate masks which cover the full cells in which the patterns diverge. Strictly speaking, they also cover all previous cells. This effectively increases the value of the final number, by adding an arbitrary number of ”low bits” to the numbers compared in step $[6]$. However, as these low bits are guaranteed to be the same for both patterns, they will not effect the *relative* relationship of the numbers being compared. Including

Algorithm: Numerographic Encoding Comparison
Input: Two compressed arrays, ca_1 and ca_2
Output: -1 if ca_1 is numerographically less than ca_2
0 if ca_1 and ca_2 are numerographically equal
1 if ca_1 is numerographically greater than ca_2
NOTE: Lower case "and" and "or" refer to logical operators.
Upper case "AND", "OR", and "XOR" refer to bitwise operators.

[1] If ($ca_1.bound == ca_2.bound$ and $ca_1.ident == ca_2.ident$)
[2] return(0)
[3] $FirstDiff$ = ($ca_1.bound$ XOR $ca_2.bound$) OR
($ca_1.ident$ XOR $ca_2.ident$)
[4] ca_1Mask = le_lssb($ca_1.bound$ AND ge_lssb($FirstDiff$)
[5] ca_2Mask = le_lssb($ca_2.bound$ AND ge_lssb($FirstDiff$)
[6] If ($ca_1.ident$ AND $ca_1Mask < ca_2.ident$ AND ca_2Mask)
[7] return(-1)
[8] else
[9] return(1);

Figure 4: Numerographic Encoding Comparison Algorithm

these bits reduces additional processing required to fully isolate the cell in question.

The function le_lssb(num) is the same as described in section 4.2. Similarly, ge_lssb(num) obtains a mask with all bits greater than or equal to the *lssb* of num set, and all others unset. This is implemented by:

ge_lssb(num) = num XOR NOT(num - 1) OR num

Whilst the algorithm itself is constant time, implementation of the algorithm requires that the algorithm be included as the main body of a loop. For the reasons discussed in section 4.2, however, the algorithm tends toward constant time in practice.

4.3.1 Comparing the Beginning of an Encoding

The algorithms presented in sections 7 and 8 require that we compare a pattern in one compressed array with the *beginning* of a second compressed array. This is easily achieved by making the following addition to the algorithm presented in figure 4.

Step $[3a]$ simply checks if anything exists in ca_1 after the point of divergence. If not, this means that all of ca_1 must have matched ca_2. Thus, with the addition of another efficient bitwise operation, we are able to check that one pattern matches only the beginning of another.

⋮

Output: -1 if ca_1 is numerographically less than the start of ca_2
0 if ca_1 is numerographically equal to the start of ca_2
1 if ca_1 is numerographically greater than the start of ca_2

⋮

[3a] If (($ca_1.bound$ AND ge_lssb($FirstDiff$) == 0)
[3b] return(0)

5 Index Structure

As mentioned in section 2, MSPs are stored in inverted lists, where the head of the inverted list represents the *last* element in the MSP. Elements of the inverted list appear contiguously within each page. For an unfragmented index, pages are also contiguous. This has the important consequence that the entire list can be accessed and examined with only a single random disk seek[1] to locate the start of the list.

Each element in the inverted list is comprised of a < encoded MSP, address> tuple, where the encoded MSP is stored in a compressed array. The address element points to the head of a list containing individual nodes which satisfy the MSP.

The list of individual nodes is comprised of < path encoding, OID> tuples, where the path encoding is also stored in a compressed array. (Tuples for nodes representing links have a different format, discussed in section 7). Such elements are also contiguous within each page, with contiguous pages for an unfragmented index.

For an unfragmented index, all of these lists are also contiguous on disk, and occur in the same order as the referencing < encoded MSP, address> elements. This has the important consequence that all relevant nodes can be found in a single forward disk scan, requiring only a single disk seek.

All lists are preceded by a header, which contains information (such as the number of elements in the list) used to facilitate efficient searching of the list.

Figure 5 shows the physical index for the inverted list headed by `actor`. For example, the MSP `/cinema/movie/actor` is represented by the label identifier sequence 1-2-3. This sequence, stored in a compressed array, is the first element of the list. This is followed by the address of the individual nodes satisfied by the MSP. Each individual node contains both the path encoding which specifies it as well as the OID.

Note also that a search for `//actor` (ie. Find all actor nodes, irrespective of the ancestors) is processed using

[1] For clarity of explanation, we do not mention any additional seeks required by the traditional indexes. Thus one random seek may translate into two or three in practice.

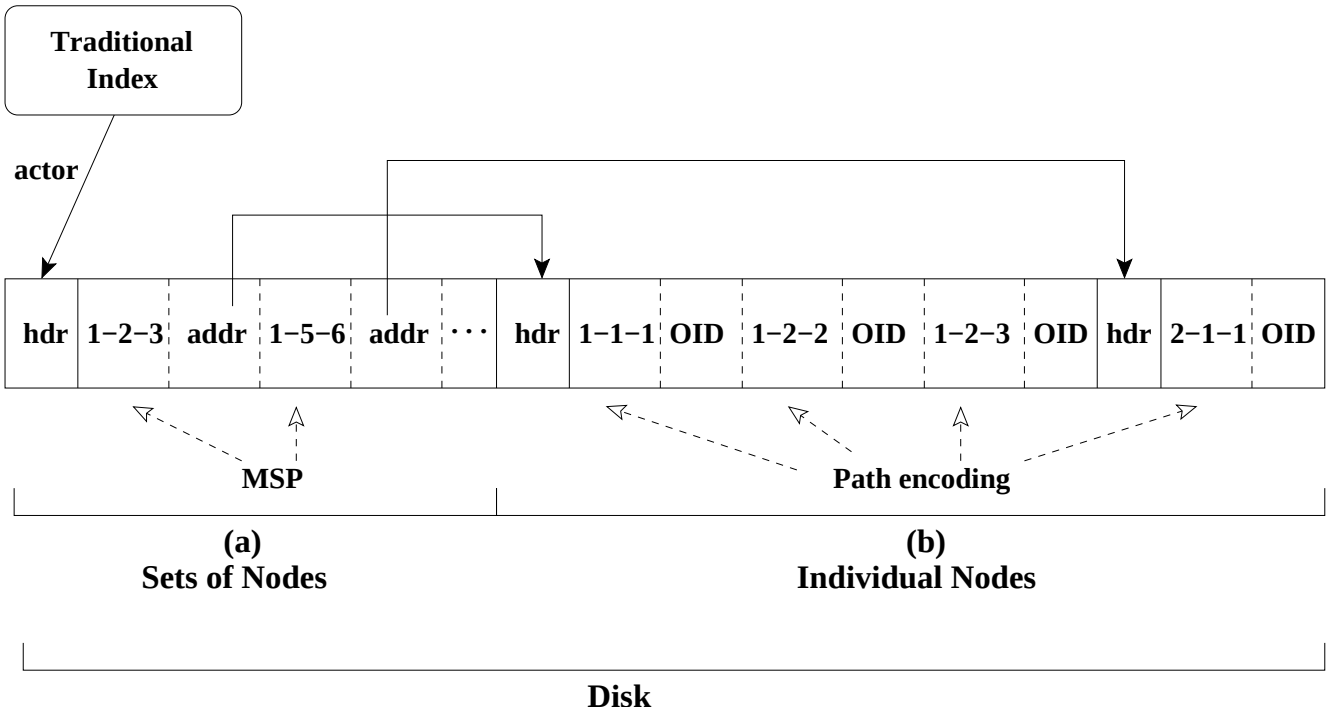

Figure 5: Physical Index Structure

a single forward disk scan.

5.1 List Element Ordering

We frequently need to locate either an element with a particular encoding, or the set of elements whose encodings share a common prefix. We facilitate searching of these lists by arranging the elements in numerographic order of the encodings. This allows logarithmic time access to individual elements using a modified binary search, using the algorithm presented in section 4.3.

Numerographic ordering has the important property that elements which share a common prefix are contiguous. This is especially important when we only wish to consider nodes with a particular ancestor.

For example, suppose we are only interested in considering `actor` nodes which belong to the second `movie` child of `cinema` from figure 2. The relevant `movie` child is specified by the path encoding 1-2. Accessing these nodes in the index is a simple matter of first finding the appropriate set of nodes (the first header in section (b) in figure 5), and then locating the first node whose path encoding begins with 1-2. We can then see that all descendants of this node appear contiguously in the list (represented by the path encoding 1-2-2 and 1-2-3). Once a path encoding no longer begins with 1-2 we know we have finished considering all relevant nodes.

Note that this property is only dependent on the numerographic ordering of the *encodings*, not the order of the

siblings in the graph. For example, suppose we insert a new `actor` node between these two siblings, and assign it an edge identifier of 4. The physical ordering of these nodes in the graph is represented by the order (1-2-2, 1-2-4, 1-2-3). The numerographic order reverses these last 2 elements. Despite the fact that the numerographic order does not reflect the physical ordering, the property that nodes with a common ancestor are contiguous is preserved.

6 Evaluating Path Expressions

All path expression evaluation shares common initial steps. These steps are shown in figure 6.

[1] Obtain the label identifiers for each label in the path expression.
[2] Use label statistics to determine appropriate foci.
[3] Generate modified, simplified regular expression(s) to represent path expression(s).

Figure 6: Initial Stages for Path Expression Evaluation

The first step (step [1]) is to map the strings used in the path expression to the relevant label identifiers which are used in our encodings. Each label identifier is stored in a hash table. As only a small number of label identifiers usually exist, this hash typically fits into memory. If a label does not exist in the hash table, the path expression contains no matches, and no further processing is required.

Foci are chosen in step [2] so as to minimise the estimated search time. This estimation is based on the selectivity of the potential focus and the degree of fragmentation for the given inverted list of MSPs. Statistics for all these factors are maintained in the label identifier hash table, and so require no additional I/O.

6.1 Modified Regular Expression Matching

In step [3] of figure 6, we generate a modified, simplified regular expression. This regular expression is applied to each candidate MSP to determine whether it satisfies a given path expression.

Basic path expressions are transformed to modified regular expressions using the following mapping:

[1] label wildcards map to any label identifier (the regexp notation ".")
[2] path wildcards map to 0 or more label identifiers (the regexp notation ".*")
[3] fixed path segments are represented the sequence of label identifiers which indicate the path.

For example, using the label identifiers from figure 2, the path expression `//showbiz/profile/*/movie` is represented by the regular expression ".* 5-6 . 3".

In evaluating these simplified regular expressions, we represent sequences of label identifiers in compressed arrays. The internal representation of the above regular expression becomes PATH_WILDca_1LABEL_WILDca_2, where the tokens PATH_WILD and LABEL_WILD represent the symbols ".*" and "." respectively, and ca_1 and ca_2 are compressed arrays containing the sequences 5-6 and 3 respectively.

This regular expression is then evaluated against each candidate MSP. The best case performance for regular expression matching occurs if the query contains no wildcards. In this case, the regular expression is typically achieved in constant time, using a single compressed array comparison as described in section 4.2. Thus, for example, we can compare the sequence 5-6 in constant time, rather than considering each element individually.

If the query contains path wildcards, but no label wildcards, we must potentially check tokens following the wildcard(s) against each element of the MSP, resulting in O(max(m, q)) operations for a query and MSP with q and m terms respectively. The least efficient expression to evaluate is one combining both path and label wildcards. In this case, it is possible that the regular expression matching will need to backtrack, resulting in O($m \times q$) operations.

In practice, regular expressions derived from queries typically contain less than 10 tokens (which would correspond to a path expression with 5 wildcards and an arbitrary number of query terms), and MSPs typically contain less than 20 terms. In practice, therefore, the worst case performance tends to have an upper bound of O(20 $\times$ 10). As such, evaluating a regular expression against an encoding tends toward constant time in practice, even where the regular expression contains wildcards.

6.2 Single Focus Processing

The simplest path expression to evaluate is one with only a single focus. Figure 7 shows the algorithm for processing a path expression using a single focus.

Algorithm: Single Focus Processing
Input: The path expression to be evaluated.
Output: The set of OIDs which match the path expression.

[1] Use the traditional index to find the inverted list, headed by the focus, of $< MSP, \texttt{address} >$ elements
[2] For each MSP which matches the path expression.
[3] Append `address` to $MspList$
[4] For each `address` in $MspList$
[5] Return the OID of each node in the list starting at `address`.

Figure 7: Single Focus Processing Algorithm

Step [1] corresponds to the operations described in section 5, by which the relevant list of MSPs is obtained. This process is achieved using only a single disk seek. As evidenced by this step, index access is predicated on the assumption that a path expression does not end with a label wildcard. In practice, this tends to be true for most queries. If a path expression does end with a label wildcard, an alternative supporting index can be used.

Step [2] is determined using the modified regular expression matching process described in section 6.1. As discussed in section 6.1, this regular expression evaluation can be considered a constant time operation.

The cost of step [2] is primarily determined by the number of elements in the MSP list which needs to be compared. In the best case, the query contains no wildcards, and the appropriate MSP can be located using the modified binary search in O(log(M)) comparisons,

If a query does not start with a wildcard, we can utilise the modified binary search to locate the first element of the list to be matched. Furthermore, because the list is in numerographic order, we only need to perform sequential scan of the list from this first element until initial token in the regular expression no longer matches. The worst case occurs when a query begins with a wildcard. In this case, the entire list must be scanned, requiring O(M) comparisons.

Step [2] always considers elements in a forward sequence. This means that step [4] guarantees elements are added to $MspList$ in the same order they appear in the MSP list. As the element order of the MSP list corresponds to the physical ordering of the individual node lists on disk, this means that for an unfragmented index, iterating through $MspList$ in step [4] and scanning the list in step [5] is guaranteed to consider nodes with a single forward disk scan.

Thus the *entire* processing for a path expression requires only a single disk seek (step [1]), with at most O(M) comparisons required.

6.3 Multi-Focus Processing

The algorithm in figure 8 describes multi-focus processing. Conceptually, this algorithm works as follows:

For the algorithm in figure 8, steps [2] and [3] correspond to point a above. Steps [5] through [9] correspond to point b when processing the first focus, whilst steps [11] through [22] correspond to the same point for subsequent foci. The difference between processing the first and subsequent foci is that subsequent foci use results from previous stages to limit the search space (point c). Step [12] corresponds to point c(i), and step [18] corresponds to point c(ii).

Strictly speaking, the above algorithm does more processing than required for basic multi-focus processing.

a The path expression is evaluated in stages, considering portions bounded by successive foci.

b Each stage obtains the set of nodes which satisfy the path expression up to the current focus. Each node is represented by the MSP describing the set to which it belongs, and the path encoding which uniquely identifies both the node and its physical location in the graph.

c The search space for all stages except the first is reduced by using the set of nodes from the previous iteration to ensure that:
   (i) Only MSPs with a valid prefix are searched.
   (ii) Only nodes with a valid ancestor are searched.

Basic multi-focus processing seeks to minimise the search space for candidate MSPs, without any reference to the physical location of individual nodes within the graph. Once valid MSPs are obtained, all nodes which match the MSP are returned. Thus, for basic multi-focus processing, there is no need to filter nodes based on their physical location within the graph. This means that maintaining and utilising the list of $ValidNodes$ (steps [7] through [9], [17], [20] and the "where" clause in step [18]) are not necessary for such processing.

Looking ahead a little, however, we see that filtering nodes based on their physical location within the graph is the basis of efficient link processing (section 7) and query processing support (section 8). As the above algorithm, including individual node filtering, is the basis of both of these techniques, we have chosen to include these additional steps here.

The improved efficiency of this algorithm comes from limiting the search space in points c(i) and c(ii) above, by materialising the prefix of valid MSPs and paths. As described in section 5.1, this allows us to find the subset of elements to consider using a modified binary search rather than a sequential scan.

In the worst case, the amortized cost of the binary search is logarithmic. In practice, however, it is frequently less. As all lists are stored and considered in numerographic order, elements of both $MspList_i$ and $ValidNodes_i$ list are also guaranteed to be in numerographic order. This means that if a pattern p_j in $ValidNodes_i$ matches the prefix of element e_k, the modified binary search for matching pattern p_{j+1} need only consider elements from e_{k+1} on. This is true for the inverted list of MSPs and the lists of path encodings.

As each disk based list is considered in forward sequence, steps [5] through [9] and [11] through [20] ensure that $MspList$ and $ValidNodes$ lists always exist in numerographic order, For the reasons discussed in section 6.2, this guarantees that all disk based lists for a single focus are examined in a single forward scan.

Algorithm: Multi-Focus Processing
Input: The path expression to be evaluated.
Output: The set of OIDs which match the path expression.

[1] Let the path expression being evaluated have n foci, $n \geq 2$, for the current query execution plan
[2] For each focus, f_i, of the path expression
[3] Perform step [1] of figure 7, considering only the path expression up to f_i,
to find the list of $< MSP,$ `address` $>$ elements contained in the inverted list headed by f_i.
[4] If $i == 1$
[5] For each materialised schema path, MSP, which matches the path expression up to f_1
[6] Append $< MSP,$ `address` $>$ to $MspList_1$
[7] For each $< MSP,$ `address` $>$ in $MspList_1$
[8] For each path encoding, PE, in the list which starts at `address`
[9] Append $< MSP, PE >$ to $ValidNodes_1$
[10] Else // $i > 1$
[11] For each element, $< MSP,$ `address` $>$, in $MspList_{i-1}$
[12] For each MSP element which starts with MSP
[13] For each element, $< MSP',$ `address`$' >$, where MSP' matches the regular expression up to f_i
[14] Append $< MSP',$ `address`$' >$ to $MspList_i$
[15] For each element, $< MSP',$ `address`$' >$, in $MspList_i$
[16] Go to head of list indicated by `address`$'$
[17] For each element, $< MSP, PE >$ in $ValidNodes_{i-1}$, where MSP' starts with MSP
[18] For each element, $< PE'$, OID $>$, where PE' starts with PE
[19] If $i \neq n$
[20] Append $< MSP', PE' >$ into $ValidNodes_i$
[21] Else
[22] Return OID

Figure 8: Multi-Focus Processing Algorithm

6.3.1 Algorithm Cost

Each focus results in only a single disk seek to locate the head of the inverted list of MSPs (step [3]). Thus, the the entire algorithm requires only n seeks (for n foci).

As discussed in section 6.1, evaluating a path expression against a single MSP or path encoding approaches constant time in practice.

Thus, for each iteration of the main algorithm (step [2]), the cost is determined by the number of comparisons required for considering the number of MSPs (steps [5], [11] and [12]) and the number of path encodings (steps [17] and [18]). The following discussions assume that $MspList$ and the disk based MSP list have p and q elements respectively.

Considering the cost of comparing the disk based MSP list, we see that the cost of the first iteration is the same as discussed in section 6.2, which is O(q) in the worst case.

The cost of subsequent iterations is determined by steps $[11]$ and $[12]$, which find each element in the disk based list which starts with an MSP contained in $MspList$. We see that the arrangement of the two lists allows us to choose between two possible comparison strategies.

The simplest strategy is to iterate through each element of both lists. As both lists are maintained in numerographic order, this iteration can be done in parallel, which is O($p + q$).

The second strategy is to obtain each element from $MspList$, and locate the appropriate element in the disk based list using a modified binary search. As discussed, numerographic order means that finding an element in the disk based list which starts with a given MSP is O(log(q)). Thus, for p elements, the worst case cost is O($p$log(q)).

As previously mentioned, however, the expected cost is less than this, as each match reduces the search space for future matches. For a random distribution of MSPs, we can expect the first match in the disk based list at the $\frac{q}{2}^{th}$ element, which reduces the search space for the next element to $\frac{q}{2}$. Using this rationale, we see that the expected search time for p elements is O$\Big(\log(q) + \log\left(\frac{q}{2}\right) + \ldots + \log\left(\frac{q}{2^{p-1}}\right)\Big)$ = O$\Big(\log\left(\frac{q^p}{2^p-1}\right)\Big)$.

It is worth noting that foci are largely chosen for their selectivity. This means that frequently $p \ll q$, making the latter strategy more appropriate.

Where node filtering is performed, symmetrical calculations apply for $ValidNodes$ and the path encoding lists in steps $[17]$ and $[18]$.

6.3.2 Tracing an Example

The effectiveness of multi-focus processing can most clearly be seen using an example. Consider the query applied to database (a) in figure 1. The first iteration of the algorithm iterates through each element in the inverted list headed by ”B”. As this list contains very few elements (1 in this example), such iteration is quick. Each matching MSP from this list is stored in $MspList_1$.

The second iteration considers all MSPs in the inverted list headed by ”5”. Note that the actual query we are evaluating starts with a path wildcard. Thus, if we were to evaluate the entire path expression only considering this list, we would need to evaluate the path expression against every MSP in the list. As this list contains many MSPs, this would require substantial processing.

As we have materialised the beginning of the MSP, however, we can locate the position to begin searching using the modified binary search described in section 4.3. As such, this iteration is O(log(M)) instead of O(M),

for an MSP list with M elements.

6.4 Updates

The most frequent updates to an XML database are the insertion or deletion of entire subtrees (which corresponds to inserting or deleting entire ”elements” from a document), or changing the value of a leaf node (frequently character data or an attribute value).

For each of the above operations, the appropriate conventional index must be updated for each node in the subtree. After such updates, the appropriate MSP and path encoding are inserted or deleted. Insertion of a node can potentially cause index fragmentation. For this reason, a buffer is left at the end of each list to reduce this likelihood. Fragmentation is incrementally reduced through periodic index maintenance.

The most expensive updates involve changing (updating, inserting or deleting) individual, non-leaf nodes in isolation. Recall that the encodings for a given node are contained in the encodings of all its descendants. Such changes must therefore be reflected in both the node itself and all descendants. To facilitate this, we build an alternative index on the encodings, which allows encodings to be retrieved based on included nodes, rather than just on foci labels.

7 Links

Links (such as hypertext links), are an important extension to the XML specification [9], and are a fundamental feature of many XML documents. For this reason, we have incorporated a ”link” operator (–>) into our query syntax. The link operator can appear anywhere a path separator (/) can appear. Thus, //B –> C, specifies a path which starts at a node, ”B”, follows any link originating at this node and terminates at a node ”C”.

Whilst it is always possible to ”follow” links using traditional joins, such a mechanism is relatively expensive, especially when dealing with paths. By making minor adjustments to our mechanisms, we are able to efficiently process path expressions which contain links.

7.1 Encoding Scheme

In order to encode links, we include a virtual node in the graph to indicate the node. This can be seen in figure 2 by the ”link” nodes. ”Link” nodes are only included if the target of the link exists in the database.

Such link nodes only exist in the index itself. The labels in the XML document which indicate a link (< xmlns:xlink . . . > etc) exist as normal nodes in the index. (These are omitted from figure 2 due to space).

As discussed in section 3.1.1, indicating a special link node is achieved by assigning a particular label identifier to represent this type of node.

7.2 Physical Structure

Due to their different nature, link nodes have a different physical index structure to other nodes. The physical index structure for the link nodes in figure 2 is shown in figure 9.

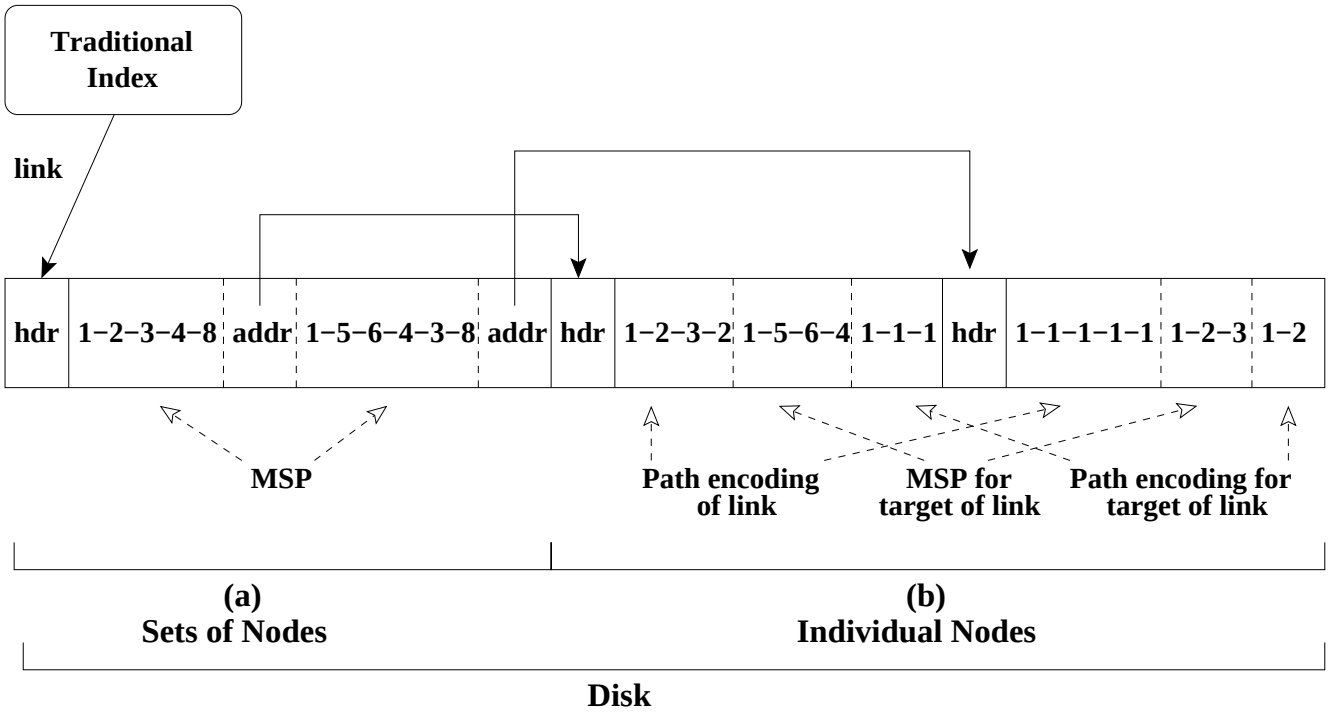

Figure 9: Physical Index Structure for Links

The inverted MSP list is the same for link nodes as for other types of nodes. The difference lies in the various path encoding lists.

Elements of the individual node list comprise < link path encoding, target MSP, target path encoding > tuples. Conceptually, these elements can be thought of as materialising the links.

Recall that any node in the index is specified by its MSP and path encoding. The target MSP and path encoding therefore specify the other end of the link.

Note that as link nodes are virtual nodes, they have no physical occurrence within the database, and therefore no associated OID.

For efficient update, we maintain an index (conceptually containing reverse pointers for the target node to the link), so that link nodes can be appropriately updated to reflect any change in the target nodes.

7.3 Processing

Links are processed in a similar fashion to multi-focus processing, discussed in section 6.3. Each link in the path expression is always chosen as a focus. The multi-focus processing algorithm given in figure 8 is modified to process links by adding the following steps:

$[7a]$ If f_1 represents a link
$[7b]$ For each element, $< PE, MSP_{tgt}, PE_{tgt} >$, in the list which starts at `address`
$[7c]$ Insert MSP_{tgt} into $MspList'_i$ in numerographic order
$[7d]$ Insert $< MSP_{tgt}, PE_{tgt} >$ into $ValidNodes_1$ in numerographic order
$[7e]$ else // continue with step $[8]$ of figure 8
⋮
$[9a]$ Replace $MspList_i$ with $MspList'_i$
⋮
$[17a]$ If f_i represents a link
$[17b]$ For each element, $< PE', MSP_{tgt}, PE_{tgt} >$, where PE' starts with PE
$[17c]$ Symmetrical steps to $[7c]$ through $[7e]$ and $[9a]$

As mentioned in section 7.2, PE list elements can be thought of materialising the link. The above steps conceptually "follow" this materialised link, and place the endpoint on the queue of valid nodes to be considered. This new endpoint is then used to limit the search space for successive foci in exactly the same way as the method described in section 6.3.

As mentioned in section 6.3, individual path encodings and the temporary $ValidNodes$ list are only effective for link processing and query processing support. We can now see how these significantly reduce both the search space and the amount of required processing.

Individual links indicate specific nodes. This means it is possible for a node to satisfy an MSP, but not be referenced by a particular link. Thus, whilst all nodes with a given MSP are valid for multi-focus processing, when processing links we need to ensure that both the MSP *and* the specific node are valid.

Individual path encodings and the temporary list $ValidNodes$ provide an efficient means of facilitating this additional processing. By steps $[7]$ and $[15]$ of figure 8, we have already identified all MSPs which contain at least one valid node.

Step $[18]$ allows us to efficiently filter nodes based on their physical location within the graph. Recall that for path encodings, the encoding of all ancestor nodes is included in the encoding for a particular node. Thus,

by selecting only path encodings where the prefix is in $ValidNodes$, we filter for nodes with a valid ancestor. Determining if one encoding begins with another is efficiently achieved in constant time using the method described in section 4.3.1. As such, the overall filtering cost is the same as described in section 6.3.1.

Unlike the multi-focus processing algorithm described in figure 8, no assumptions can be made about the order in which link endpoints are added to the various $ValidNodes$ lists. As such, these elements must be inserted into these lists in numerographic order. This adds O($p \log(p)$ + $v \log(v)$) (for an $MspLsit$ and $ValidNodes$ list with p and v elements respectively) to the overall complexity of the multi-focus processing algorithm described in section 6.3.

8 Query Processing Support

Rather than purely enabling path expression evaluation, our indexing mechanism also enables support for the overall query processing process. Minor modifications to the algorithms presented in section 6, enable processing of additional path-based query constructs such as path-based predicates and path related query operators. Whilst such query constructs are potentially expensive for the underlying database to process, the mechanisms presented here mean that the processing is no more expensive than an additional index access. Due to space restrictions, we present the modifications required for path-based predicate processing. Similar modifications allow efficient processing of path-based operators, such as ancestor operators.

8.1 Query Predicates

Path-based query predicates (or ”branchy” queries) are potentially expensive for the underlying DBMS to process. Whilst the predicates themselves may be arbitrary path expressions, typical processing may require multiple index lookups, joins and/or pointer chasing to evaluate such queries.

Path-based predicates indicate that nodes in the result set must exist within a conforming sub-graph, rather than just a conforming path. This sub-graph is typically specified as a central path (indicated by the non-predicate path expression) with one or more secondary paths leading off the central path (specified by the predicate path expressions).

It is possible to employ our index for such predicate processing by making some modifications to the algorithm in figure 8. The modifications differ slightly depending on whether we are evaluating a predicate or non-predicate path expression. Due to space restrictions, the modifications described here assume that all predicate path expres-

sions are evaluated before the non-predicate path expression. Symmetrical modifications are used to generalise this to all cases.

We make modifications to the following steps:

Let NCA$_p$ be the nearest common ancestor (NCA) of the secondary path indicated by predicate path expression p and the central path.

A For each predicate path expression, p

[22] Insert $< MSP_{NCA}, PE_{NCA} >$ into $PredicateList$ in numerographic order, where MSP_{NCA} and PE_{NCA} are the segments of MSP' and PE' which terminate at NCA$_p$.

B For non-predicate path expressions

[5] ... and which begins with MSP_p, where MSP_p is the start of MSP_{NCA} (where $< MSP_{NCA}, PE_{NCA} >$ exists in $PredicateList$) AND len(MSP_p) = min(len(MSP_{NCA}), len(MSP))
[8] Symmetrical change to [5] for PE_p
[13] Symmetrical change to [5]
[18] Symmetrical change to [8]

Predicate query processing is possible due to the fact that we encode the physical location of each node in the graph. By comparing path encodings, we are thus able to effectively perform ”graph traversal” by utilising bitwise comparisons on our indexed encodings.

The modifications can be thought of as obtaining the common ancestors which satisfy the predicates (step [23] for modification A), and projecting these onto potential candidates for the result set (modification B). Due to the nature of our encodings, however, this ”projection” is accomplished efficiently using bitwise comparisons of elements in sorted lists, rather than underlying graph traversal or relational joins.

Many query processing approaches require separate index and database accesses to materialise the set of valid common ancestors. In contrast, we obtain this set with no additional index or database access. Instead, the set of common ancestors is extracted from the path encodings (”where” clause in modification B) of nodes which satisfy the predicate(s). As nodes are filtered based on their physical location within the graph, the path encodings and temporary $ValidNodes$ list in figure 8 are important for query processing support.

A single $PredicateList$ of all valid common ancestors is maintained, irrespective of the predicate they originate from. As the list is built from successive path expression evaluations, elements need to be inserted in numerographic order (step [22], modification A).

Modification B shows how to ”project” the set of valid common ancestors on the potential result set. Concep-

tually, this step ensures nodes are only considered if they have a valid ancestor.

Recalling from section 6.3, however, we see that each iteration of the algorithm in figure 8 potentially considers less than the full path from the root to the final node. For this reason, we need to ensure that the modifications only consider the segment of the path from the root to the NCA which corresponds to the portion of the path being considered.

len(ca) refers to the number of cells in compressed array ca. If we recall that each cell indicates an edge in the path, we see that the limitations on len(MSP_p) ensure that we only consider equivalent portions of the encodings.

The searches of $PredicateList$ (both for insertion in modification A and retrieval in modification B) require only a single forward scan over the course of the entire algorithm, as both elements considered by the main algorithm and items in $PredicateList$ are guaranteed to be in numerographic order.

The additional processing required to obtain the encoding of the relevant common ancestor is a constant time operation involving efficient bitwise operations. The only cost additional to that described in section 6.3.1 involves inserting elements into $MspList$ and $ValidNodes$ in numerographic order. Thus, the overall cost of the algorithm is the same for link processing, described in section 7.3.

9 Results

We have implemented this index on a computer with a single Pentium III 800MHz processor, 256MB RAM, running linux Redhat 7.0. The data itself is stored in the soda [19] native semistructured database.

We ran tests over two sets of data which span the range of various ”types” of semi-structured data - one strictly conforming to a simple, well defined schema, and one no pre-defined structure at all. Whilst we expect that most semi-structured data will fall somewhere between these two extremes, we felt this approach gives good upper and lower bounds on the effects of the structure of the data on index performance.

For the former ”type” of data we used the complete works of Shakespeare, scaled up or down as appropriate to obtain the required data size. For the latter ”type”, we randomly generated data sets with no pre-defined structure. Each data set contained approximately 3,000,000 nodes. The random data was constrained to have approximately 1,000 1^{st} level nodes and a maximum depth of 6. Node names were selected randomly selected from a pool of 100, with the constraint that each MSP had at least 3 matching nodes.

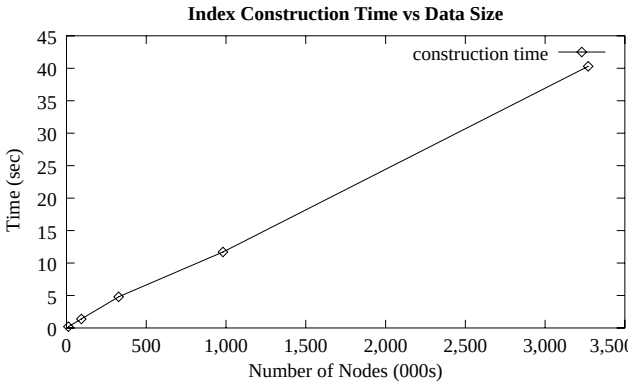

Figure 10: Index Construction Time vs Data Size

9.1 Index Construction Time

Figure 10 shows the time taken to construct the index for various data sizes. Index construction requires only a single scan of the entire database.

As can be seen the construction time is both linear with respect to data size and very fast. Care has been taken with the time measurements to ensure the data was properly clustered physically in the database.

9.2 Index Size

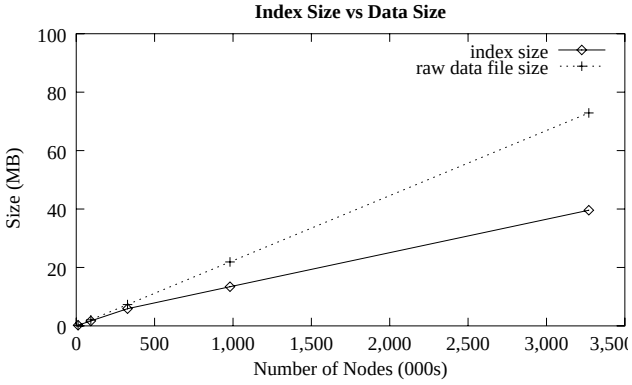

Figure 11: Index Size vs Data Size

Figure 11 compares index size with to data size. As can be seen, index size is linear with respect to data size.

The precision of the linear relationship is partially a result of our implementation. For performance considerations we stored the encoding in multiples of 8 bytes (2×4 byte unsigned integers). Whilst the various encodings require differing number of bits, storing encodings in multiples of 8 bytes tends to ”smooth” out this variation,

resulting in the linear space measurements.

Whilst data size (in bytes) can vary widely, we have included the size of the file containing the raw data, to give an indication of the relationship between the raw data size and index size.

9.3 Path Expression Length

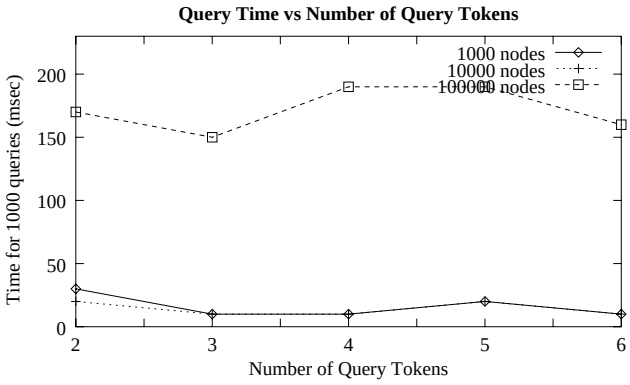

Figure 12: Number of Query Terms vs Time

Figure 12 shows the effect of the length of the path expression on query time. Queries were generated with an equal proportion containing a) no wildcards, b) path wildcards, c) label wildcards, d) a combination of path and label wildcards. In these results, a ”query term” refers to either a label or label wildcard. For example, the path expression `/a/b/*//c` has 4 query terms.

The time measurement is only for evaluating the path expression and loading the OIDs from the index. We have not included time to load the actual data, as such additional ”noise” is highly DBMS dependent.

For 100,000 nodes, the result sets tended to be significantly larger (in number of pages) and so additional I/Os were required to load the OIDs.

These results demonstrate that in practice, evaluating path expressions is independent of the number of query terms for all path expressions, including those with wildcards.

9.4 Query Cost

To demonstrate the relative effectiveness of our index, we have compared it with two other indexing techniques. Native semi-structured databases tend to be proprietary systems, who do not publish the details of their indexing techniques. Any head-to-head comparisons with these systems these would be distorted by the efficiency of the underlying native DBMS. In contrast, much work has been done in the public arena on storage and indexing of

	Query	I/O - Pages Regular Data						I/O - Pages Random Data					
		Edge Map		Cooper et al		MSP Index		Edge Map		Cooper et al		MSP Index	
		value	Δ	value	Δ	value	Δ	value	Δ	value	Δ	value	Δ
1	/a/b/c	97	1.0	64	1.5	69	1.4	194	1.0	3	64.7	9	21.6
2	//a//b//c	4207	1.0	603	6.7	607	6.6	81056	1.0	9766	8.3	349	232
3	/a/b[d]/c	135	1.0	-	-	13	10.3	235	1.0	-	-	17	13.8
4	//a//b[.//d]//c	5938	1.0	-	-	16	371	135158	1.0	-	-	694	194

Figure 13: Performance Results

XML data in relational databases. For this reason, we have chosen to compare our work with 2 other indexing schemes, implemented on a relational database. This ensures that the final results are truly independent of the underlying storage mechanism.

The first comparison is with the basic *edge-mapping* scheme, which considers the XML data as a set of nodes and edges stored in two relational tables [18].

The second indexing technique we chose was proposed by cooper et al [8], which also encodes and indexes MSPs. Encoded MSPs are stored in a modified patricia trie, where relevant nodes in the trie point to the actual data elements. Path expressions are evaluated by traversing the trie until it is either fully matched or found to have no solution. The trie can also be utilised to store pre-computed path expressions or queries. As we are focusing on ad-hoc queries, however, we have chosen to consider only MSPs stored in the index.

Due to space limitations, we have chosen not to compare our approach with other well known storage and indexing techniques, such as STORED [10], which has been shown to tend toward edge mapping for irregularly structured data [8].

Figure 13 shows the average number of I/Os to evaluate a single query of the specified type. These are based on simulations of a large number of queries over a wide range of values. Queries were randomly generated with the constraint that at least 1 match existed within the data set. The cache size was set to 10% of the data size. Our index is superior to conventional methods by at least 1 order of magnitude, becoming even more effective for irregularly structured data.

9.4.1 Single Path Expression Without Wildcards

Query **1** represents a simple path expression with no wildcards.

For both regular and random data, edge mapping is significantly slower than the other two approaches, due to the number of joins required to evaluate the path expression. Even though these joins are supported by indexes, they still require multiple index lookups to accomplish. It is significant to note that the queries were constrained

to have 3 query terms. Increasing the number of query terms results in a corresponding increase in the number of required joins, and thus additional index lookups.

For both the other approaches, the regular data required substantially more I/Os than for the random data. This is due to the fact that the regular data has a greatly reduced number of MSPs (by around 2 orders of magnitude). As such, a matching MSP has many more data elements. In fact, for both cooper et al and our approach, the index required only 3 and 8 I/Os respectively, with the remainder of the I/Os associated with loading the actual data.

For both the regular and random data, our index required slightly more I/O than cooper et al. This is partially due to the fact that our index also includes the path encoding for each node, with a corresponding increase in the amount of space required to store the list of individual nodes.

9.4.2 Single Path Expression With wildcards

Query **2** represents a simple path expression with path wildcards.

Once again, edge mapping proved to be significantly more expensive in terms of I/Os, even though the query processing was hand tuned for maximum efficiency.

To evaluate such a query, Cooper et al search the index using a top down traversal. For regular data, with a small number of MSPs, the trie is very small (fitting on 1 or 2 pages), and so the overhead is minimal. The random data, however, contained over 800,000 MSPs, spread over many pages. As such, traversing such a trie requires many I/Os.

In contrast, our approach required a single scan of the relevant MSP list. Whilst the list contained many MSPs for the random data, traversing this list required significantly less I/Os. Our approach performed between 1 and 2 orders of magnitude better than edge mapping for queries with wildcards.

The advantage of our approach becomes more pronounced as the structure of the data becomes less consistent. Such a situation becomes increasingly prevalent for large databases with heterogeneous data and/or with very loose schema.

9.4.3 Branchy Queries

Query **3** represents a branchy query with no wildcards, whilst query **4** represents a branchy query with wildcards. It is not possible to compare our work with Cooper et al, as their method does not support branchy query processing for arbitrary queries.

Edge mapping requires a significantly greater number of I/Os. If no wildcards are used, each query term,

whether in the main path expression or the predicate, results in a join. If the query contains wildcards, the entire subgraph must be materialised until the path expression can be evaluated against each candidate path.

By utilising the path encodings stored with the OIDs, we are able to perform the entire query processing without the need to visit the actual data. As mentioned in section 9.4.1, the majority of the I/Os for our index are associated with loading the actual data. As we only need to load the data for the (reduced) result set, we require a significantly reduced number of I/Os.

For all data types, our approach is one order of magnitude better than edge mapping for queries with no wildcards, and 2 orders of magnitude for queries with wildcards.

10 Related Work

The issue of storing and querying semi-structured data has gained increasing importance recently [1, 5, 7, 6, 15, 17]. A popular approach has been to store semi-structured data in a relational database. Such an approach frequently utilises the data schema, whether this be pre-defined [18], or extracted from the data through data mining [10]. Such approaches have difficulty coping with data of irregular or frequently changing structure. Florescu and Kossman have examined storing semi-structured data in a relational database as a set of edges and attributes, without any knowledge of the structure of the data [11].

An alternative approach has been to store semi-structured data in a native semi-structured database [14, 19]. Path expression evaluation in such systems tends to require multiple index lookups. Recently, Cooper et al [8] have proposed an path index which requires only a single index lookup.

The idea of using signatures to identify paths is proposed in [16]. However, performance on paths with descendant operators had not been addressed. Finally, an extensive survey on semistructured and web data, ranging from data models to query languages to database systems, was presented in [12].

Our work is also related to path navigation that has been studied in object oriented databases, in which sequences [4] are used to support long paths.

In all the systems mentioned above, except for [8], multiple index lookups are usually required for evaluating a path expression. Cooper et al [8] have proposed a mechanism for encoding paths as strings, whilst blah and blah have proposed an alternate encoding mechanism for encoding paths. These encodings are then stored in modified tries. Path expressions are then evaluated using a single index lookup by traversing this index structure. Queries with wildcards are only supported if these have been pre-computed into the index. The approach does not provide

support for processing nodes based on their location within the graph of the data set.

Focused searching is conceptually similar to list balancing presented in [2].

11 Conclusion

In this paper we have presented a method for implementing a fast, versatile path index. Our index is shown to be effective for evaluating all path expressions, including those with wildcards, as well as supporting more complex processing such as evaluating path expressions with links and evaluating entire (sub) queries with path based predicates. Performance is shown to be independent of the number of terms in the path expression, even where these contain wildcards. Our index is shown to be faster than conventional methods by up to two orders of magnitude for certain query types, irrespective of whether the data adheres to any pre-defined schema. Experiments show our index is small, fast, and scales well.

References

[1] S. Abiteboul. Querying semi-structured data. In *International Conference on Database Theory (ICDT)*. 1997.

[2] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB Journal*, p 53-64, 2000

[3] M. Barg and R.K. Wong. Structural proximity searching for large collections of semi-structured data. In *Proceedings of ACM CIKM*, November 2001.

[4] E. Bertino. Index configuration in object-oriented databases. *VLDB Journal*, 3(3):355–399, 1994.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD*, 1996.

[6] D. Chamberlin, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. Xquery: A query language for xml. In *W3C Working Draft, 15 February 2001*, Available at `http://www.w3.org/TR/2001/WD-xquery-20010215/`.

[7] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An xml query language for heterogeneous data sources. In *Proceedings of the WebDB Workshop*, 2000.

[8] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semi-structured data. In *Proceedings of VLDB Conference*, September 2001.

[9] S. DeRose, E. Maler, and D. Orchard. Xml linking language (xlink) version 1.0. Available at `http://www.w3c.org/TR/2001/REC-xlink-20010627/`

[10] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1999.

[11] D. Florescu and D. Kossman. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical report, INRIA Technical Report 3684, 1999.

[12] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of VLDB*, pages 436–445, 1997.

[14] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[15] J. McHugh and J. Widom. Query optimization for xml. In *Proceedings of the 25th VLDB Conference*, 1999.

[16] S. Park and H.-J. Kim. A new query processing technique for xml based on signature. In *Proceedings of DASFAA*, 2001.

[17] J. Robie, J. Lapp, and D. Schach. Xml query language (xql). In *The XSL Working Group, World Wide Web Consortium*, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[18] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 25th VLDB Conference*, September 1999.

[19] R.K. Wong. The extended xql for querying and updating large xml databases. In *Proceedings of ACM Symposium on Document Engineering (DocEng)*, November 2001.