Efficient Query Relaxation for Semistructured Data

Michael Barg Raymond K. Wong
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
wong@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0312
June 2003

Abstract

Semistructured data, such as XML, allows authors to structure a document in a way which accurately captures the semantics of the data. This, however, poses a substantial barrier to casual and non-expert users who wish to query such data, as it is the data's structure which forms the basis of all XML query languages. Without an accurate understanding of this structure, users are unable to issue meaningful queries. This problem is compounded when one realizes that data adhering to different schema are likely to be contained within the same data warehouse or federated database.

This paper describes a mechanism for meaningfully querying such data with no prior knowledge of its structure. Our system returns approximate answers to such a query over semistructured data, and can return useful results even if a specific value cannot be matched. We discuss a number of novel query processing and optimization techniques which enable us to perform our query relaxation in near linear time. Experiments show that our mechanism is very fast and scales well.

1 Introduction

The richness of the XML data format allows data to be structured in a way which accurately captures the semantics required by the author(s). Such requirements are naturally influenced by the purpose of the author(s), as well as the context in which the document is being written, resulting in data with the same *semantic* content having vastly differing structure. This poses substantial barriers to casual users and non-domain experts who wish to query the data, as it is the structure of the data which forms the basis of all XML query languages [1].

Without at least some notion of the structure, a user cannot meaningfully query the data. This problem is compounded when one considers that heterogeneous data are likely to be included in the same data warehouse, federated database or integrated data repository [13].

Additionally, users face the problem of information with the same *meaning* being identified with differing tags. For example, a search for ”cars” may miss data which talks of ”automobiles”. Such issues are starting to be addressed by the semantic web project [18], which deals with specifying computer-understandable ontologies to describe the data. This project, however, does not yet have a mechanism for automatically making connections *between* ontologies, and is of no use if the data does not adhere to an ontology.

Suppose we are looking for trends in ”insurance claims” related to ”smoking” for ”women” over ”40”. The relevant information may be contained in insurance company records, court transcripts, or even newspaper articles. Some documents may talk of ”litigation” or ”liability” instead of ”insurance claims”. Even if we know the terminology, and are interested only in court transcripts, we do not know the structural relationship between the terms of interest. We are left in the predicament of knowing exactly what we are looking for, but not knowing how to find it.

The issue of document structure can be even more insidious for large collections of inconsistently structured documents. Consider student homepages, with XML content marked up in a manner which makes sense to individual students. Suppose we want to find all ”post graduate” students interested in ”nanotechnology” who are members of ”research” teams with ”defence department” funding. This situation involves searching a huge number of documents, all with irregular and inconsistent structure.

As more and more personal or small scale data is marked up on the web, and as the number of schemas continue to grow, this situation will become increasingly prevalent. Query relaxation provides an intersection between search engines, which typically provide a keyword search that ignores document structure and database queries, which precisely specify the desired information but are too restrictive for disparately structured web data.

Consider another example, where we wish to find the phone number of *Bob*, the new sales manager. Unfortunately, his details do not exist in the database yet. Rather than telling us there is no ”Bob”, it would be much more helpful if the database could relax the query and return the phone number of the sales department secretary instead.

Query relaxation for structured data typically involves queries where both the structure of the data and the terms are known. Answers typically approximate data values which are hopefully ”close enough” to be of use to the user.

XML data, however, poses its own more complex set of issues. As the primary barrier is the structure of the data, a more useful approach is to return an answer where the *structure* of the data approximately matches the *structure* specified by the query. To be truly useful, such a notion of ”approximate” matching should incorporate some concept of *semantic* as well as *structural* similarity. Two structures should be considered ”similar” if they are both close in the structural sense, and convey semantically similar information. Additionally, such query relaxation should ideally be able to substitute semantically equivalent terms, where such equivalence considers both the terms and the context.

Given that the most likely use is real time interactive querying sessions, processing speed is especially crucial.

For query relaxation, we consider the XML database (such as a Lore database [12]) as a general graph, comprised of one or more individual data sets. Whilst raw XML can always be represented as a tree, the *interpreted* structure (ie. where links are materialized as edges) can be an arbitrary graph. As links generally indicate a ”relatedness” between elements, we must consider this interpreted structure to maximize the effectiveness of our query relaxation.

Note that for semistructured data, it is frequently appropriate to return approximate answers even if the data *does* contain an exact match to a query. Disparately structured data may contain both exact matches and ”approximate” matches which are also appropriate for the query.

Prior Work Much effort has been recently devoted to approximating various aspects of answers to database queries. Most of this work has focused on queries where the query structure is known to be valid, but approximate values are desired. One motivation has been to provide a ”rough answer” with minimal processing for exploratory queries over large data sets [2, 7, 11, 15]. Another motivation has been to provide ”close” answers, where an exact match cannot be found [6, 9]. Some early work has begun to explore approximating the structure of the data to provide useful answers [5, 8, 10, 17].

• *Value approximations for evaluation efficiency*. Value approximations are frequently required to enable efficient evaluation of exploratory queries over very large data sets. Various approaches have focused on generating a synopsis of the of the data, through random sampling [2] or histogram based techniques [11, 15], and then processing the queries directly on these synopses. These approaches have been shown to be useful for various query constructs. Recent work has successfully applied wavelet based decomposition to approximate query answering, which enables the entire query to be efficiently processed using the wavelet-coefficient synopses [7].

• *Value approximations for query relaxation*. Another approach has been to provide approximate answers to users, in the hope they will be ”close enough” to be useful. Recent work has focused on evaluating ”Top-*k*” queries for relational databases [6, 9], which return the *k* answers which best match the users criteria. This work focuses on a range of mapping techniques which transform top-*k* queries into range queries which can be processed by the underlying RDBMS.

• *Structure approximations*. Some recent work has started to look at the issue of approximating the structure of the data for queries over semistructured data. Early approaches [5, 10] sought to rank 2 sets of nodes based on their proximity (in the structural sense) to each other, without any reference to any desired relationship contained in the query. More recent approaches [3, 8, 17] have sought to incorporate the term relationship implied in user queries. Such approaches typically seek to determine the degree to which the query and data match, primarily by observing the addition and/or absence of desired nodes. Such approaches consider only the tree structure of the raw XML document.

Our Contributions

• *Wide range of structural relaxations*. 6 classes of structural transformations exist for any structural query relaxation (see section 3). Most existing work considers only 1 class of structural transformation ([8] considers 2). We present a novel mechanism for efficiently considering all 6 classes of structural transformations. This significantly impacts the quality of the final answers.

• *Fast query relaxation*. In contrast to much of the existing work, we focus on efficiency and performance. Our query relaxation processing both very fast and scalable. Our system can rank 20,000 nodes according to how well each node satisfies the query criteria, calculating the best fit out of 4×10^8 paths in 0.17 seconds. Our mechanism scales *linearly* with both the selectivity and number of query terms, despite the exponential increase that each factor adds to the search space. This is particularly significant as most existing techniques tend to be polynomial w.r.t. database size [1] for the simplified tree structure and a more limited range of relaxations.

• *A new, context sensitive approach to mapping semantically equivalent terms*. By recognizing that ontologies and conceptual hierarchies such as wordnet [14] can be represented as graphs, we propose a

[1] [3] is linear w.r.t. the number of joins

framework to adapt our mechanism to achieve a new, context sensitive mechanism for mapping query terms to data terms. The framework seeks to utilize the semantic information contained within the graphs to map terms in the absence of any explicit schema or ontology mapping.

Overview of our Paper. In section 2 we provide an overview of the system and our approach. In section 3 we discuss the various factors involved in calculating the similarity score. In section 4 we describe the query relaxation algorithms. In section 5 we discuss the framework for extending this work to contextually and semantically equivalent term mapping. Results are presented in section 6, and we conclude with section 7.

2 Overview

2.1 The System

A native XML database called Soda2 [19] is used to test our prototype, although our query relaxation engine is independent of the underlying storage mechanism. The engine itself is built using over 17,000 lines of C code. To process relaxed queries, the database evaluates one query for each query term. The results of these queries are passed to the query relaxation engine, which utilizes a structural index to progressively evaluate the quality of the final results.

Database objects are opaque to the query relaxation engine, which only deals with object identifiers. The query relaxation engine retrieves the encodings for each object in constant time. These encodings, including scores, are progressively combined to form a representation of the relevant subgraph. Finally, objects that are ranked according to their final score are returned.

2.2 Terminology

The rest of the paper uses the following terminology. Let Q be a query over a semi-structured database. Such a query can be represented by a query tree, Q^T, which indicates the required topological relationship between target nodes. Suppose, for example, we wish to find the phone number of a restaurant in Soho. Figure 1 shows the query tree for `/restaurant[./Soho]//phone_number` (i.e. `restaurant` has a child node `Soho` and a descendant `phone_number`).

Let $Q^T(a, b)$, represent an edge in the query tree Q^T. Note that each edge specifies the required topological relationship, $Q(a, b)$, between two nodes. For example, the edge $Q^T(\texttt{restaurant}, \texttt{Soho})$ indicates the query requirement $Q(\texttt{restaurant}, \texttt{Soho})$, that `restaurant` nodes must have a child `Soho`.

A result term, t_i, is a query term which must be returned to the user. In the above example, we are explicitly requesting all `phone_number` nodes, and thus the result term is `phone_number`. Note that it is possible for a given query tree to contain a number of different result terms, or the same result term repeated in different segments of the query tree. Result nodes, rn, are nodes which correspond to result terms. (eg. all relevant `phone_number` nodes).

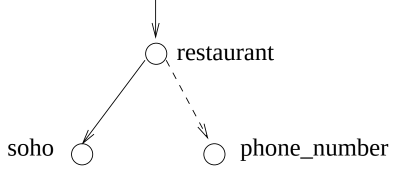

Figure 1: Query Tree for: `/restaurant[.//Soho]/phone_number`

For a given edge, $Q^T(a, b)$, we call a and b the head and tail of $Q^T(a, b)$ respectively. The head has the property that min(*dist*(a, t_i)) $<$ min(*dist*(b, t_j)), $\forall t_i, t_j$. If no such min(*dist*(a, t_i)) exists, we choose a s.t. it closer to the root of Q^T than b. Informally, the "head" of an edge is the end which is closer (in terms of number of edges) to the nearest result term. If both ends of the edge are equidistant, we choose the "head" as the end which is closer to the root of the query tree. Note that the head is defined for edges of the query tree, *not* the actual data.

2.3 Overall Approach

Efficient query relaxation poses a number of challenges.

For a single query tree edge, $Q^T(a, b)$, traditional query processing returns only nodes which *exactly* match the criteria. Query relaxation, however, requires us to return all r_a nodes, ranked by $Score(r_a, r_b)$, which indicates the degree to which r_a and r_b satisfy the criteria, where r_b is the node which *best* satisfies $Q(a, b)$.

A naive approach is to directly calculate all $Score(r_a, r_{b_i})$ resulting in $O(|a| \times |b|)$ operations to rank the results of a single query tree edge! Furthermore, calculating $Score$ depends on the path between two arbitrary nodes. Thus, for an arbitrary disk based graph with V vertexes, calculating even a single $Score$ could result in $|V^2|$ disk seeks!

To overcome these problems, we employ a structural index which contains an entry for each node in the graph. This entry contains the encoded subgraph containing all paths from the root to the given node. As many XML documents (or portions of these documents) are trees, many of these subgraphs will be a single path. The encodings themselves are very small (each encoding for a graph with 100,000 nodes fit into 2$\times$32 bit integers). As comparisons and calculations heavily utilize bitwise comparisons and optimizations, such operations are performed very quickly.

To rank the set of r_a nodes, we first obtain a large encoding, B, which represents the subgraph containing all r_b nodes. $Score(r_a, r_b)$ is then calculated by "overlaying" the encoding for each r_a node on B and observing the "best" fit between the given r_a node and the query criteria. $Score(r_a, r_b)$ calculation makes heavy use of bitwise operations and optimizations, and tends toward constant time in practice. The entire process tends toward O($|a| + |b|$) if B must be dynamically generated, or O($|a|$) if B has already been generated for a previous query tree edge.

Query tree edges are considered in a particular order (described in section 4.1), which means that the head in one edge will always be the tail (or result node) of a future edge. In order to calculate an overall score for the query, we employ a greedy algorithm to aggressive prune the search tree. The head in each iteration is "labeled" with the progressive minimum score, which is a combination of $Score(r_a, r_b)$ and the progressive score for all previously considered edges. This progressive score eventually becomes the final score used to rank the results. In this way the final score is progressively calculated with minimal additional overhead.

3 Similarity Considerations

The notion of "similarity" incorporates both topological deviation and the semantics such deviation implies. For example, consider the query `/restaurant/Soho`. Figure 2(a) shows the structure which exactly matches the query. If this structure is to be approximated, the new topology should be the closest structure in the *semantic* sense. Informally, this can be defined as "a different structure which means the same thing".

Figure 2(b)-(g) shows alternative structures which do not match the *structural* requirements of the query, but do match the *semantic* requirements. (These represent the 6 classes of structural relaxation of figure 2(a). Most current techniques consider only 2(b). [8] also considers 2(e)) As can be seen, if

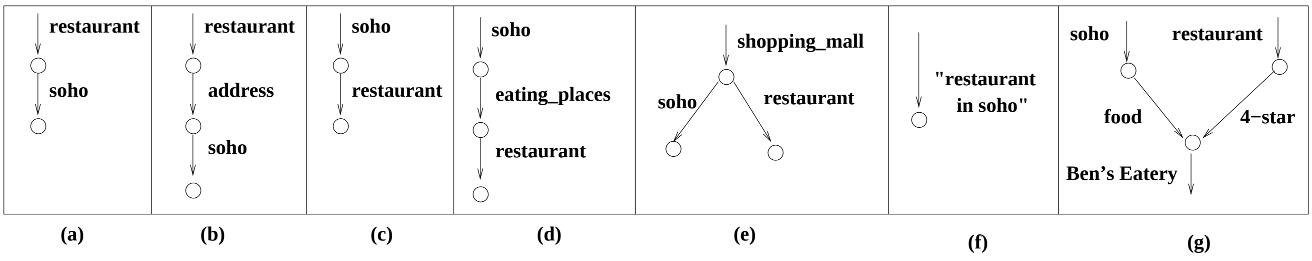

Figure 2: Semantically Similar Topologies

the structure implied by the query requires that a be a child of b, it is possible that *any* other topological relationship between a and b may also satisfy the *semantic* requirements of the query. Determining this computationally, however, is potentially very expensive.

3.1 Deviation Proximity

The metric used to determine similarity must (a) measure graph similarity, (b) be applicable to graphs (not just trees), and (c) be fast to calculate. Whilst a number of metrics, such as the tree edit (and related) distances, have been proposed, these frequently tend to violate points (b) and/or (c) [16].

As such, we propose a new metric, *proximity deviation*, as the basis of our structural similarity metric. Informally, this captures how ”far” one structure deviates from a desired structure. Deviation proximity is defined as follows.

> Let r_a and r_b be 2 specific nodes with values a and b respectively. For a given edge, $Q^T(a, b)$, of the query tree, the deviation proximity of node r_b to r_a, is the proximity of the *actual* position of r_b to the *nearest* position, r_b', which satisfies the topological relationship specified by $Q^T(a, b)$.

Deviation proximity can be illustrated considering the differing topologies in figure 2. Assume that Q(`restaurant, Soho`) is specified by `/restaurant/Soho`. The only topology which exactly satisfies this criteria is figure 2(a). Deviation proximity is calculated with reference to this topology.

Deviation proximity is calculated by comparing the *actual* position of the node `Soho`, with the position shown in figure 2(a). Conceptually, we assume that each topology has a node, `Soho`$'$, which is the child of the node `restaurant`. For figure 2(b), we assume that `Soho`$'$ lies on the path to `Soho`. Deviation proximity is thus the minimum number of edges between `Soho` and (the virtual) `Soho`$'$. The deviation proximities for figures 2(a) through 2(g) are therefore 0, 1, 2, 3, 3, 1 and 3 respectively.

Note that if Q(`restaurant, Soho`) only requires that `Soho` be a *descendant* of `restaurant`, both figures 2(a) and 2(b) exactly satisfy the criteria. For the other topologies, choosing `Soho`$'$ to be the same as in the previous example yields the minimum number of edges between `Soho` and `Soho`$'$. This results in deviation proximities of 0, 0, 2, 3, 3, 1 and 3 respectively.

An important property of deviation proximity (DP) is that r_a and r_b exactly satisfy $Q(a, b)$ iff $DP(r_a, r_b) = 0$.

This is vital if we wish an answer set to include results which both exactly match the criteria, as well as those which are approximated.

Deviation proximity thus provides a metric which captures the ”amount” by which an *actual* topology varies from a desired topology as specified in the query. The semantic relevance of the deviation score is further refined by application of various scoring functions, discussed below.

3.2 Scoring Transformations

The final similarity score is obtained by applying the deviation proximity to a range of scoring transformations, which serve to refine the semantic relevance of the final score. The precise number and makeup of the transformations used are dependent on the exact database design and content. For example, a corporate database containing documents adhering to a limited number of known schema will have different transformations from one containing ”all web data”.

The precise transformations can be complex, depending upon a number of factors. Space limitations preclude us from discussing these transformations in detail. We briefly discuss some of the relevant factors.

• **Proximities which Span Documents**. Depending on the manner in which the database stores multiple documents, it is possible that $DP(r_a, r_b)$ may be low, even though r_a and r_b exist within different documents. This can be particularly misleading (in the semantic sense) if either r_a or r_b are near the root element. As such, it may be appropriate to penalize proximities where r_a or r_b are in different documents (or documents with differing schema etc). We chose to penalize proximities where r_a or r_b were in different documents, and the path between r_a and r_b which resulted in $DP(r_a, r_b)$ passed through the root of both documents. (This definition did not penalize proximities which spanned documents due to links).

• **Depth Based Scaling** XML documents inherently contain hierarchical data, where the hierarchy is defined by the author and often semantically based. In such a situation, the root element may define the document content, with subsequent levels progressively refining the semantics of the document. Thus, a topological approximation which occurs at a relatively shallow level (representing a more ”general” element) may indicate a more ”general” or ”broad” approximation (in the semantic sense). In this case, it may be appropriate to scale deviation proximity based on the depth at which the approximation occurs. Semantically, the ”deeper” the deviation occurs, the smaller the deviation in the semantic sense.

• **Topology Mapping Relevance** It is important to consider the semantic similarity between the actual and ”virtual” topologies used to calculate the deviation proximity. For example, it is likely that the topology in figure 2(a) is semantically closer to figure 2(b) than to figure 2(e). For generalized data (such as ”all web data”, it is possible to construct a set of transformations which describe the likely semantic similarity between various topologies. For the sake of generality, this is the approach we used in our experiments. If the database contains a more limited set of known schema, a more specific set of such transformations can be constructed.

4 Query Relaxation Processing

4.1 Converging Order

The order in which we consider edges from the query tree plays an important part in the overall query processing. Informally, we wish to consider paths through the query tree which ”converge” on a result term (it is not important which one). This allows us to progressively calculate the overall deviation score. If the head of one edge becomes the tail of the next, we can store the progressive scores at the head/tail nodes, and thus minimize the amount of processing required for calculating and updating the progressive scores. Furthermore, if the last edge we consider in the query tree contains a result term, we can then directly insert this in the final result set, without needing any additional processing to retrieve the result nodes from the graph or any interim data structure. This ensures that the overall algorithm is O(n), where a given query tree has n edges.

We refer to this order of considering edges from the query tree as *converging order*. Figure 3, shows a query tree, with one possible converging order (Note that for any given query tree, converging order is not necessarily unique). Nodes labeled *r* indicate result terms, and edge labels correspond to the order in which edges are considered.

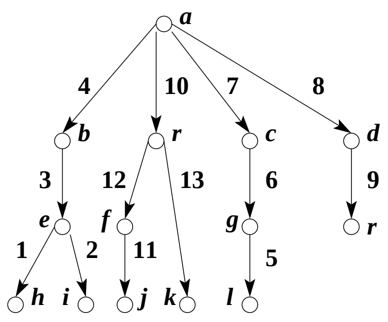

Figure 3: Sample Query Tree

The algorithm which formally specifies converging order is shown in figure 4. The salient features about converging order is that for subtrees (rooted at children of the root) which do not contain a result term, the tree is considered ”bottom up” (steps $[2]$ through $[12]$). Branches which do contain a result term are considered ”top down” until the result term is reached (steps $[13]$ through $[19]$). (During this ”top down” traversal, any branches which do not contain a result term are considered ”bottom up” (steps $[13]$ through $[15]$), and then the ”top down” traversal continues (steps $[16]$ through $[19]$)). Any subtrees rooted at a result term are considered ”bottom up” (steps $[20]$ through $[23]$).

Steps $[5]$ and $[10]$ enable us to return useful values if a specific data value is not found. These steps ignore a particular leaf of the query tree, if no exact match exists in the database. Note that any CDATA value (such as `"Bob"` in `name = "Bob"`) is guaranteed to correspond to such a leaf node. As such, we enable the system to return useful answers in cases where specific data values do not exist.

We can now start to see how converging order assists in query relaxation. Consider the query tree and ordering shown in figure 3. The first edge we consider in the query tree is $Q^T(e,h)$, which specifies the query requirement that nodes matching h must be a child of nodes matching e.

The first pass of the algorithm in figure 6 finds the minimum score of all nodes which match e to all nodes which match h. This score is stored with the encoding for each element of e.

Note that the actual e and h nodes may have any topological relationship. It is for this reason that traditional tree matching approaches are unsuited to query relaxation.

The second pass of the algorithm in figure 6 (which considers $Q^T(e,i)$) calculates the minimum score of all nodes matching i to all nodes matching e. As mentioned in section 4.2, this requires us to access the encoding for each element matching e. At the same time, we retrieve the previous score (stored with the element) and combine it with the new score, for the progressive score for that element.

Converging order means that e will not be the head of any more query tree branches. When the element is incorporated into a large encoding as the tail of a subsequent edge ($Q^T(b,e)$), this progressive score is included in the large encoding. As such, we always obtain the most recent, relevant, progressive score with no additional lookups.

4.2 Similarity Score Calculation

Similarity score calculation is central to query relaxation processing. As such, the efficiency of this algorithm is vital to the overall performance of the system. As discussed in section 3.2, the transformation function used to calculate the scores depends on a number of graph-based metrics. Armed with these metrics, calculating the score is typically linear. The most important (and potentially most expensive) job of the scoring algorithm is therefore to obtain the relevant metrics as efficiently as possible.

```
[1]  If the root, $rt$, is not a result term
[2]  Let $start = orig\_start = rt$
[3]      For each child, $c$, of $start$, where the subtree
         rooted at $c$, $s(c)$, does not contain a result term
[4]          Let $start =$ any leaf node in $s(c)$
[5]          If no node in the database matches $start$, then
             $start =$ parent$start$
[6]          Whilst there are unconsidered edges in $s(c)$
[7]              Consider edge $(start,$ parent$(start))$
[8]              If parent($start$) has unconsidered children
[9]                  Let $start =$ any unconsidered leaf node
                     which is a descendant of parent($start$)
[10]                 If no node in the database matches
                     $start$, then $start =$ parent($start$)
[11]             Else let $start =$ parent($start$)
[12]         Consider the edge $(c, orig\_start)$
[13]     Let $start = orig\_start = rt$
[14]     Whilst we still have paths to explore
[15]         Symmetrical steps [3] through [9]
[16]         For each child of $start$, $c'$, where $c' \neq c$
[17]             Consider the edge $(start, c')$
[18]             If $c'$ is a result term, stop exploring this path
[19]             Let $start = c'$
[20] For each subtree rooted at a result term, $rt'$
[21] Let $start = orig\_start = rt'$
[22]     For each child, $c$, of $rt'$
[23]     Symmetrical steps [3] through [11]
```

Figure 4: Converging Order Algorithm

The most basic metrics needed by the transformation function are the minimum distances between the given *head* element (r_a) and the *nearest* descendant, ancestor and sibling *tail* element. It is important to obtain all 3 of these metrics to ensure that the correct deviation proximity and score can be calculated. Additionally, it is important to obtain any progressive scores for the each of these nearest *tail* nodes, so this can be incorporated into the new score.

As the required distances involve both the distance and topological relationship between *any* 2 nodes, calculating this distance might suggest the need to compare each head node with *all* tail nodes. This approach is O($|a| \times |b|$), which is obviously untenable.

Furthermore, even if we somehow know the location of the relevant tail node, the deviation score is based on the topological relationship between two nodes. This might suggest the need to fully locate *both* nodes in the graph. This effectively corresponds to additional redundant graph traversals, even if the graph is encoded and traversals are accomplished using bitwise operations.

To avoid these shortfalls, we utilize a two phase approach. We first obtain an encoding which represents the subgraph containing all *tail* elements. Next, we ”overlay” the encoding for each *head* element on this large encoding, until the two encodings diverge. This ”overlaying” process makes heavy use of bitwise operations and optimizations.

4.2.1 Encoding Schemes

Score calculation utilizes the graph encoding schemes from [5], designed to allow graph algorithms to be efficiently performed on a compact representation of the graph.

Nodes of interest (called *terminal nodes*) are identified as being the terminus of one or more paths.

A terminal node is thus represented by the encoding of the subgraph containing all paths from the root to the node. Edges are identified by a number called an *edge identifier*.

The most basic encoding, *path encoding*, represents a graph with a single path and terminal node. For graphs with multiple paths, those with a single terminal node use *multi-path encoding*, whilst those with multiple terminal nodes use *multi-element encoding*.

Both multi-path and multi-element encoding consist of path encoding segments and *annotated nodes* (the collective term for terminal nodes and nodes with 2 or more incoming or 2 or more outgoing edges).

Annotated nodes (ANs) are ”annotated” with the minimum distance to the *nearest* terminal node, which is stored in the encoding. This alleviates the need to ”visit” each *tail* to obtain the minimum distance. Encodings are implemented in the compressed array data structure [5].

The encoding schemes are reproduced in appendix A.

```
Algorithm: Similarity Score Calculation
Input:     Query Tree Edge, $Q^T(a,b)$, which requires topological relationship $ReqdReln$ between $a$ and $b$
Output:    Set of $< r_a, Score >$ tuples, where $Score$ is the minimum similarity score for $r_a$

Note:      $AN_x$ indicates the annotated node where path encoding segment $pe_x$ terminates
           $EdgeId_{pe_a}[idx]$ indicates the edge identifier in path encoding segment $pe_a$, starting at bit index $idx$

[1]  MinDist.$desc$ = MinDist.$anc$ = MinDist.$sibling$ = MAXINT
[2]  Obtain multi-element encoding, $e_B$ representing all nodes, $r_b$, which match $b$
[3]  For each node, $r_a$, which matches $a$
[4]      Obtain encoding, $e_a$, which represents $r_a$
[5]      Loop
[6]      Compare path encoding segments, $pe_a$ and $pe_B$ of $e_a$ and $e_B$ respectively
[7]          If segments diverge at bit index $DivIdx$, $DivIdx <$min(len($pe_a$), len($pe_B$))
[8]              new_prox = min_dist($AN_B$) + BitCount($pe_B, DivIdx$, len($pe_B$)) + min_dist($AN_a$) +
                                BitCount($pe_a, DivIdx$, len($pe_a$))
[9]                  if MinDist.$sibling >$ new_prox
[10]                     MinDist.$sibling$ = new_prox
[11]                     $ProgScore.sibling(r_a) = ProgScore(AN_{e_B})$
[12]             If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
[13]         else if len($pe_a$) $<$ len($pe_B$) // we have ”reached” $AN_a$
[14]             Symmetrical steps to [8] through [11]
[15]             (If $AN_a$ represents a terminal node, update MinDist.$desc$ and $ProgScore.desc$)
[16]             If $\exists pe'_a$, s.t. $pe'_a$ originates from $AN_a$ and $EdgeId_{pe'_a}[0] = EdgeId_{pe_B}$[len($pe_a$)]
[17]                 Let $pe_a = pe'_a$. Remove first len($pe_a$) bits from $pe_B$. Continue loop.
[18]             else if no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
[19]         else if len($pe_B$) $>$ len($e_a$) // we have ”reached” $AN_B$
[20]             Symmetrical steps to [14] through [18]
[21]             (If $AN_B$ represents a terminal node, update MinDist.$anc$ and $ProgScore.anc$)
[22]         else // $pe_a$ and $pe_B$ are identical - we have ”reached” both $AN_a$ and $AN_B$
[23]             Symmetrical steps to [8] through [11]
[24]             (Update $ProgScore$/MinDist.$desc/anc/sibling$ according to conditions from steps [8], [15] and [21])
[25]             For each $pe'_a, pe'_B$, s.t. $pe'_a$ and $pe'_B a$ originate from $AN_a$ and $AN_B$ respectively,
                 $EdgeId_{pe'_a}[0] = EdgeId_{pe'_B}[0]$
[26]                 Add $pe_a, pe_B$, state to queue
[27]             If no paths in queue, exit loop. Otherwise, restore state from head of queue and continue loop.
[28]     $Score = T(ReqdReln$, MinDist.$desc/anc/sibling) +$ corresponding $ProgScore$
[29]     Insert $< r_a, Score >$ into result set.
```

Figure 5: Similarity Score Algorithm

4.2.2 Algorithm Overview

The similarity score algorithm is described in figure 5. The algorithm obtains the multi-element encoding representing all *tail* elements (step $[2]$). The encoding representing each *head* element is then obtained (step $[4]$). Conceptually, this encoding is ”overlaid” on the multi-element encoding until they diverge (steps $[6]$ through $[27]$). During this ”overlaying”, the necessary metrics are progressively obtained (steps $[8]$ through $[11]$, $[15]$, $[21]$, $[24]$). Finally, the element score is calculated (step $[28]$). Note that this score arises from combining the results of the specific specific transformation function described in section 3.2 (T) with the progressive score from the relevant *tail* node.

Looking at this ”overlaying” process in detail, we see

1. Corresponding path encoding segments are compared (step $[6]$).
2. If the segments differ before the end of both segments, the paths have diverged (step $[7]$). In this case, the minimum distance is calculated (steps $[8]$ through $[11]$) and any queued paths are considered (step $[12]$).
3. If the beginning of the longer segment exactly matches the shorter segment (steps $[13]$ and $[19]$), we are at an AN. After updating the relevant scores (steps $[15]$ and $[21]$), we see if any outgoing segment from the AN matches the remainder of the longer segment (step $[16]$). If it does, we continue comparing these segments (step $[17]$), otherwise, consider queued paths.
4. If the path encoding segments are identical, we have reached an AN in both encodings. We now have the possibility that more than one path will be found in common. In addition to updating the relevant score, each such path is queued (steps $[25]$ and $[26]$).

To maximize performance, care has been taken to reduce the cost of all steps. Calculations utilize bitwise operations instead of recursive algorithms, wherever possible.

For example, comparing 2 path encoding segments (step $[6]$) is achieved using a single series of bitwise comparisons. In practice, this allows us to compare paths of any length in constant time [2]. Similarly, determining whether paths diverge (step $[7]$), or whether 1 or more ANs are reached (steps $[13]$, $[19]$ and $[22]$) is also accomplished using a series of constant time bitwise operations.

The emphasis on bitwise operations can be clearly seen for score calculation (step $[8]$). Conceptually, this step counts the number of edges between the point of divergence and each annotated node (using BitCount()), and adds this to the minimum distance stored at the AN (min_dist(AN)). The function BitCount(ca, fr, to) is a non-iterative bit counting algorithm which counts the number of bits in the portion of the compressed array ca where each set bit indicates an edge, between bit index fr and to. This enables us to calculate the necessary distance in constant time, instead of iterating through each edge. Details of the specific bitwise operations and functions can be found in [5].

We can now see the significance of storing the minimum distance (and the progressive score) at each AN. Storing these allows us to calculate and retrieve the relevant amount at no additional cost, eliminating the need to ”visit” each node to obtain the relevant metric.

4.2.3 Algorithm Cost

Generating the large encoding in step $[2]$ is an O($|r_b|$) process. Depending upon the given query execution plan, it is sometimes possible to retrieve this encoding from the cache and so avoid this cost.

[2]Whilst the comparison cost depends on the size of the encoding, in our experiments, all path encoding segments fit into $2\times$ 32 bit integers and so could be compared in constant time

The main cost comes from the distance calculation phase (steps [5] through [27]). As discussed, the cost of each step is constant time in practice. The overall algorithm cost is determined by the main loop (step [5]).

The worst case cost for the loop is O($E(e_a)$), where the encoding e_a represents $E(e_a)$ edges. In practice, however, the cost is substantially lower.

As previously discussed, the algorithm only continues to compare paths (or their encodings) as long as they converge. Furthermore, path segments are compared in a single loop iteration. This means the loop only repeats if the paths converge and an AN has been reached in either path.

As a result, distances are frequently calculated in very few (or even a single) iteration. Our experiments show that this phase tends toward constant time in practice.

As distance is calculated for all *head* elements, this process tends toward O($|r_a|$) in practice.

Thus, in practice score calculation is an O($|r_a| + |r_b|$) process if the encoding in step [2] must be generated, or O($|r_a|$) if it is retrieved from the cache.

4.3 Query Relaxation Processing

It is now possible to consider the overall query relaxation process. The query relaxation process is shown in figure 6. This algorithm considers query tree edges in converging order (step [1]), calculates the score for each edge (step [2]), and ensures future progressive scores are correctly calculated (steps [3] through [12]).

```
Algorithm: Query Relaxation Processing
Input:     Query Tree, $Q^T$
Output:    Set of ranked result nodes, $< r, ProgScore >$, where $ProgScore$ is the overall score representing
           the appropriateness of $r$,

[1]   For each query tree edge, $Q^T(a, b)$, considered in converging order
[2]       Obtain $< r_a, Score >$ for all $r_a$ which match $a$
[3]       if $a$ will be the head of a future query tree edge
[4]           store $< r_a, Score >$
[5]       else
[6]           $ProgScore(r_a) = Combination(Score(r_a)_1, \ldots Score(r_a)_n)$
[7]           if $a$ will be the tail of a future query tree edge
[8]               incorporate $< r_a, ProgScore(r_a) >$ into multi-element encoding.
[9]           else // $a$ is a result node
[10]              store $< r_a, ProgScore(r_a) >$
[11]      if this is last query tree edge
[12]          $ProgScore(r_a) = Combination(ProgScore_1 \ldots ProgScore_n)$, for stored result node $ProgScore$s
[13]          Insert $< r_a, ProgScore(r_a) >$ into results
```

Figure 6: Query Relaxation Processing

In addition to specifying various topological criteria, the query tree also specifies boolean relationships between these criteria. For this reason, the final progressive score of a query tree node which is the head of multiple edges, can only be calculated after all such edges edges have been considered. For example, in figure 3, the edges $Q^T(e, h)$ and $Q^T(e, i)$ must both be considered before the final progressive score for e can be determined.

By examining the query tree, the query relaxation processor knows how many scores are required for each query tree node, and allocates the appropriate amount of space in the cache when the records are retrieved from the database. This typically increases the storage and retrieval efficiency for the various scores (steps [4] and [6]).

If a query tree node will no longer be the head (step [5]), converging order guarantees that it will either be the tail of a future edge (step [7]), or it is a result node (step [9]).

The specific operations used to combine any two progressive scores ($Combination()$ in step [6]) depend on the boolean relationship between the relevant edges in the query tree. For example, if two branches in the query tree are related by a boolean AND, progressive scores are added, whereas if they are related by a boolean OR, minimum progressive score is used. The $Combination$ function in step [12] additionally caters for the possibility of progressive scores for multiple result terms. In this case, the $Combination$ function from step [6] is applied to the progressive scores from each result term and summed into a final score.

4.3.1 Algorithm Cost

As discussed in section 4.2.3, the cost of calculating the score is O($|a| + |b|$) if the multi-element encoding must be generated, or O($|a|$) if it can be retrieved from the cache. By progressively constructing the multi-element encoding for a future query tree edge in step [8], we allow this edge to retrieve the encoding from the cache and thus reduce the score calculation cost to O($|a|$).

Thus, the total cost of performing the query relaxation processing is O($|a| + 2|b| + \ldots + |m|$) $\approx$ O($n \times (|a|_{avg} + |b|_{avg})$), for a query tree with m nodes and n edges and an average of $|a|_{avg}$ and $|b|_{avg}$ matches for the head and tail of each query tree edge.

5 Further Applications

A substantial barrier to querying semi-structured data is the use of different terms to represent the same ”thing”. The query relaxation method presented provides a framework for efficient context and semantically sensitive term substitution. For example, a query dealing with ”cars” may return results for ”automobiles” form one web source and ”motors” from another. Alternatively, a query searching for ”zip code” may result in data containing the ”number_id” for the relevant ”suburb”.

Whilst space limitations preclude us from giving a detailed discussion of these issues, we present a broad overview of our approach.

5.1 Overview

By it’s very nature, semistructured data specifies semantic relationships between data elements. Whilst the semantic relationships may or may not be explicit, recognizing that they exist increases the quality of the term substitution.

The first thing we recognize is that the query itself contains implicit semantic and contextual information.

Query tree edges are likely to represent a ”has-a” relationship. This is true, for example, for `restaurant/phone_number`, which implies that `restaurant` ”has a” `phone_number`. Whilst parent-child nodes do not always encapsulate a ”has-a” relationship (eg. `restaurant/Soho`), this is the ”safest” semantic relationship to assume in the absence of other information. If the user is engaged in an interactive query session, the semantic relationships between the various query terms can be precisely established through explicit questioning.

5.2 Context Sensitive Term Substitution

For data without a specified ontology, the query relaxation processing can still be adapted to provide contextually appropriate term substitution. For example, the system must be able to determine that the query term ”car” refers to automobiles and not to train or cable car carriages.

The wordnet [14] lexical database contains English words, arranged in a conceptual hierarchy, with various words joined by various semantic relationships. Conceptually, the data forms an arbitrary graph, with different ”types” of edges.

Armed with the semantic relationship between the various query terms, we can therefore apply the method described in this paper to determining the most appropriate term. Assuming the wordnet database is indexed as described in section 2, we can approach term substitution using the approach outlined in figure 7.

```
Algorithm: Context Sensitive Term Substitution
Input:     – Query term $a$, where $a$ does not exist
             in data source
           – All Query Tree Edges,
             $Q^T(a, b_1) \dots Q^T(a, b_n)$ and
             $Q^T(c_1, a) \dots Q^T(c_n, a)$, which
             include $a$
Output:    Set of ranked result nodes, $< a', score >$,
           where $score$ indicates the contextual
           appropriateness of $a'$ as a substitution for $a$

[1] Retrieve all terms $a, b_1 \dots b_n, c_1 \dots c_n$
    from indexed wordnet and generate
    (modified) multi-element encoding
[2] For each non-leaf term, $d_i$ within a data source
[3]     For each occurrence, $d_{i_j}$ of $d_i$ within wordnet
[4]         Obtain encoding $e_d$ for the occurrence
[5]         calculate $< d_{i_j}, score_j >$ using (modified)
            algorithm from figure 5
[6]     Insert $< d_i, score >$ into results,
        s.t. $score = \min(score_j)$
```

Figure 7: Contextually Appropriate Term Substitution

The solution presented in this paper efficiently calculates a score based on the difference between an actual and desired topological relationship.

In the current context, graph edges represent semantic relationships between terms. This allows us to apply the same principle to contextually sensitive term substitution.

The approach outlined in figure 7 builds a (contextual) graph representing the desired semantic relationships contained in the query (step [1]). The process considers non-leaf data terms, which typically correspond to schema terms. This drastically reduces the number of terms to be considered. Terms in the data source are then ”overlaid” on this graph (steps [2] through [5]), ranked by the ”degree” to which various data source terms correspond to the relevant query term.

Some minor modifications are needed for this process to work. The multi-element encoding must be modified to include the edge type, which corresponds to different types of term relationships. Similarly, the score calculation algorithm in figure 4.2 must be modified to consider the edge type. This algorithm must be further modified to allow calculation based on the relationship of the desired node (a) with multiple candidate nodes ($b_1 \dots b_n, c_1 \dots c_n$), potentially each with differing edge types.

5.3 Semantically Sensitive Term Substitution

The semantic web project [18] seeks to enable ”machine-understandable” web data, largely through ontologies which specify the semantics of the data. The ontologies themselves are arbitrary graphs, with edges denoting the various relationships between terms.

The approach presented in figure 7 can be modified to effectively provide automatic term substitution, taking into accounts the term relationships specified in the published ontology. The modified $score$ function in step [5] is further modified to consider the relationship between various data terms as specified in the ontology. This effectively allows term substitution which is contextually and semantically appropriate for a specific ontology.

5.4 Further Work

Our work in this area is still in very early stages. A number of issues remain to be solved before term substitution can be seamlessly and efficiently incorporated into the overall query relaxation mechanism. such issues include:

Combining structural and term approximation scores. Both term substitution and structural approximation result in individual relevance scores. Further work needs to be done to investigate efficient mechanisms for generating and combining these scores into a single metric. Whilst efficiently generating individual scores is described in this paper, the process as a whole must be further investigated, particularly as the order of generation (ie. are terms substituted before structure is approximated?) will effect the value of both scores.

Term substitution efficiency. Whilst the efficiency of the score calculation algorithm remains as presented in this paper, the process as a whole potentially requires many more executions, depending on the number of data sources or schema the process must be run for.

6 Results

Our query relaxation engine was implemented in over 17,000 lines of C code. Tests were run on a Pentium III 800 MHz processor with 256MB RAM running Linux Red Hat 7.2. A native XML database [19] was used to store the data. The encodings representing the data were indexed and retrieved using the path index detailed in [4].

Choosing the metric of database size is somewhat misleading for these results, as the process is dependent on the number of nodes, not the size of the data contained at each node. A document with 100 nodes can be as small as 1 KB or as large as 1MB. To give a fair indication, therefore, we consider only the number of nodes in the graph, without any reference to the size of the data stored at a single node.

Data was generated to ensure that semantically similar content was adhered to in a large number of disparate schema. The data was scaled up or down as appropriate to test the scalability of our approach. It was decided to generate data rather than use a publicly accessible data source due to the above mentioned requirements of the data set.

6.1 Qualitative Results

Figure 8 shows a summary of the types of answers obtained for various queries. Queries 1, 2 and 3 were formulated on the basis of ”reasonable” or ”logical” assumptions regarding the ”likely” structure of the data. The queries assumed no prior knowledge of the structure of the data.

Query 4 was formulated with prior knowledge of the underlying structure of the data. This is representative of the situation where the structure of the data is known, but the values may not be. Note

Query Relaxation Results	
Desired Info	Find all restaurants in Soho
Actual Query	//restaurant/Soho
Results	Restaurants with a child named ”Soho”, followed by restaurants with an address in Soho, followed by restaurants contained in a ”Soho” tourist guide document.
Desired Info	Find all phone numbers of restaurants in Soho
Actual Query	//restaurant[./Soho]//phone_number
Results	Phone numbers with an ancestor ”Restaurant” who had a child named ”Soho”, followed by the phone numbers of restaurants with an address in Soho, followed by a 1800 number to an ”eating out” guide which included restaurants in Soho.
Desired Info	Find all hotels in Sydney with rooms less that $100/night
Actual Query	//hotel[.//Sydney][.//price < 100]
Results	Hotels with a descendant of ”Sydney” and a descendant of ”Price” whose value was less than $100, followed by hotels in a ”Sydney” tourism guide with price less than $100 followed by hotels in Sydney in the ”budget” category (where budget was defined as <$100 in that document).
Desired Info	Find the telephone number of the sales manager, ”Bob Smith” of company ”Fishy Enterprises” (NB: Suppose there is no such person at this company)
Actual Query	//sales_manager[./company/name=”Fishy Enterprises” and ./name=”Bob Smith”]/phone_number
Results	Telephone numbers for the sales managers at ”Fishy Enterprises”, followed by the phone number of the sales department, followed by the phone number of the secretaries to the sales managers.

Figure 8: Query Relaxation Results

that even though the desired information did not exist in the database, the results returned were useful, as they offered a likely means of obtaining the required information, utilizing a mechanism external to the database itself (phoning the appropriate department, for example). Similarly, if the user decided it was not overly important to have the phone number of Bob Smith, a list of alternative contacts was provided.

For all queries, data which exactly matched the criteria (if it existed) was returned first. Subsequent results (with low rank) all contained information which was semantically appropriate to the information desired by the user.

Other mechanisms which consider the structure implied in the query [8, 17] are unable to return results where the data are joined using links, and are unable to process certain topological transformations.

6.2 Performance Results

6.2.1 Relative Speed

As much of the existing work on query relaxation for semistructured data fails to discuss performance in detail, it was difficult to compare our mechanism directly with existing approaches.

To demonstrate the efficiency of our solution, we implemented a naive modification of the query relaxation mechanism. For fairness, the naive implementation utilized our encoding mechanism, to ensure that the performance differential was not due to a high number of disk seeks. The only difference between the naive implementation and the optimized mechanism presented in this paper is in the implementation of the similarity score calculation. This demonstrates the centrality of this algorithm to performance.

Figure 9 shows the query relaxation processing time for a 2 term query (ie. with a single query tree edge), as the data size increases. The x-axis shows the selectivity per query tree edge (ie. $|a| + |b|$). Roughly half the selected nodes correspond to the query tree edge head ($|a|$) and tail ($|b|$) respectively. The total number of nodes in the database were scaled linearly with the selectivity.

As can be seen, the optimized mechanism is substantially more efficient than the naive implementation. When the head and tail each match approximately 20,000 nodes, our mechanism is 3 orders of magnitude faster. Performance times for the naive implementation increase quadratically with data size, whereas our mechanism shows a near linear increase.

For the naive implementation, the query relaxation processing costs dominated the total processing time (selectivity of approx 400) a full order of magnitude earlier than for our mechanism (selectivity of approx 4,000).

It is significant to note that when the head and tail each match approximately 20,000 nodes, the query relaxation processing required only 0.17 second. This is especially significant when we consider that the process is required to select the most appropriate data graph edge from at least 4×10^8 such edges.

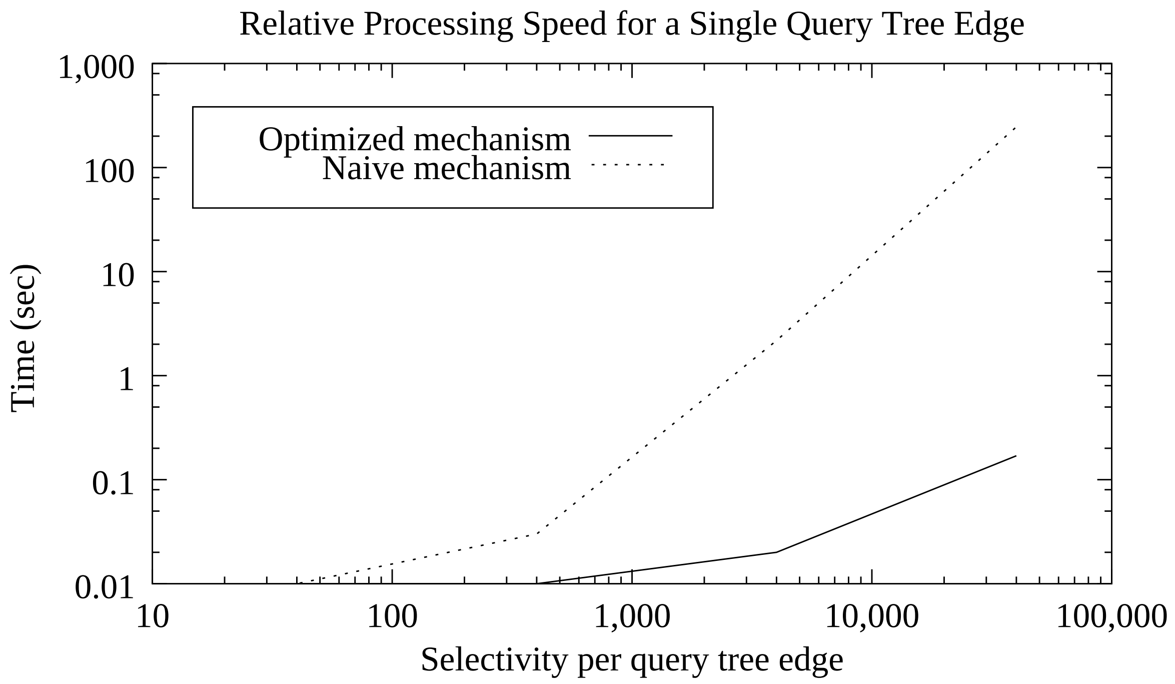

Figure 9: Processing Speed for a 2 Term Query

6.2.2 Number of Query Terms and Performance

Figure 10 shows the effects of the number of query tree edges on performance, for a given data size. The tests were run over data sets of differing sizes, with the number of nodes matching a query tree edge scaled proportionately to the data size. As in section 6.2.1, roughly half the selected nodes correspond to the head and tail of each query tree edge.

As can be seen, query processing time increases linearly with the number of query tree edges (or query terms). This is significant, as each additional query tree edge potentially results in an exponential increase of overall query relaxation processing time.

The linear relationship further illustrates the benefits of our encoding mechanisms, which enable multiple edges to be compared in a single bitwise operation. For this reason, the increase depends only upon the number of nodes matching the head or tail, not the total number of nodes in the edges between any head-tail pair. This fact alone yields significant benefits as the data size (and thus the potential length of the path between any two nodes) increases.

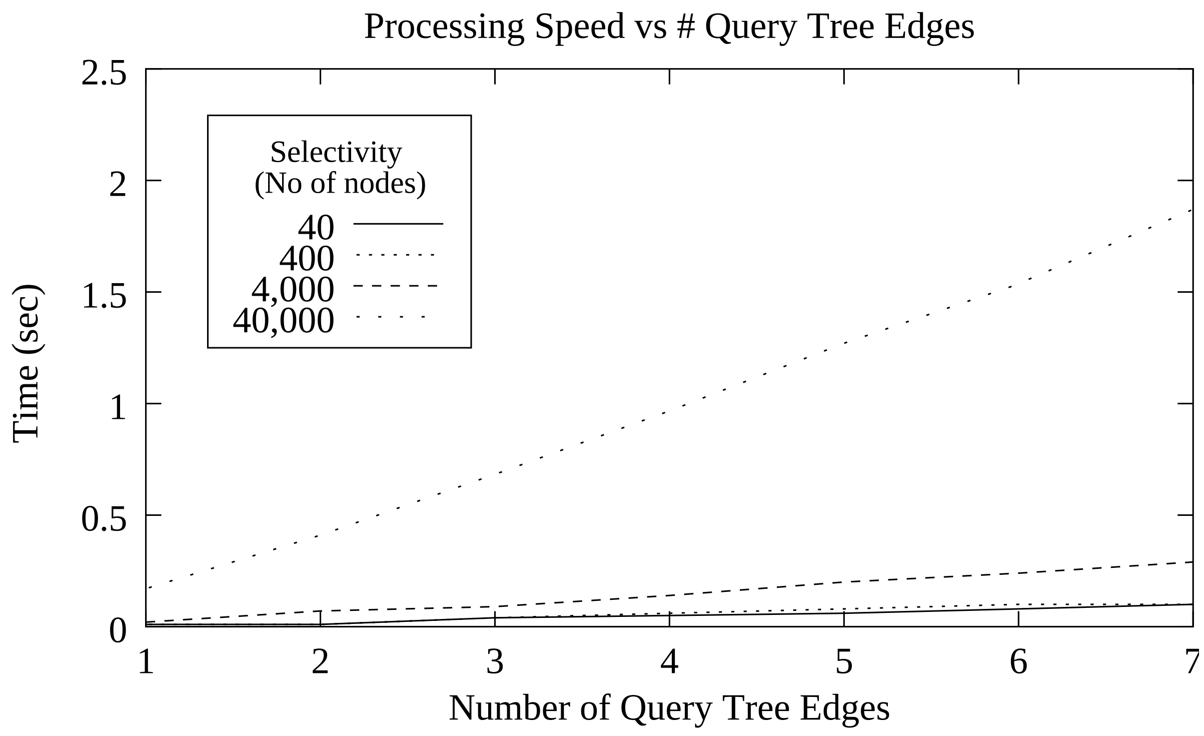

Figure 10: Processing Speed vs # Query Tree Edges

6.2.3 Selectivity and Performance

Figure 11 shows the effects of increasing the query term selectivity on performance, for a given number of query terms. Both the selectivity and the data size were scaled proportionately. Roughly half the number of selected nodes for each query tree edge correspond to the head and tail. The results are shown for multiple queries with differing number of query terms.

Where each query tree edge matched less than around 4,000 nodes, the total processing time was dominated by index access and other system overhead. For greater selectivity, the similarity score calculation was the main contributor to the processing time.

Importantly, once the query relaxation cost becomes the dominant factor, the query cost increases linearly for a given query.

7 Conclusions

In this paper we have presented a method for implementing a fast, efficient query relaxation mechanism for semi structured data. Our mechanism returns results where the structure of the data approximates the structure specified in the query. Our notion of approximation incorporates both structural and semantic similarity. Our mechanism also returns helpful results in instances where specific data

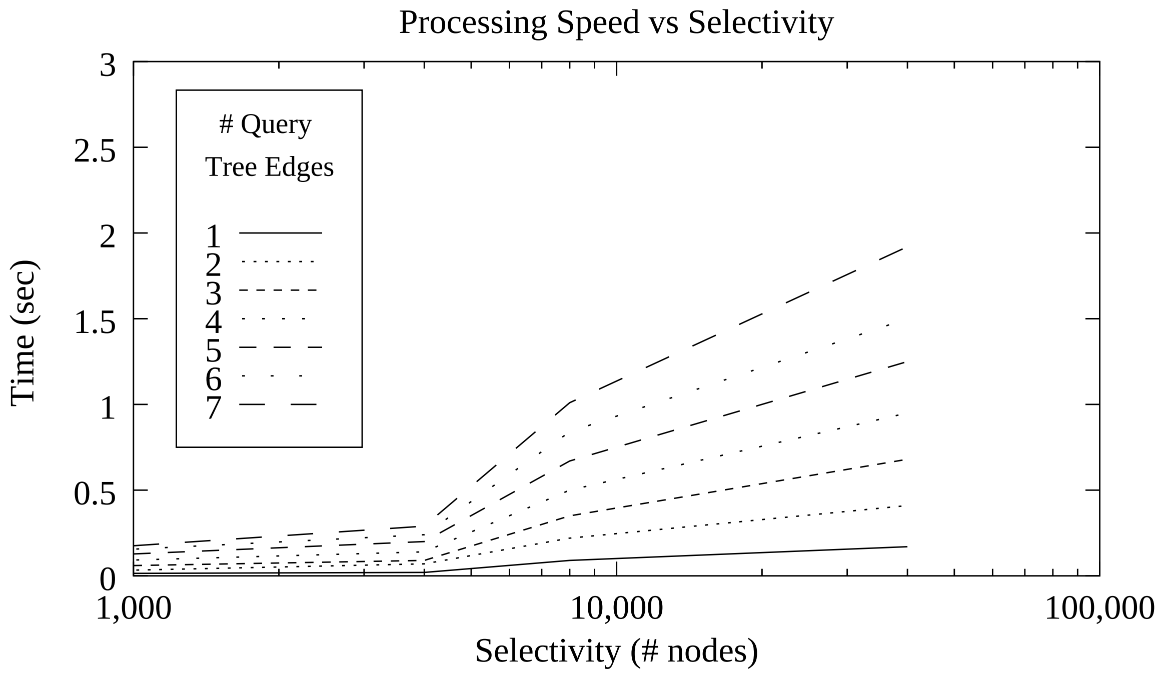

Figure 11: Processing Speed vs Selectivity

values specified in the query cannot be found. We have described a framework for calculating similarity scores, which incorporate both structural and semantic similarity, in an efficient manner. Our mechanism considers an arbitrary graph and all possible structural transformations. Experiments show that our query relaxation is very fast. Processing time scales linearly with the both selectivity and number of query terms. This is especially impressive when one considers that each of these factors exponentially increases the search space.

References

[1] S. Abiteboul. Querying semi-structured data. In *ICDT*, 1997.

[2] S. Acharya, P. Gibbons, V. Poosala and S. Ramaswamy. Join synopses for approximate query answering In *SIGMOD*, 1999

[3] S. Amer-Yahia and S. Cho and D. Srivastava. Tree Pattern Relaxation. In *EDBT*, 2002

[4] M. Barg and R. Wong A Fast and Versatile Path Index for Querying Semistructured Data. In *DASFAA*, 2003.

[5] M. Barg and R. Wong Structural Proximity Searching for Large Collections of Semi-Structured Data In *ACM CIKM*, 2001.

[6] N. Bruno, S. Chaudhuri, L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. In *ACM Transactions on Database Systems*, 27(2):153-187, 2002

[7] K. Chakrabarti, M.Garofalakis, R. Rastogi and Kyuseok Shim. Approximate query processing using wavelets. In *VLDB Journal*, 10, 199-223, 2001

[8] P. Ciaccia and W. Penzo. Adding Flexibility to Structure Similarity Queries on XML Data In *FQAS*, 2002

[9] S. Chaudhuri and L. Gravano. Evaluating Top-k Selection Queries. In *VLDB*, 1999

[10] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *VLDB*, 1998.

[11] Y.E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query-Answers . In *VLDB*, 1999.

[12] J. McHugh *et al.* Lore: A database management system for semistructured data. In *SIGMOD*, 1997.

[13] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.

[14] WordNet. http://www.cogsci.princeton.edu/ wn/.

[15] V. Poosala and V. Ganti. Fast Approximate Answers to Aggregate Queries on a Data Cube. In *SSDM*, 1999.

[16] D. Shasha, J. Wang and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *PODS*, 2002

[17] T. Schlieder. Similarity Search in XML Data using Cost-Based Query Transformations In *WEBDB*, 2001

[18] Semantic Web. http://www.w3.org/2001/sw

[19] The SODA Research Group. *The Semistructured Object Database System*. `http://dba.cse.unsw.edu.au`

A Appendix: Encoding Schemes

Efficient implementation of the similarity score algorithm (figure 5) relies largely on our encoding schemes, which are an extension of the encoding schema presented in [5]. The encoding schemes are reproduced here.

The overall aim of the encoding is to represent graphs in as small a space as possible, in a manner which facilitates proximity deviation calculation. It is important that this calculation can be efficiently made by considering only the encodings themselves.

In order to efficiently encode a subgraph, each edge in the main graph is assigned the smallest unused positive number which is unique *only amongst all 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 the edge identifier. This decision has the important consequence that all such numbers will be relatively small compared to the total number of edges/vertexes in the graph.

A.1 Representing Single Paths

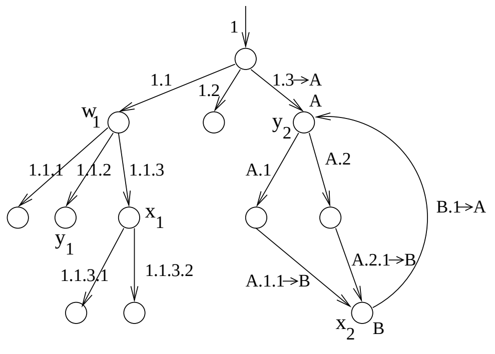

Figure 12: Graph with Indicated Edge Identifiers

Paths in the main graph are identified by the sequence of individual edge identifiers, which implicitly start from a (virtual) incoming edge to the root. Nodes are identified as being the terminus of one or more paths. This concept is illustrated in figure 12. The node y$_1$ is identified by the sequence of edge identifiers ”1.1.2”. Note that this sequence of edge identifiers both uniquely identifies the node itself and the path from the root to the node.

A.2 Representing Multiple Paths

The method described in section A.1 is extended to represent general subgraphs. Suppose we want to encode the subgraph containing all paths from the root to y_2. This must include not only the direct path to y_2, but the cycle from y_2 to itself. Obviously our method of listing edges sequentially is not sufficient when multiple paths are involved.

To deal with multiple paths in a subgraph, we specially number nodes which contain either more than 2 incoming or more than 2 outgoing edges, *within a single subgraph*. Note that we are not concerned about the total number of incoming and outgoing edges from a node. We are only concerned with the number of incoming and outgoing edges which are included in the subgraph of interest. This is illustrated in figure 12 by the nodes labeled ”A” and ”B”. Note that even though many nodes in the graph have more than 2 incoming or outgoing edges, within the subgraph containing all paths from the root to y_2, there are only 2 such nodes.

Such nodes (referred to as *common nodes*) are numbered separately from the edge identifier numbering. In figure 12, the node y_2 is labeled ”A” for clarity. In the encoding scheme, y_2 is implemented as the number ”1”, with a marker bit set to indicate this number refers to a common node and not an edge identifier. Common nodes are given numbers which are unique *within the subgraph being encoded*. This is generally substantially smaller than the total number of such nodes within the entire graph. (Thus, for example, a different common node may also be identified as ”A” in a different subgraph).

The efficiency of the deviation proximity determination algorithm requires that we store the minimum distance from each common node to the terminal node, considering only forward edges. The mechanism for labeling multiple paths can now be seen. Figure 13 gives an overview the encoding scheme.

1. The method used in section A.1 (path encoding) is used as long as the subgraph contains only a single path.
2. Each common node is included in the encoding.
3. One greater than the minimum distance from the above common node to the terminal node (min dist + 1) is included in the encoding.
4. Each path encoding segment (except the first) is assumed to originate from the *previous* common node in the encoding. The first path encoding segment is assumed to begin with the (virtual) incoming root edge.
5. Each path encoding segment is followed by the common node where the path encoding segment terminates.

Figure 13: Overview of Multi-Path Encoding

The entire encoding for the subgraph of all paths from the root to y_2 is therefore given by the mapping:

$$\texttt{1.3.} \rightarrow \texttt{A.A.1.1.1.} \rightarrow \texttt{B.2.1.} \rightarrow \texttt{B.2.1.} \rightarrow \texttt{A}$$

where A indicates a common node, and $\rightarrow$A indicates that a path encoding segments terminates at common node A.

Given that each of these numbers are represented in the minimum possible space, the entire subgraph is represented in only 48 bits.

A.3 Representing Multiple Elements

So far the encoding methods we have looked at have all had only a single node of interest (the terminal node of all the paths). Obviously, to represent a subgraph containing many nodes of interest, we need some mechanism for representing multiple terminal nodes.

Recall that in figure 13, we include a value to indicate the minimum distance from the common node to the terminal node. If we change this definition slightly, so that we store the distance for each common node *and* each terminal node, and the number indicates the minimum distance to the *nearest* terminal node, we then have an easy means of identifying terminal nodes.

A terminal node is thus identified as any node with a pre-computed distance of zero. As both common nodes and terminal nodes have the shortest distance explicitly stored, they are collectively referred to as *annotated nodes*.

In order to accommodate multiple terminal nodes, therefore, the mechanism described in figure 13 is modified as follows:

- All references to ”common nodes” are replaced with references to ”annotated nodes”.
- Replace step 3 with:

  $3'$ One greater than the minimum distance from the above common node to the *nearest* terminal node (min dist + 1) is included in the encoding.

Note by including the minimum distance to the *nearest* terminal node, we alleviate the need to consider each terminal node for a single deviation proximity calculation. This minimum is determined by considering the encoding itself, and so is very fast in practice. Typically, a single multi-element encoding is constructed by combining other encodings (either path encodings or multi-path encodings). If the definition of distance considers only shortest paths passing through common ancestors, the minimum distance can efficiently be incrementally calculated as the multi-element encoding is generated.

A.3.1 Deviation Scores

Looking ahead a little, we need to store progressive deviation score values at certain nodes, as follows:

- Insert the following steps after step $3'$:

  $3'$(a) Include $ProgScore(r_b)$ in the encoding, where r_b is the nearest terminal node to the current annotated node, AN, in step $3'$. (If $\exists r_{b_i}, r_{b_j}$ s.t. $dist(AN, r_{b_i}) = dist(AN, r_{b_j})$, include $\min(ProgScore(r_{b_i}), ProgScore(r_{b_j}))$.

Storing the progressive score allows the proximity determination algorithm to efficiently cumulate this score. Storing the score enables us to obtain the progressive score for the appropriate annotated node, without the need for visiting that node. A progressive score needs to be included when we generate a multi-element encoding which represents the tail nodes of $Q^T(a, b)$, and a progressive score exists for these nodes.

A.4 Implementing the Encodings

Our encodings are implemented in a data structure called a compressed array. Compressed arrays are specifically designed to store, compare and examine lists of numbers, requiring minimal storage space. Numbers are typically stored in a compressed array in twice the minimum number of bits (for example, the number ”1” is stored using 2 bits, the number ”3” requires only 4 bits). This provides a substantial space saving over typical implementations which store numbers in 4 byte integers. The other feature of compressed arrays is their ability to compare entire sequences of numbers in a single bitwise operation. Compressed arrays are described in detail in [5].