Efficiently Handling Wildcard Queries in XML

Neha Singh
IIT Bombay, Mumbai, India
neha_ns@cse.iitb.ac.in

Nitin Gupta
Cornell University, Ithaca, NY 14850
niting@cs.cornell.edu

ABSTRACT
A fundamental problem in the field of querying repositories of XML documents is the non-existence of an efficient solution to queries involving wildcards. Often, a wildcard node in a XML query either refers to an XPath step with the wildcard nodetest, or as a specifier for a set of nodes. We introduce a navigational model for querying XML data over distributed warehouses that efficiently processes such complex twig query patterns. Our model follows the ancestral path of nodes which have a high probability of returning the answers, and therefore intelligently computes the needed structural identifiers. A thorough experimental evaluation on a synthetic and a real dataset demonstrate the efficacy of our approach.

1. PROBLEM AND MOTIVATION
XML is emerging as a standard for information representation and exchange on the World-Wide Web. Therefore, the problem of querying XML documents has recently been a topic of much attention. Although different query languages differ in grammars, they share a common feature, i.e. tree patterns. Also known as twigs, these tree patterns involve element selections with specified tree structures, and have been defined to enable efficient processing of the structural part of the queries.

A fundamental problem in the field of querying these distributed repositories of XML documents is the non-existence of an efficient solution to queries involving wildcards. Often, a wildcard node in a XML query either refers to an XPath step with the wildcard nodetest, or as a specification for a set of nodes. For example, consider the twig patterns given by Figure 1. The * in Q1 can be replaced with any node satisfying the structural constraints imposed by the query, and therefore acts as a wildcard nodetest. The *color** node in Q2, however, refers to the set of nodes whose tag begins with the keyword *color*. Such queries are commonly used when the element names are unknown or do not matter. Wildcard nodes are also used a common shorthand notation to represent a set of element names, for example, the *color** node in the second query. Another important application of wildcard nodes is in secured XML documents where some element names are intentionally removed to hide their original labels [11].

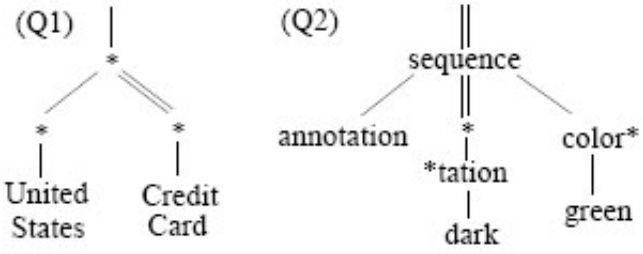

Figure 1: Twig Pattern for queries Q1 and Q2.

State-of-the-art algorithms for matching XML query twig pattern not involving wildcards, such as Holistic Twig Join [7] use structural identifiers (*DocId, LeftPos : RightPos, LevelNum*) to capture the structural relationship between the nodes in the XML database. Hence they fail on such wildcard queries because the structural identifiers corresponding to wildcard node of the query are unknown. Commonly known solutions to such incomplete queries include:

- **Sophisticated storage schemes.** Path indices and Dewey IDs encode complete path information for each node, and use these as opposed to the traditional structural indices [15]. However, path indices are again restrictive as they do not support predicate on non-leaf nodes in a query. They also occupy much more space than structural indices, and require a much longer prefix join increasing the computation time manifold [16].
- **Postprocessing.** Post-processing is the most widely adopted technique for solving incomplete queries, and involves verifying the documents after solving sub-queries [1]. However, note that in a massive pool of documents, this involves pushing queries to the data warehouse, or transferring irrelevant documents (completely) over the network, which adds to the computation cost.

Wildcard steps are expensive to evaluate using a naive implementation. XPath queries involving wildcards are P-complete, and even the most efficient algorithms today typically requires multiple nested scans over the entire XML repository [12].

We introduce the Navigational Selection Algorithm (NSA), a step towards efficiently processing queries involving wildcards. NSA is a navigational model for querying XML warehouses. In a nutshell, NSA follows the ancestral path of nodes which have a high probability of returning the answers, and therefore intelligently computes the needed structural identifiers. NSA performs a backward search [5, 14] on the XML graph and interleaves this with the state-of-the-art structural twig pattern solvers to find the answers efficiently. We present the effectiveness of our model using a wide range of queries over the benchmark XMark dataset [18], and the real ProteinDB dataset [4], distributed over a number of remote repositories. We show

that our model not only performs better in query execution time, but also drastically reduces the total amount of data transferred over the network for distributed repositories.

2. RELATED WORK

To the best of our knowledge, the only previous works that deal with wildcards are [8, 15]. [8] focuses on query reduction and does not propose any efficient algorithm for handling wildcards per se. The proposed layered axis is not generic enough to handle all types of wildcards. [15] is extremely inefficient even when we consider queries with simple string-content matching predicates on non-leaf nodes. On the technical aspect of the approach, the closest research we found was MTrees [16] that uses a specialized index structure, and generic search algorithms [19]. MTree is an index that allows efficient graph traversal, and integrates algorithms to solve XPath queries. However, they do not focus on the specification of the query, and therefore end up suggesting only an alternative to structural joins [19]. There has been considerable work in the field of reachability and efficient labeling of graphs for structural joins [10, 20, 9, 7]. But as mentioned before, all of these are complimentary to what we propose in this paper, and could easily replace the twig pattern solver of the system. There has also been a lot of research on searching in graphs [2, 5, 19]. We use one of these which we discovered had the requisite properties for XML graph search. The last main related area of research has been on node ranking [13, 3]. Though all of these are really good for keyword search or ranking the answers, they are not the best for graph traversal.

3. BACKGROUND

We discuss certain aspects of the Bidirectional Search algorithm [5] and the Activation Model [14] that are quintessential for the understanding of our model. Bidirectional Search was proposed by Kacholia et al. [2, 5, 14] for schema-agnostic text search on graphs. Given a keyword query, it aims to find a minimal subtree in the data graph that contains at least one node corresponding to each keyword. The weight of an edge along reflects the strength of the proximity relationship between two nodes (small values correspond to greater proximity). Each answer tree obtained is assigned a relevance score, and answers are presented in decreasing order of that score. Finding such a subgraph is a NP-complete problem (computation of minimum Steiner trees). The search algorithms described in the paper offer heuristics for incrementally computing query results.

An interesting aspect of this model is the activation spread. The bidirectional algorithm has just two iterators for traversal, and therefore it prioritizes the nodes on some basis for execution. The authors propose a novel prioritization scheme based on spreading activation (a kind of Pagerank [6] which decays with distance). The overall tree score can depend on either the edge score or the node prestige, and both need to be taken into account when defining activation to prioritize search. Higher the node prestige for a node, higher is its priority for expansion. This technique allows preferential expansion of paths that have less branching, and the same mechanism can be extended to implement other useful features, such as enforcing constraints using edge types to restrict search to specified search paths, or prioritizing certain paths over others. We leverage upon this property of the algorithm in order to evaluate twig patterns over graphs of XML repositories.

4. PROPOSED SOLUTION

4.1 DATA MODEL

The XML document is represented as a directed acyclic graph. Here we discuss a heuristical ranking mechanism to calculate the node and edge weights for the XML graph that allows efficient processing of queries involving wildcards. Nodes in the graph represent XML elements and edges represent the parent-child relations, inter-document and intra-document links between the XML elements.

Node Weights: Node weights form a key ingredient of the navigation algorithm as they determine how the graph is traversed. As you will later, we use the navigational model to locate good candidates for NSIs occurring in the upper portion of the query twig pattern and post-processing for those occurring near the leafset. Thus for efficient navigation of XML database, we assign higher weight to nodes that have higher probability for being solutions for the wildcard nodes occurring in the upper portion of the query twig pattern.

1. **Depth/Level.** A node at a lower depth clearly has a high probability of being among the ancestor nodes. Therefore, the depth of a node is near-inversely proportional to the weight of a node.
2. **Subtree Size.** A large subtree size effectively means that the given node has a large number of descendants, and thus reflects that a node a higher probability of being among the ancestor nodes. Therefore, the subtree size is directly proportional to the node weight.
3. **Popularity.** We derive the concept of *popularity* fromPageRank [6], wherein not only does the size of the subtree affect the weight of a node, but also the weight of nodes in this subtree. The popularity of a node is defined as a function of the weight of nodes in its subtree, excluding itself. Popularity also allows us to take into account the frequency of keywords occurring in that particular node and its subtree.

Based on the above factors, we use the following normalized form of node weights:

$$\mathcal{W}(n) = \mathcal{P}(n).\frac{log(|\mathcal{S}(n)|)}{\mathcal{D}(n)} \quad \text{where} \quad \mathcal{P}(n) = \sum_{m \in \mathcal{S}(n)} \gamma_1^{\mathcal{D}(m)} \mathcal{W}(m).\frac{1}{\sum_{w \in \mathcal{T}(n)} \mathcal{R}(w)}$$

In the above equations, S(.) returns the subtree of a node, D(.) returns the depth of a node in the document in which it occurs,

T(.) retuns the set of keywords in a node, and R(.) returns the rank of a keyword, which is directly proportional to its frequency of occurrence. The function P(.) is PageRank-style popularity function, which takes into account the weight of the subtree nodes. γ_1 represents the decay in the effect due to distance (for example, far-lying nodes have less effect on the popularity of a node).

Edge Weights: The edge weights play an important role in activation spread, and therefore must be chosen based on the weights of adjacent nodes. We would also ideally like to spread more activation to ancestor nodes than descendant nodes. Therefore, we use the following edge weights: For a node *n*, let A be the set of nodes that have an outgoing edge to *n*, D be the set of nodes that have an incoming edge from *n*, and E be the set of edges. Then the weights of edges (m, n) $\in$ E,m $\in$ A and (n,m) $\in$ E,m $\in$ D are given by (a) and (b), respectively.

$$\text{(a)}\ \frac{\mathcal{W}(m)}{\sum_{p\in A}\mathcal{W}(p)}\cdot\frac{\gamma_2\times|A|}{\gamma_2\times|A|+|D|} \qquad \text{(b)}\ \frac{\mathcal{W}(m)}{\sum_{p\in D}\mathcal{W}(p)}\cdot\frac{|D|}{\gamma_2\times|A|+|D|}$$

where γ_2 represents the skew in activation flow among ancestors and descendants. The first fraction in (a) gives the relative weight of one node compared to other competing nodes.-The second fraction in (a) gives the fraction of the total activation that *n* should pass to nodes which have an incoming edge to *n.* The formula in (b) similarly corresponds to nodes to which *n* has an outgoing edge.

4.2 NSA ALGORITHM

In the following subsections, we first describe the inverted index used by our system, then the query model used for querying the XML data graph, and finally the Navigational Selection Algorithm.

4.2.1 Indexing To enable finding the list of structural identifiers containing keywords from the query tree pattern, an inverted index is constructed, where each keyword maps onto a list of structural identifiers, each of which is given by (*DocID, Start:End, Level, Pointer to Node in Data Graph*). For optimized query processing, the inverted index used by our system requires the presence of a function which estimates the number of structural identifiers for any given regular expression. This is to check whether or a not a node in the query is a good candidate for inclusion in the holistic join. Consider the prefix-based index structure as shown in Figure 2. Given the keyword "a*", both "asia" and "africa" would have to be considered. But an estimate on the number of structural indices for "a*" is rather easy to pre-compute, and can immediately return the number 35. For large inverted indices, the structural identifiers are distributed over a number of remote machines using DHTs or equivalent techniques. We present the effect of both of these options in the experimental evaluation.

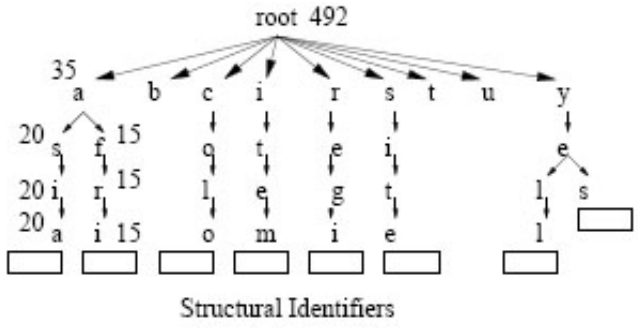

Figure 2: A prefix-based index structure with counts

4.2.2 Queries and Conversions In order to process the query efficiently, we apply certain conversion rules to the query. This is to allow the integration of wildcards in specifying nodes of the query tree pattern. Consider a query Q whose twig pattern is represented as a tree. Let the set of nodes of this tree be given by V_Q. Let $T_Q(.)$ return the keywords present in a query node. We categorize the possible keyword forms in a node as:

- **Fully specified words**. Any words not involving wildcards are termed as fully specified words. In Q1 and Q2, these correspond to the words United, States, Credit, sequence, annotation, green etc. Finding the structural identifiers of the nodes containing these words is trivial and any standard index can be used.
- **Prefix-based words**. These are the words without a preceding wildcard (for example, *color**). Finding the structural identifiers of the nodes containing these words is also relatively simple by using non-trivial but popular index structures such as the prefix-based index discussed earlier.
- **Prefix-void words.** These are the words with a preceding wildcard (for example, **tation*). Finding the structural identifiers of the node containing such words is non-trivial and requires complex pattern-matching based indices. For simplicity of indexing, our index structure currently does not return any structural identifiers for such keywords, and implementing the same is a part of ongoing research.
- **Unspecified words.** Unspecified words are wildcards (*) which do not provide any information about the keyword - the keyword may not even exist.

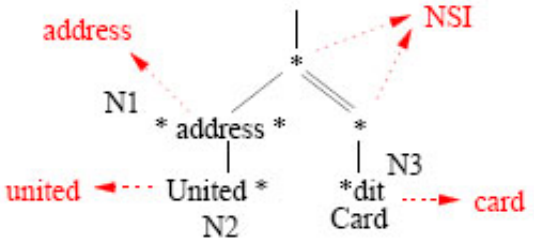

Figure 3: Query with different keyword types

A query node may consist of multiple keywords, which may fall in different

categories. For example, in Figure 3, the nodes N1,N2 and N3 have the set of keywords {*, address}, {united, *} and {*dit, card}, respectively. Let L(w) be the structural identifier list returned by the inverted index for a keyword w, and $L_Q(n)$ be the function that computes the identifier list for a query node n. We propose the following conversion rules for keywords and identifier lists that simplify query processing:

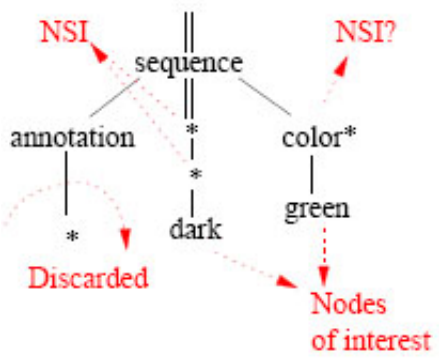

Figure 4: Query Preprocessing for NSA

1. $L(*) = L(*w) = \infty$. This rule says that the identifier list corresponding to unspecified words and prefix-void words is unknown, i.e., any possible identifier can be used in their place. We term such lists as *infinity* lists.
2. $L(w) \cap \infty = L(w)$. This rule states that the intersection of an identifier list with an infinity list returns the identifier list. For example, the identifier list corresponding to the query node "United *" is same as the list for the keyword "United".
3. $L_Q(n) = \cap_{w \in T_Q(n)} L(w)$. This rule is interesting for nodes containing more than one keyword. Given a set of keywords for a query node, this rule finds the intersection of the identifier lists corresponding to the keywords of the node.

4.2.3 Algorithm

The Navigational Selection Algorithm is a multi-phase process, and tries to optimize the query and intermediate results in each of these phases. In a nutshell, the NSA proceeds as follows: the algorithm identifies a set of nodes in the data graph and begins a backward search from these nodes in order to find the needed structural identifiers. Intermittently, it performs a twig pattern join using the known identifiers and the NSI found by backward search to obtain final answers. NSA also employs a twig pattern solver. Although any twig pattern solver that uses the aforementioned structural indexing can be employed, in order to simplify the explanation of the algorithm, we use the Holistic Twig Join [7] as the base twig pattern solver.

Query Preprocessing. In the first phase, NSA fetches the identifier list for each query node using the conversion rules specified above. Some of these identifier lists might be infinite, resulting in Needed Structural Identifiers (NSI). If there is no descendant query node for a NSI node, then we might as well discard it (example the * node below "annotation" in Figure 4). This is a reasonable assumption since such query nodes only test the presence of a node in the documents without any structural constraints. The immediate descendants of NSI nodes in the twig query pattern that are not NSI nodes themselves are called *Nodes of Interest (NOI)* (example the keyword "dark" in Figure 4). Others nodes are treated either as NSI if the identifier list is very big, or as ordinary nodes for reasonable sized identifier lists. Upon obtaining the nodes corresponding to NSI, and subsequently the nodes of interest, the preprocessing step follows the pointer from structural identifiers of the nodes of interest to mark the corresponding nodes in the data graph as *search roots*.

Navigation and pattern matching. In this phase, the NSA starts a search in the data graph beginning with the search roots. Search roots are nodes in the data graph that correspond to the nodes of interest located by the preprocessing step. Backward search beginning from these nodes is an attempt to locate the corresponding NSI in the data graph. By exploring nodes, the algorithm populates the lists for NSI. Intermittently, it performs a twig pattern join (holistic-twig-join) using the obtained identifier lists to get answers. However, if the identifier list corresponding to a NSI is small (i.e. the NSI query node does not have enough candidate identifiers), NSA increases the activation of this particular node's subtree and further explores the graph. It then attempts a join, and keeps iterating this process until all answers have been found.

Algorithm 1 gives the pseudo-code for second phase of the Navigational Selection Algorithm. It takes as input *NOISet*, the nodes of interest, L_Q, a mapping from query nodes to known identifier lists (initially empty for NSI nodes), and *NSIMap*, a mapping from NOI nodes to the corresponding NSI query node. The algorithm constructs a mapping from nodes in the data graph to query nodes, given by *qnode*. NSA also maintains a priority queue, *queue*, which is used to prioritize the nodes for searching. The *Navigate()* function explores α_1 nodes at a time. It removes from queue α_1 nodes with the maximum priority and adds their corresponding identifiers to the identifier list of the query node that they may belong to. It then adds to *queue* the parents of these nodes, or in case the parents are already in the *queue*, increases their priority. This is synonymous to backward search discussed earlier, with activation spread based on node weights.

A number of parameters such as α_1, β_1, β_2, γ_1, γ_2 and γ_3 have been used in the NSA and weight functions. These parameters have to be specified by the system administrator based on the dataset and workload. Automating this is a part of ongoing research. The use of these various parameters is as follows: α_1 is the amount of graph to be explored before a join is attempted. It works in harmony with β_1, the minimum number of candidate identifiers required for NSI nodes before holistic twig join is called. β_2 is the measure of the size of graph to be explored before the algorithm quits. To find all answers, β_2 can be assumed to be |V|. $\gamma_1$and γ_2 have been explained earlier. γ_3 is the amount of "priority" or "activation" increase per node during each iteration – it effectively strikes a balance between a depth-first and a breadth-first strategy for exploring the graph.

Answer Post-processing. The answers obtained from the first two phases of NSA may not be entirely correct. Recall that

prefix-void words had effectively been converted to unspecified words by query conversion, and some node tests were discarded in the preprocessing phase. Answer post-processing is necessary to ensure the correctness of returned answers. We assume that the occurrence of such keywords in limited, and therefore will not result in large unnecessary data getting transferred over the network. The experimental results validate this.

5. UNIQUENESS OF THE APPROACH

The algorithm described above addresses an unsolved problem in the field of complex twig patterns. We use the navigational model to locate good candidates for wildcard nodes occurring in the upper portion of the query twig pattern and post-processing for those occurring near the leafset. To prevent the wasteful exploration of the XML graph thus cutting down on resulting processing and communication overhead, we have developed a ranking mechanism for prioritizing exploration of good candidate nodes. The experimental results show the efficacy of our solution.

6. EXPERIMENTAL RESULTS

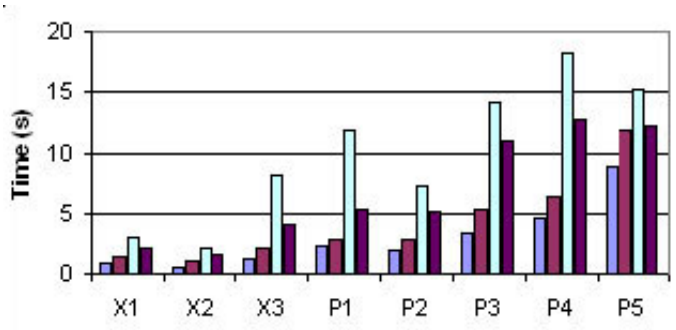

Figure 5 (a): Query Execution Time

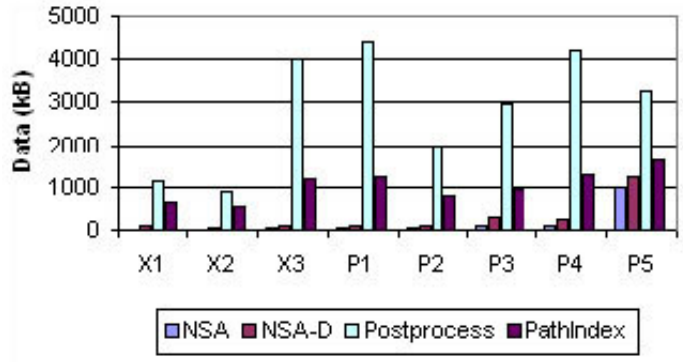

Figure 5 (b): Total data transfer

Algorithm 1 Navigational Selection Algorithm

Require: *NOISet*: Set of NOI nodes in the query
Require: $\mathcal{L}_Q()$: Query node →list of identifiers
Require: *NSIMap()*: NOI →NSI node

Algorithm-NSA()

1: **let** *qnode*(): data nodes → query nodes
2: **let** *queue* be an empty priority queue
3: **for** each node n in *NOISet*
4: **for** each identifier i in $\mathcal{L}_Q(n)$
5: $m \leftarrow$ node in data graph pointed to by i
6: *qnode(m)* $\leftarrow n$
7: add m to *queue* with priority $\mathcal{W}(m)$
8: **let** *explore*= 0; *answers*= 0
9: **while** (*explore* $< \beta_2$)
10: **for** each node n in *NOISet*
11: **if** $|\mathcal{L}_Q(NSIMap(n))| < \beta_1$
12: **for** each identifier i in $\mathcal{L}_Q(n)$
13: increase priority by γ_3 in *queue* of the
14: node pointed to by i if it is still in *queue*
15: *explore*←*Navigate*(*explore*)
16: **let** *join*=*true*
17: **for** each node n in *NOISet*
18: **if** $|\mathcal{L}_Q(NSIMap(n))| < \beta_1$
19: *join*←*false*
20: **if** *join*
21: *answers*←*SSA*()

Navigate()

22: **loop** α_1 times
23: **let** n be the node in *queue* with max priority
24: **if** *qnode(n)* $\neq n$
25: $\mathcal{L}_Q(NSIMap(qnode(n))) \leftarrow \mathcal{L}_Q(NSIMap(qnode(n))) \cup$ identifier of n
31: **for** each parent m of n in the data graph
26: **if** m is not in *queue*
27: *qnode(m)* $\leftarrow$ *qnode(n)*
28: add m to *queue* with priority $\mathcal{W}(m)$
29: **else if** m is already in *queue*
30: increase priority of m by $\gamma_3 \times \mathcal{W}(n)$ in *queue*
31: remove n from *queue*
32: *explore* $\leftarrow$ *explore* +1

We implemented the proposed system in Java. The experiments were performed on a 1.63GHz Lenevo Laptop with 2GB of RAM, with 10 remote machines have a 2GHz processor with 1GB of RAM each that mainly acted as data stores, spread over a local intranet. The datasets used were the well known synthetic dataset XMark [18] (100-400MB) and a real dataset ProteinDB [4] (250MB), evenly and randomly distributed over the remote machines. All times are trimmed averages of 10 runs. Given the compact representation of the graph, and selective search over the dataset, we also tested the NSA where the index structure was distributed over remote machines and only the data graph was present on the host machine (as described in Section 4.1). 8 different types of queries were performed on the XMark and ProteinDB datasets. For comparison, we implemented the postprocessing technique discussed earlier, and an algorithm that efficiently used PathIndices to solve the queries. Figure 5(a) and Figure 5(b) show a snapshot of the results obtained for query execution time and data transfer, respectively. NSA is the navigational algorithm with the inverted index at the host machine, while NSA-D has the distributed index structure. *Postprocess* is the naive technique of processing after obtaining all answers. For PathIndex, we picked time when the index is at the server, whereas we picked data transfer over the network with an implementation having a distributed index. X_i are the queries executed on XMark and P_i represent those executed on ProteinDB.

The results clearly show at least a two-fold improvement over path-index and multiple orders of magnitude improvement over the naive postprocessing technique.

XMark. The queries over XMark did not include suffix-based queries, and therefore did not require any kind of postprocessing. The results show that both NSA and NSA-D, despite the network data transfer, outperformed PathIndex, though not by big margins. However, the gain in time is primarily due to reduced processing due to structural indices used by NSA as opposed to the memory intensive path-indices, which do not have associated algorithms such as holistic twig joins.

ProteinDB. ProteinDB is a more versatile database. The node variety of nodes for ProteinDB in terms of content, depth and replication is much wider than the synthetic XMark. Therefore, we did a larger number of our tests on ProteinDB. NSA and NSA-D showed a two-fold improvement in query execution time over path indices, and multiple orders of magnitude improvement in the amount of data transferred over the network. An exception to this is the last query, which included suffix-based wildcard nodes, which led to NSA and NSA-D performing as poorly as the naive algorithm.

7. CONCLUSION

Using the rich search framework and popular techniques to solve a simple twig pattern solver, we have developed an efficient system for solving more complicated twig patterns involving incompletely specified keywords and nodes. Using a comprehensive set of experiments, we showed that our algorithm has stability, efficiency and the ability to perform on a distributed XML warehouse. Furthermore, our algorithm is often orders of magnitude faster than any of the existing techniques to solve the incomplete query patterns.

8. REFERENCES

[1] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying peer-to-peer warehouses of xml resources. In ICDE, 2005.

[2] B. Aditya, S. Chakrabarti, R. Nasre, R. Desai, A. Hulgeri, S. Sudarshan, and H. Karambelkar. User interaction in the banks system. ICDE, 2003.

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In VLDB, 2004.

[4] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P.E. Bourne. The protein data bank. Nucleic Acids Res, 28(1):235–242, January 2000.

[5] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In ICDE, 2002.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In WWW, 1998.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In SIGMOD, 2002.

[8] C. Y. Chan, W. Fan, and Y. Zeng. Taming xpath queries by minimizing wildcard steps. In VLDB, 2004.

[9] L. Chen, A. Gupta, and M. E. Kurul. Efficient algorithms for pattern matching on directed acyclic graphs. In ICDE, 2005.

[10] E. Cohen, E. Halperin, H. Kaplan, U. Zwick. Reachability and distance queries via 2-hop labels. In SODA, 2002.

[11] W. Fan, C. Y. Chan, and M. Garofalakis. Secure xml querying with security views. In SIGMOD, 2004.

[12] G. Gottlob, C. Koch, and R. Pichler. The complexity of xpath query evaluation. In ACM PODS, 2003.

[13] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: ranked keyword search over xml documents. In SIGMOD, 2003.

[14] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In VLDB, 2005.

[15] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In VLDB, 2005.

[16] P. M. Pettovello and F. Fotouhi. Mtree: an xml xpath graph index. In SAC, 2006.

[17] J. Robie, M. Kay, D. Chamberlin, A. Berglund, M. F. Fernández, J. Siméon, and Scott Boag. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007.

[18] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In VLDB, 2002.

[19] D. Shasha, J. T. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In PODS, 2002.

[20] H. Wang, H. He2, J. Yang, P. S. Yu, and J. Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In ICDE, 2006.