



An indexing Technique to Speed up XML Query Processing: PLIT

Ajit Pratap Singh¹, Er. Narendra Kumar Gupta²

M .Tech Student, Dept. of CSE, S.H.I.A.T.S -Deemed University, Allahabad, India¹

Asst. Professor, Dept. of CSE, S.H.I.A.T.S -Deemed University, Allahabad, India²

ABSTRACT: The XML format has become the standard for data exchange because it is self-describing and it stores not only information but also the relationships between data. Therefore it is used in very different areas. To find the right information in an XML file, we need to have a fast and an effective access to data. Similar to relational databases, we can create an *index* in order to speed up the querying for the information. There are several ways of indexing XML data but previous research showed that one of the most effective approaches is to index *root-to-leaf paths* in the input file. So we took the inspiration from existing path-based indexing concepts, enhanced those ideas, and created a new *native XML indexing* method derived from the combination of existing approaches in order to improve the evaluation time of regular path expressions in *XPath queries*.

KEYWORDS: Indexing XML, path-based indexing, XPath queries, regular path expressions.

I. INTRODUCTION

In past few years there has been an expansion of semi-structured data mostly stored as XML files and used for saving and exchanging information over the Internet. The simplicity is only one of the factors why the format became so popular. As more and more files occurred in this format, we wanted to access the stored data and search for the specific information according to prior criteria. For this purpose languages such as XPath [19] or XQuery [20] have been created. They allow searching for elements, attributes, or text values based on either specific values or regular expressions. If there are multiple conditions in an XPath query, we can combine them, and we get the *regular path expression* pattern. The path expression usually matches several elements in the input XML file (the result set). The challenge is to find these elements quickly and efficiently, especially in large files with a high number of elements and with different structures. One came with an idea of *indexing* the XML data in order to quickly get the the results for any query.

No matter which indexing technique we use, if we had an XPath expression, the most problematic queries would be those with *'/'* (relative paths) or (wildcards). These queries match numerous distinct elements and are difficult to handle compared to expressions with absolute paths only.

In this paper, our goal is to combine the best concepts of existing indexing methods and enhance them in order to improve the evaluation time of XPath queries. To achieve this, we make contributions to these areas:

- The previous research showed that indexing paths is one of the effective ways of indexing XML documents, so we use this approach and we create *a new indexing method* based on indexing paths (Section 2).
- Additionally, we combine it with one of the numbering schemes in order to accelerate the evaluation of XPath regular path expressions (see Section 3).
- Finally we compare the new concept with existing solutions in terms of time complexity while evaluating sample XPath queries (Section 4).

II. XML INDEXING

There are various distinct approaches of indexing XML files. The main difference between them is that each method focuses on the specific topic such as decreasing the number of I/O operations [9], converting the XML format into tables in a relational DBMS to leverage the database engine ([8], [5], [16], [17] or GRIPP [6]), or relying on the simplicity of numbering schemes and joining elements (XISSL [11] or twig joins [18]). The indexes might be based on a

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

known data structure, e.g. the Patricia tries [12] are used in [7] or [9], but more often they use custom structures.

The method of labeling nodes in the tree with numbers might help with discovering ancestor-descendant (A-D) relationships. Dietz's numbering scheme [1] inspired us to design our numbering scheme based on intervals that evaluates A-D relationships in constant time. We found also motivation in XISS for the decomposition of XPath expressions, producing the intermediate results (called candidates in our approach), and element joins.

Specializing on paths, the DataGuide [2] handles raw paths and provided the basis for future path indexing methods such as XDG (Extended DataGuide [4]) or Index Fabric [7].

The interesting work of mapping all root-to-leaf paths into multi-dimensional points (MDX [3] or UB-trees [10]) influenced us on designing our indexing method. These concepts try to avoid using structural joins because of their time complexity compared to indexing paths. We understand the inefficiency of element-based approaches but we also see the potential slowdown for the multidimensional mapping methods when evaluating queries with multiple wildcards and relative paths. In this case, the domain used for finding matching tuples might grow faster.

The aim of the proposed indexing scheme is to follow the multi-dimensional techniques and leverage the path-based indexing with focus on grouping paths according to common characteristics, *path labels* (see Section 2.2). We expect this idea to eliminate many unsuitable paths and, as the result, to speed up the evaluation of any query (especially those problematic ones; see Section 3).

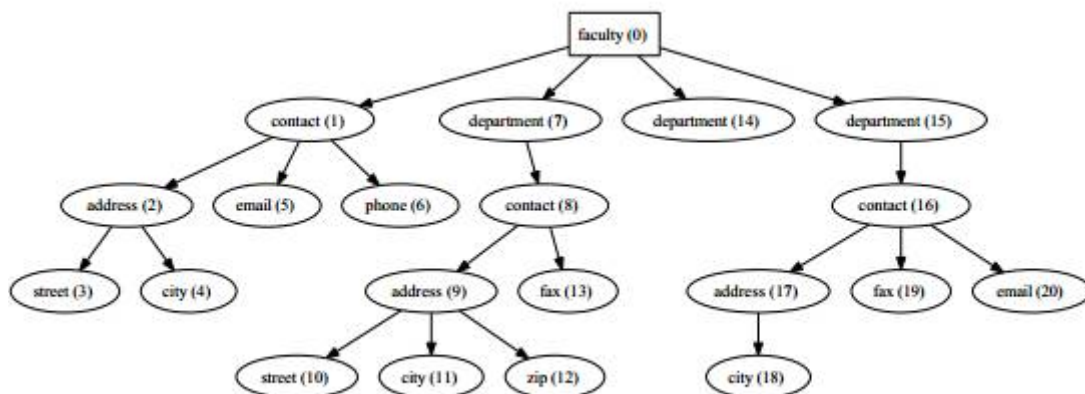


Fig. 1. The sample XML file

III. PATH-BASED INDEXING

First of all, we assign elements in the source file the unique identification numbers (NodeIDs) and convert the element (tag) names into integers (TagIDs). We prefer numbers over strings because the comparison of integers is faster than comparing strings with variable lengths although it brings some memory overhead. We use the numbering method that assigns a node the NodeID when the corresponding element is visited for the first time (using the SAX parsing method). The root has the number 0, while the number of following nodes is always incremented by 1 (see the sample XML file with element names and NodeIDs in Figure 1).

Next, we store all root-to-leaf paths according to their *path labels*. Whenever we reach a leaf node, a new root-to-leaf path occurs, and we store the Path reference according to its path label. While the Path reference is indicated by the sequence of NodeIDs, the path label is determined by the sequence of TagIDs of all nodes on the path from the root to the current (leaf) node.



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

Table 1. Terms definition and description

Term	Description
NodeID	The unique node number given by the numbering method.
LastNodeID	NodeID of the last visited leaf node in the subtree determined by the current node.
TagName	The name of an element.
TagID	The unique number for the TagName. For every new TagName we assign the next TagID (increased by 1).
Path Label	The sequence of TagIDs of the nodes on the path from the
Path Prefix	The sequence of TagIDs that identifies a prefix of at least one path.
Path Template (PTID)	The unique number for a path label. The path labels are converted into integers (PTIDs).
Path	The sequence of NodeIDs of the nodes on the path from the

The path label is stored only once but one path label can contain several Path references. Moreover, we convert the path label into *Path Template ID* (PTID) that groups similar Path references together (for the detailed specification of used terms see Table 1).

Example 1. If we take the sample XML file from the Figure 1 and we visit a leaf node *fax* (13), the Path reference (sequence of NodeIDs) will be (0,7,8,13). Converting element names *"/faculty/department/contact/fax"* to TagIDs, we get the path label *'/0/7/1/9'*. For details about the conversion, see the Table 2(a) (NodeTags table) and the Section 2.3.

a) NodeTags table

(b) Paths table

TagName	TagID	Path Template IDs	PTID	Path label	Path references
faculty	0	{0,1,2,3,4,5,6,7,8,9}	0	0/1/2/3	{{(0,1,2,3)}
contact	1	{0,1,2,3,4,5,6,7,9}	1	0/1/2/4	{{(0,1,2,4)}
address	2	{0,1,4,5,6}	2	0/1/5	{{(0,1,5)}
street	3	{0,4}	3	0/1/6	{{(0,1,6)}
city	4	{1,5}	4	0/7/1/2/3	{{(0,7,8,9,10)}
email	5	{2,9}	5	0/7/1/2/4	{{(0,7,8,9,11); (0,15,16,17,18)}
phone	6	{3}	6	0/7/1/2/8	{{(0,7,8,9,12)}
department	7	{4,5,6,7,8,9}	7	0/7/1/9	{{(0,7,8,13); (0,15,16,19)}
zip	8	{6}	8	0/7	{{(0,14)}
fax	9	{7}	9	0/7/1/5	{{(0,15,16,20)}

Table 2. Structures with data for the sample XML

Structures: While we parse the input XML file, we maintain several structures that store the root-to-leaf paths and other necessary information that we will later use when we evaluate queries. There are three major data structures: the NodeTags table, the Paths table, and the Prefix Tree. While the first two are more like database tables, the last one is definitely a tree structure (inspired by a Patricia trie [12]).

NodeTags table: This table provides conversion between the Tag Name and the TagID. Furthermore it saves all path template IDs (PTIDs) where the given TagID appear; see the Table 2(a).

Paths table: This table contains the conversion between the Path label and the PTID. As mentioned before, a single Path label provides grouping for several Path references that are also stored here; see the Table 2(b). **Prefix tree.** The Prefix Tree covers all distinct *prefixes of path labels* from the source file. Each node in the tree has the TagID

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

and the path from the root to a given node represents the path prefix. Nodes also contain all PTIDs which do have the selected prefix. We use this structure to find all PTIDs to which a specific NodeID might belong ..

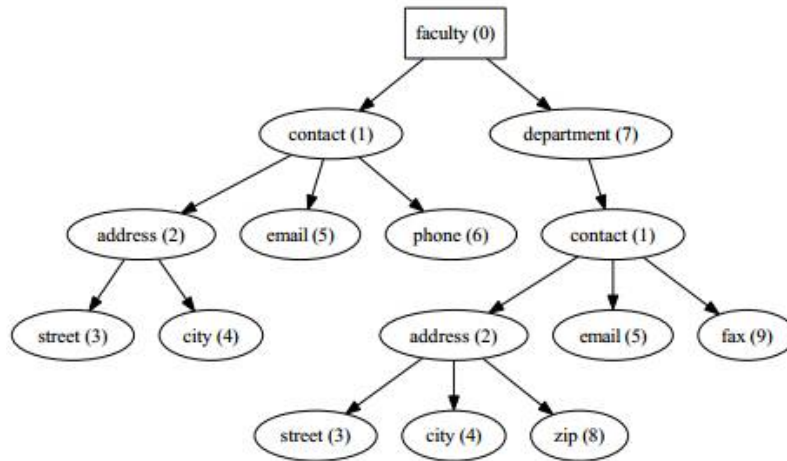


Fig. 2. PrefixTree for the sample XML file

Example 2. If we take the node *contact (1)* from the sample XML (see Figure 1), it belongs to almost all PTIDs (according to the NodeTags table). And using the PrefixTree, we find the prefix of '/0/1' that matches the PTIDs {0,1,2,3}. So the chosen node occurs only on paths with these path labels.

Evaluating XPath queries:

Before we start evaluating an XPath query, we need to parse and prepare the query. We describe these steps in the following sections in more detail.

Parsing XPath expressions:

In the beginning, we convert the string query, that describes a regular path expression, into the structure that is appropriate for the evaluation. We selected the graph structure (XPathTree) created by two types of nodes (XPathNodes): *steps* and *predicates*.

Steps: If we divide the path expression into sequence of step expressions, they will be represented by these nodes.

Predicates: We express the further filter expressions by the *Predicate* nodes. While the *Step* nodes cannot be added as sub nodes, each node in the tree might contain one or more *Predicate* sub nodes.

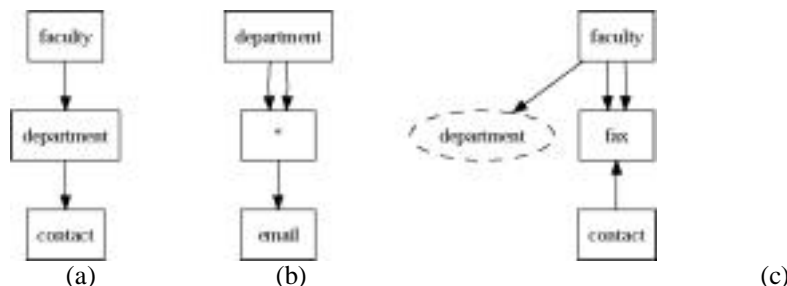


Fig. 3. The tree structure of sample queries.

The *Steps* are represented by boxes, while the *Predicates* are shown as dashed ellipses. We use oriented edges to determine the forward and reverse axes (the edge leads from an ancestor to a descendant). We also put multiple edges to indicate that the node is generally a descendant (not necessarily a childnode) The corresponding XPath expressions are:

(a) `faculty/department/contact`, (b) `department//*[email]`, and (c) `faculty[department]//fax/ancestor::contact`



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

IV. XPATH TREE PRE-PROCESSING

After we parse the XPath expression and create the corresponding XPathTree, we need to convert the stored element names into TagIDs. While converting, we check element names whether they exist in the index. If a name does not occur in the NodeTags table, the result is instantly available because no nodes will be in the result set and no further evaluation is needed (we suppose all predicates to co-exist at the same time).

Algorithm 1 Visit procedure for evaluating a node in the XPathTree **Require:** *xnode* is the current XPathNode in the XPathTree that is being visited :

```
1: xnode.Candidates = GetCandidates()
2: for all (predicate in xnode.Predicates) do
3: Visit(predicate) {recursive call for a predicate}
4: xnode.VoteForCandidateNodes(predicate.Candidates)
5: end for
6: if xnode.HasPredicates then
7: xnode.FilterCandidateNodes()
8: end if
9: if (xnode.IsStepNode) then
10: MergeCandidates (lastNode, xnode)
11: if (xnode.NextStepNode is not null) then
12: Visit(xnode.NextStepNode)
13: end if 14: end if
```

V. XPATH TREE EVALUATING

When we build and validate the XPathTree, we can start the evaluation. We use slightly different evaluating methods according to the type of the current XPathNode. When traversing the XPathTree, the *Step* nodes are evaluated with the top-down approach, while the *Predicate* nodes are processed in the bottom-up style (from the lowest level upwards using depth-first search). The reason for distinct methods is that all predicates must be resolved before we can continue with the next XPathNode.

The Algorithm 14 describes the procedure that we apply to all XPathNodes. There are several steps that we need to explain in more detail but the main principle is that we save candidate nodes (NodeIDs) for each XPathNode we visit. So the candidate nodes of the last node represent the result set.

1. Save candidates

The first step is to save candidate nodes for the current node. We store them in the table called *candidates* (line 1). It contains PTIDs, where the XPathNode occur, with corresponding NodeIDs. Each PTID is stored only once but one NodeID might be saved for more PTIDs. It is because we focus later on merging *candidates* according to PTIDs rather than NodeIDs. To identify the PTIDs, we try to find the smallest PTID set that is common for as many XPathNode nodes as possible (using the NodeTags table). For a *predicate* node, this means to take PTIDs that are common for XPathNodes on the path from the last *Step* node. Because we evaluate predicates bottom-up, we create the set of PTIDs on the way "down". For a *Step* node we take the path from the first *Step* node. This holds only if all axis directions on the path are the same. If we have an alternating¹ XPathNode,

XPathNodes on the path into two groups for which the smallest PTID set must be computed separately. We pre-compute the minimal PTID set for all corresponding nodes when the first node in a specific group is visited.

When we obtain the minimal PTID set, we use the Paths table to find the candidate nodes (NodeIDs) according to the Path references for a PTID. If the TagID does not represent we find the positions of the TagID in the Path label identified by the current PTID. We search only for positions that are either *after* (forward axes) or *before* (reverse axes) the position of the last node and we take all NodeIDs on those positions from the Path references. If the TagID reflects and the XPathNode does not have any predicates, we skip it and save the minimal and the maximal number

¹ Alternating XPathNode is a node that *changes* the axis direction (the incoming edge has different direction than the outgoing edge).



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

of positions to be skipped when searching for positions in the next XPathNode. The numbers are determined by the current axis: (1,1) for the direct relative (parent, child), and (1, TO) for other axes (ancestor, descendant).

Example 3. If we take the Query 3 in Figure 3(c), the alternating XPathNode is the *fax* node. Therefore the minimal PTID subset can be computed only for set of nodes {faculty, fax} and {fax, contact}. The candidates for this XPathNode are shown in Table 3.

Table 3. Candidates table for the *fax node* in Figure 3(c)

PTID	NodeIDs
2 9	{5}
	{20}

2. Evaluate predicates and voting:

If there are any predicates for the current XPathNode, we need to handle them before we continue with the next XPathNode. Because predicates give additional filtering criteria, not all candidate nodes from the current XPathNode's candidates will meet the new criteria. Therefore we use *voting* for candidate nodes (line 4). Every predicate gives a vote for all candidate nodes in the current XPathNode's candidates table (no matter of their PTIDs) that are reachable from a NodeID stored in the predicate's candidates. The reachability is dedicated from the axis type and the NodeIDs. Because we consider only tree structures, we can use the interval (NodeID, LastNodeID) that is available to any NodeID to test whether a path between two NodeIDs exists. The path from a node n_1 to a node n_2 exists only if

$$(n_2.NodeID > n_1.NodeID) \text{ and } (n_2.NodeID < n_1.LastNodeID). \quad (1)$$

3. Filter candidates:

Whenever we use the voting mechanism, we need to finalize the candidates. We call this action *filtering* candidates (line 7). The candidate nodes that have not received enough *votes* will be removed. Number of votes needed for being kept equals the number of predicates that has been included in voting. After we eliminate unsuitable candidate nodes, we need to update the PTIDs for the next step. By updating PTIDs we mean find all PTIDs where a NodeID might occur. If the axis of the next XPathNode has the same direction, we take the PTIDs only from the smallest PTID set for the current XPathNode. Otherwise, we need to consider potentially all PTIDs. To take only correct PTIDs, we use the PrefixTree that determines only such PTIDs in which the given NodeID might occur. We cannot use only the Paths table because we will find PTIDs for any NodeID with the same TagName and that produces bigger set than we need for a specific NodeID. So we use the PrefixTree instead. The path prefix that we need for navigation in the PrefixTree is defined by the current NodeID.

4. Merge candidates:

If we have two *Step* XPathNodes, we need to merge their candidate nodes (line 10). Usually we take the candidates of the last *Step* XPathNode and apply the same voting mechanism on the current XPathNode as with predicates. After voting, we automatically filter candidates. The result for the current XPathNode contains previous candidate nodes that received votes.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a new XML indexing method based on storing root-to-leaf paths and grouping them according to common Path labels in order to enhance the evaluation time of regular path expressions in XPath queries. The experimental results showed that there is an improvement of the evaluation time in the category of small and medium-sized files. Although handling medium files is still comparable to other approaches, evaluating large files and complex queries did not bring the anticipated results.

For the future, there are still several issues in the current prototype that we would like to improve, such as speeding up the branch queries or optimizing the tree structure that stores the XPath query before evaluating. Next, we want to design an optimal structure for saving the *PLIT* index to a hard drive. Furthermore, we would like to provide support for graph-oriented XML files (not only trees) which means to replace the interval-based path testing with a more general structure (such as Rho-index [15]). Finally, we aspire to use our indexing method in the environment of distributed XML processing.



International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 5, May 2016

REFERENCES

1. Paul F. Dietz: *Maintaining a Order in a linked list*. Proceedings of the 14th Annual ACM Symposium on Theory of Computing. San Francisco, California (1982) 122127.
2. R. Goldman, J. Widom: *DataGuides: Enable query formulation and optimization in semistructured databases*. Proceedings of 23rd International Conference on Very Large Data Bases. Athens, Greece (1997) 436-445.
3. M. Kratky, R. Baca, V. Snasel: *On the Efficient Processing Regular Path Expressions of an Enormous Volume of XML Data*. Lecture Notes in Computer Science, Vol. 4653. Springer-Verlag, Germany (2007) 1-12.
4. J.M. Bremer, M.Gertz: *An Efficient XML Node Identification and Indexing Scheme*. Technical Report CSE-2003-04. Dept. of Computer Science, University of California at Davis, (2003).
5. G.Marks, M.Roantree: *Pattern Based Processing of XPath Queries*. IDEAS 2008 - International Symposium on Database Engineering and Applications. Coimbra, Portugal (2008).
6. S. Trifil, U.Leser: *Fast and Practical Indexing and Querying of Very Large Graphs*. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. Beijing, China (2007).
7. B.f. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon: *A Fast Index for Semistructured Data*. Proceedings of the 27th International Conference on Very Large Data Bases. San Francisco, USA (2001) 341-350.
8. Torsten Grust: *Accelerating XPath Location Steps*. Proceedings of the 2002 ACM SIGMOD international conference on Management of data. New York, USA (2002) 109- 120.
9. D. Barashev, B. Novikov: *Indexing XML to Support Path Expressions*. Proc. of the 6th East European Conf. on Advances in Databases and Information Systems (ADBIS 2002), Vol. 2: Research Communications. Bratislava, Slovakia (2002) 1-10.
10. M. Kratky, J. Pokorny, V. Snasel: *Indexing XML Data with UB-trees*. Proc. of the 6th East European Conf. on Advances in Databases and Information Systems (ADBIS 2002), Vol. 2: Research Communications. Bratislava, Slovakia (2002) 155164.
11. Quanzhong Li, B.Moon: *Indexing and Querying XML Data for Regular Path Expressions*. Proceedings of the 27th VLDB Conference. Roma, Italy (2001) 361-370.
12. Donald Knuth: *The Art of Computer Programming*. Volume III, Sorting and Searching, Third Edition. Addison Wesley, Reading, MA (1998).
13. Wolfgang Meier: *eXist: An Open Source Native XML Database*. Lecture Notes in Computer Science, Vol. 2593/2009. Springer Berlin, Heidelberg (2003) 169-183.
14. A. Schmidt, F. Waas, I. Manolescu, M. Kersten, R. Busse, M.J. Carey: *XMark: A Benchmark for XML Data Management*. Proceedings of the 28th VLDB Conference. Hong Kong, China (2002) 974-985.
15. S. Barton, P. Zezula: *Rh,o-i,nd,ex, - An Index for Graph Structured Data*. 8th International Workshop of the DELOS Network of Excellence on Digital Libraries. Schloss Dagstuhl, Germany (2005) 57-64.
16. M. Yoshikawa, T.Amagasa, T. Shimura and S. Uemura: *XRel: a Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases*. ACM Transactions on Internet Technology (TOIT). New York, NY, USA (2001) 110- 141.
17. Zhiyuan Chen and G.J. Korn and F. Koudas and N. Shanmugasundaram and J.D. Srivastava: *Index Structures for Matching XML Twigs Using Relational Query Processors*, Data & Knowledge Engineering. Amsterdam, The Netherlands (2007) 283-302.
18. Chen, T. and Lu, J. and Ling, T.W.: *On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques*, Proceedings of the 2005 ACM SIGMOD international conference on Management of data. New York, NY, USA (2005) 455-466.
19. XML Path Language (XPath) 2.0. <http://www.w3.org/TR/xpath20>.
20. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery>.
21. eXist-db Open Source Native XML Database. <http://exist.sourceforge.net>.
22. Albrecht Schmidt: *xmlgen - The Benchmark Data Generator*. <http://www.xml-benchmark.org>.
23. Qizx, a fast XML repository and search engine fully supporting XQuery. <http://www.xmlmind.com/qizx>.
24. Microsoft SQL Server 2005. <http://www.microsoft.com/sqlserver/2005>.

BIOGRAPHY



Ajit Pratap Singh is born in Uttar Pradesh, India, in 1990. He has received B.Tech degree in Computer Science & Engineering in 2013 from UPTU and currently perusing M.Tech degree in CSE from Sam Higginbottom Institute of Agriculture, Technology & Science, Technology & Science, and Deemed to- be- University in Allahabad (U.P).



Er.N.K.Gupta is working in the department of CSE of SHIATS from last 10 years. He has completed his UG & PG from ALLAHABAD UNIVERSITY. He is pursuing PhD from SHIATS. He is expertise in RDBMS & DATA MINING/OBJECT ORIENTED TECHNOLOGIES field. He has guided more than 40 PhD students & 50 M.Tech and published more than 20 NATIONAL & INTERNATIONAL REPUTATED JOURNALS.