Enhancement XML Documents Query by developing Indexing Technique

Seif ELduola Fath El Rhman El Haj
https://orcid.org/0009-0006-4395-5001
Technology University (Sudan), Seifelduolaf@yahoo.com

Received: 13/05/2025 *Accepted: 13/06/2025* *Published: 01/09/2025*

Abstract:

This paper describe famous sort of storing tree , B+ - tree, XR-tree, and XB-tree and analyzes how well they support XML query. XML is flexible exchange format that has gained popularity for representing many classes of data, including structured documents, heterogeneous and semi-structured records, data from scientific simulations, digitized images, among others. As a result, querying XML documents has received much attention. XML documents are typically queried with a combination of value search and structure search. While querying by values can leverage traditional database technologies, evaluating structural relationship, specifically parent-child or ancestor-descendant relationship, between XML element sets has imposed a great challenge on efficient XML query processing. Many index-based approaches have been proposed for storing and efficient XML queries.

Keywords: XML; B+-tree; XR-tree; XB-tree; XML Query.

INTRODUCTION

XML is the standard language for representing and exchanging semi-structured data in many commercial and scientific applications and much research has been undertaken on providing flexible indexing and query mechanisms to extract data from XML documents [3, 6, 5, 2, 1, 4]. XML data are viewed as an ordered tree structure, and queries are specified using path expressions. Indexing structures used in relational databases are well-known and highly efficient. Using these indexing structures as a starting point for indexing XML documents, a natural evolution in the features and efficiency of said indexes has occurred and will continue to develop.

Queries with a path expression have been one of the major foci of research for indexing and querying XML documents. In the past few years, there have been two main thrusts of research activities for processing path join queries for retrieving XML data, namely, approaches based on *structural index* and *numbering schemes*. The approaches based on the structural index facilitate traversing through the hierarchy of XML documents by referencing the structural information of the documents (*e.g.*, data guide [32], representative objects [36],

1- index [35], approximate path summary [34], F&B index [33]). These structural indexes can help reduce the search space for processing path or twig queries. The other class of approaches are based on a form of numbering scheme that encodes each element by its positional information within the hierarchy of an XML document it belongs to. Most of the numbering schemes reported in the literature are designed by a tree-traversal order (*e.g.*, pre-and post-order [10], extended preorder [16]) or textual positions of start and end tags (*e.g.*, containment property [24], absolute region coordinate [23]). If such a numbering scheme is embedded in the labeled trees of XML documents, the structural relationship (such as ancestor-descendant) between a pair of elements can be determined quickly without traversing an entire tree.

Literature review:

XML table indexes provide a more efficient mechanism for searching data stored in aggregate form. XML table indexes are a set of tables created to project out in column form commonly sought metadata from stored XML documents. By projecting the data includes into column form, queries on the XML documents can be efficiently processed as they can leverage the enhanced functionality provided by the database tables. The XML table indexes may use aliases, partitioning, constraints and other functions to further improve query flexibility and performance.[1]

The exponential growth of bioinformatics in the healthcare domain has revolutionized our understanding of DNA, proteins, and other biomolecular entities. This remarkable progress has generated an overwhelming volume of data, necessitating big data technologies for efficient storage and indexing. While big data technologies like Hadoop offer substantial support for big XML file storage, the challenges of indexing data sizes and XPath query performance persist. To enhance the efficiency of XPath queries and address the data size problem, a novel approach that is derived from the spatial indexing method of the R-tre family.[2]

Structural join has been established as a primitive technique for matching the binary containment pattern, specifically the parent–child and ancestor–descendant relationship, on the tree XML data. While current indexing approaches and evaluation algorithms proposed for the structural join operation assume the tree-structured data model, the presence of reference links in XML documents may render the underlying model a graph instead. [3]

XML Management Systems

There are mainly two types of XML storage considered in the literature: Relational and Native. By relational storage, we mean that XML documents are mapped into relational tables and XML queries are translated into SQL queries. In contrast, in native storage, XML data can be stored in a versatile format and we can evaluate XML queries with algorithms that are tailored for XML.

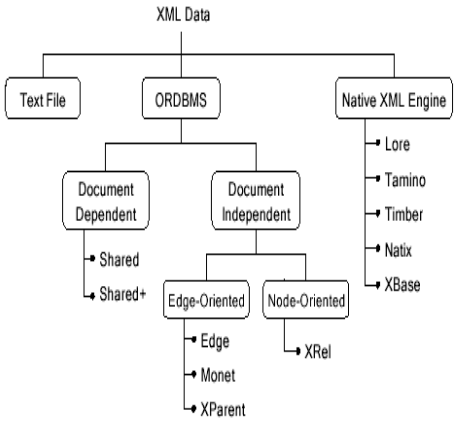

Figure 1: systems for managing XML documents

Relational Storage of XML Data:

Various techniques have been proposed to leverage the power of widely available object-relational databases for storing and querying XML data. The basic idea is that we shred XML documents into relational tables and access the data with SQL queries.

When XML data is stored in a relational database, a relational schema must be defined. We can generate the table schema by either using or not using the schema information of an XML document to be stored. Such schema information could be given in the form of either a Document Type Definition (DTD) or an XML schema [14]. When the relational schema is generated based on the document schema, say DTD, different DTDs will lead to different table schemas, resulting in a document-dependent mapping. On the other hand, since any XML document can be modeled as an ordered tree, a single relational schema is able to describe the tree structure for all XML documents. No DTD information is required by this approach and all XML documents can share the same relational schema, resulting in a document-independent mapping.

Native XML Engines:

Native XML engines are systems that are specially designed for managing XML data. The storage and query processing techniques adopted by different systems may vary from each other in a noticeable way. One approach is to model XML documents using the Document Object Model (DOM) [15]. Internally, each node in a DOM tree has four filiation pointers and two sibling pointers. The filiation pointers include the first child, the last child, the parent, and the root pointers. The sibling pointers point to the previous and the next sibling nodes. The nodes in a DOM tree are serialized into disk pages according to depth-first order (filiation clustering) or breadth-first order (sibling clustering). Lore [13, 11] and XBase [17] are two instances of such a

storage approach. The current release of TIMBER [18] transforms each node of the data tree into an internal representation and stores it into SHORE [19] as an atomic unit of storage. TIMBER is being engineered to package nodes in page-size containers due to SHORE's considerable overheads in dealing with small objects. Natix [20] uses a native storage format with the following features: (1) sub-trees of the original XML document are stored together in a single (physical) record; (2) the inner structure of sub-trees is retained; and (3) to satisfy special application requirements, the clustering requirements of sub-trees are specifiable through a split matrix. Documents stored in Tamino [6,7] are grouped into collections. Within a collection, several document types can be declared and each incoming document validates against one of these types. The elements and attributes parsed from an incoming document can be stored in Tamino itself or in an external/built-in SQL database.

Several methods have been proposed to evaluate queries over a native XML DBMS, where the queries specify both path and keyword constraints. These broadly consist of graph traversal approaches, optimized with auxiliary structures known as structure index.

Index Structures:

Indexing structures used in relational databases are well-known and highly efficient. Using these indexing structures as a starting point for indexing XML documents, a natural evolution in the features and efficiency of said indexes has occurred and will continue to develop. This section starts by introducing a labeling scheme for nodes in a tree, presents preliminary index structures (B+-tree and XR-tree) used for XML documents, moves on to more sophisticated and efficient index methodologies XB-tree.

Node Labeling

When constructing a B+-tree, XR-tree, or XB-tree index on an OEM structure, the nodes must be labeled with a standard labeling scheme. Many labeling methods exist [10], but the most common and widely-used is an extension to Dietz's numbering scheme (tree traversal order [11]) called extended preorder traversal [8]. Using this labeling method, each node in the tree is labeled with a pair of numbers <order, size>. This extension allows insertions to be made into the tree without the need for global reordering. It maintains the original idea of Dietz's scheme by imposing three conditions on the values for order and size.

1. For a tree node y and its parent x, order(x) < order(y) and order(y)+size(y) _ order(x)+ size(x). In other words, the interval [order(y), order(y)+size(y)] is contained in the interval [order(x), order(x) + size(x)].

2. For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, then order(x)+ size(x) < order(y).

3. For any node x, size(x) _Xy size(y) for all y's that are a direct child of x. By using an arbitrarily large integer for size(x), future insertions into the structure can be made without the need for global reordering.

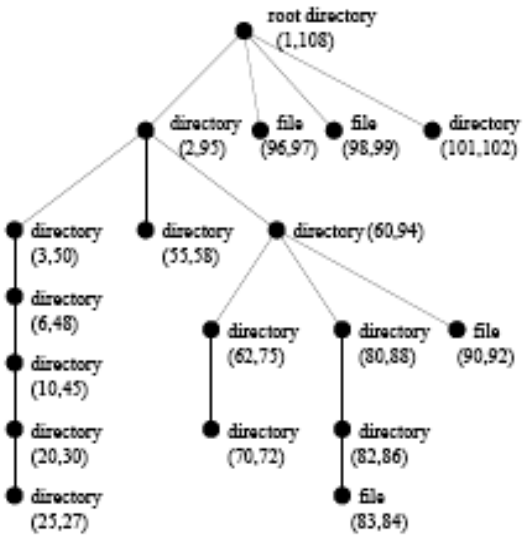

Figure 2: Example of an XML tree

B+-Tree

In relational database systems, the B+-tree (a variation of the B-tree) is used to implement a dynamic multilevel index [13]. Offering advantages to indexed sequential files, a B+-tree does not require reorganization of the entire file to maintain performance. In other words, the tree will automatically reorganize itself with small, local changes when insertions and deletions occur. Due to its hierarchical nature, the B+-tree was used in an algorithm for processing XML structural joins [14]. B+-trees have been an unqualified success in supporting external dynamic1-dimensional range searching in relational database systems [16], while R-trees [17] and XR-trees [15] are successful in indexing high-dimensional data points. Despite the increasing popularity of XML, to the best of our knowledge, we have seen no index structures that specifically deal with strictly nested XML data.

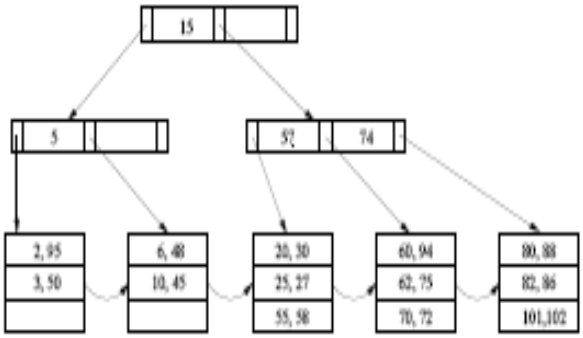

Figure 3: B+ -tree

XR-Tree

XR-tree (or XML Region Tree), a dynamic external memory index structure specially designed for XML data. Different from traditional B+ - trees, XR-trees index element nodes on their region codes, specifically (start, end) pairs. The novel feature of XR- tree is that, for any element E, all its ancestors (or descendants) in a given element set E indexed by an XR-tree can be retrieved with optimal O(log N + R) worst case I/O cost, where N is the size of E and R is the number of elements retrieved[18]. Such a unique feature of XR-tree makes it possible to most effectively skip both ancestors and descendants during a structural join if XR-trees are built on two joining element sets. The idea of XR-tree is motivated by an internal memory data structure: interval trees [19]. There are also works on interval management in external memory [20] and indexing time intervals [21]. XR-tree stands out among those proposed approaches in that it deals specifically with regions of XML elements while existing approaches manage arbitrary one-dimensional intervals.

The XR-tree [22], known as the XML Region Tree, is a B+-tree that is built on the start points of the element intervals. Designed for strictly nested XML data, this type of index structure allows all ancestors and descendants for a given element to be identified with optimal worst case disk input/output cost. The XR-tree outperforms the B+-tree for processing structural joins, but it lacks the capability to handle highly recursive XML elements with the same efficiency [23]. The structural relationship between two element nodes can be quickly determined by a region encoding scheme, where each element is assigned with a pair of numbers (start; end), based on its position in the data tree [28, 27,25], with the following held: for any two distinct elements u and v, (1) the region of u is completely before or after v, or (2) the region of u completely contains v or is contained by the region of v. Formally, element u is an ancestor of element v iff u:start < v:start and v:end < u:end. Since regions of two distinct elements never intersect partially, the formula can be simplified as u:start < v:start < u:end. Region codes for element nodes can be effectively generated by a depth-first traversal of the tree and sequentially assigning a number at each visit [28, 26].

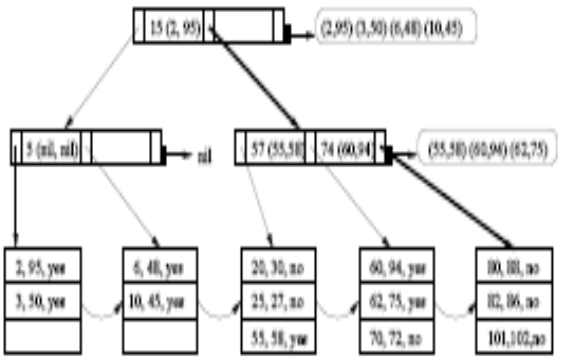

Figure 4: XR-Tree

XB-Tree

The XB-tree was developed by Bruno et al. [24] for use in processing holistic twig joins (a specialized version of structural joins). The XB-tree combines the structural features of both the B+-tree and the R-tree. It indexes the pre-assigned intervals of elements in the tree (similar to a one-dimensional R-tree) and then constructs the index on the start points of the intervals (similar to the standard B+-tree) [23]. The main difference is that the size portion of the <order, size> label must be propagated up the index. The main advantage of the XB-tree is that it quickly processes requests to find ancestors and descendants. A performance study [23] found that the XB-tree outperforms both the B+-tree and XR-tree for processing structural joins in XML documents. But, The find ancestors operation in the XB-tree is optimal when the intervals in the index nodes in the XB-tree are the minimum bounding intervals of the intervals in their child nodes. This is because every search path, except for the last search path, will yield at least one ancestor element that contains the given descendant element. Definition (Valid Path) [25].

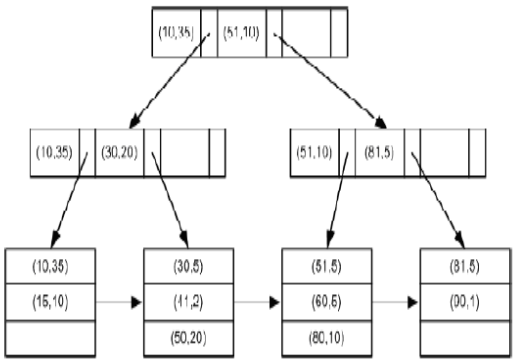

Figure 5: XB-Tree

Basic Search

The structural join operation requires two basic search: findDescendants and findAncestors. We will discuss how the three index-based solutions carry out these search procedures. We also investigate the conditions under which the findAncestors search in the XB-tree is optimal. The findDescendants procedure retrieves all the data entries that can be covered by an interval. The B+-tree, XR-tree, and XB-tree all utilize the efficient B+-tree range search to find matching descendant occurrences. Suppose we want to find all the files which are contained in a particular directory element. We first traverse down the various index trees for the file element by comparing the start points of the keys in the index nodes. Next, a sequential scan on the leaf nodes is carried out until we encounter a data entry whose interval lies beyond the given ancestor directory element interval. On the other hand, the findAncestors procedure looks for all the data entries that can cover an given interval. Consider Figure 2 again. Suppose we want to search for all the directory elements that contain the file element with the interval (90,92).

Table 1: A comparison between B+-tree, XR-tree, and XB-tree basic search

B+-tree	XR-tree	XB-tree
1- A sequential scan on the list.	1- Uses the B+-tree equality search.	1- Starts from the root node and descends to the leaf nodes.
2- Find the elements whose intervals contain the interval of the given file element.	2- The search key is the start point of the given descendant file element	2-Only the paths with the intervals that can cover the given interval would be searched
3- The search commences from the first element in the ancestor list, and ends when the start point of an ancestor element is greater than the start point of the descendant element.	3- During the search process, all the directory elements in the result set can be collected from the stab lists of the non-leaf nodes.	3- the data entries in the leaf nodes whose intervals can cover the given interval are output as results.
4- It is not effective for the low ancestor selectivity data.		4-findAncestors operation is optimal when the intervals in the index nodes in the XB-tree are the minimum bounding intervals of the intervals in their child nodes.

Discussion

The B+-tree has to perform a sequential scan on the list of directory elements to find the elements whose intervals contain the interval of the given file element. That is, the search commences from the first element in the ancestor list, and ends when the start point of an ancestor element is greater than the start point of the descendant element. Thus, the directory elements labeled with the intervals (2,95) and (60,94) would be retrieved (see Figure 3). Clearly, this solution is not effective for the low ancestor selectivity data. The XR-tree uses the B+-tree equality search to traverse down the index tree for the ancestor directory element. The search key is the start point of the given descendant file element. During the search process, all the directory elements in the result set can be collected from the stab lists of the non-leaf nodes, and finally from the leaf page. Consider the index tree for the directory element in Figure 4. Suppose we want to find all the directory elements which are the ancestors of the file element (90,92). Using 90 as the search key, we will retrieve the element intervals (2,95) and (60,94) from the stab lists of the non-leaf nodes in the search path.

All the elements in the result set will be obtained when we finally reach the leaf page (the first leaf page from the right). The search path is highlighted in Figure 4. To process the findAncestors queries, the XB-tree starts from the root node and descends to the leaf nodes in a manner that is similar to the R-tree, that is, only the paths with the intervals that can cover the given interval would be searched. Finally, the data entries in the leaf nodes whose intervals can cover the given interval are output as results. In the worst case, the findAncestors operation may need to search the entire XB-tree. The findAncestors operation in the XB-tree is optimal when the intervals in the index nodes in the XB-tree are the minimum bounding intervals of the intervals in their child nodes. This is because every search path, except for the last search path, will yield at least one ancestor element that contains the given descendant element.

Conclusion and Future Work

In this paper, we have analyzed the index structure technique of *B*+-tree, XB-tree and XR-tree for XML structural join. After applying the findAncestors algorithm, the XB-tree produced the optimal output for querying XML data. This paper mainly focused on processing of structural join, which is a core operation for XML query processing. Future work can be done on query evaluation strategies for complex XML queries (i.e. a combination of multiple structural joins) over XML data on which proper XB-tree indexes have been built.

Reference:

[1] Muralidhar Krishnaprasad, Zhen LiuTechniques of efficient XML meta-data query using XML table index, Correspondence Address: HCKMAN PALERMO TRUONG & BECKERAORACLE 2OSS GATEWAY PLACE SUTE 550 SAN JOSE, Oct 2024 CA 95110-1089 (US)

[2] Vikas Goel . An improvised indexing technique for XML data over multiple channels in wireless environment: (1, Xm) method, August 2019, International Journal of Communication Systems 32(3), DOI:10.1002/dac.4122

[3] Arroyuelo, D., F. Claude, S. Maneth, V. M¨Akinen, G. Navarro, K. Nguyen, J. Sir En and N. V Alim Aki, "Fast In-Memory XPath Search using Compressed Indexes", Software: Practice and Experience (Wiley), vol45, issue. 3, pp. 399-434, March. 2015. DOI: https://doi.org/10.1002/spe.2227

[3] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for XML data. In ACM SIGMOD, June 2002.

[4] B. F. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semi-structured data. In VLDB, pages 341–350, September 2001.

[6] R. Goldman and J. Widom. DataGuides: Enable query formulation and optimization in semi-structured databases. In VLDB, pages 436–445, August 1997.

[7] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In ACM SIGMOD, June 2002.

[8] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01), pages 361–370, San Francisco, CA, United States, September 11-14 -2001

[9] T. Milo and D. Suciu. Index structures for path expression. In Proceedings of 7th International Conference on Database Theory (ICDT), pages 277–295, January 1999. California, San Diego, 1999.

[7 Su Cheng Haw and G. S. V. Radha Krishna Rao. Query optimization techniques for XML databases. International Journal of Information Technology, 2(1):97–104, 2005. 29.

[11] Paul F. Dietz. Maintaining order in a linked list. In Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC'82), pages 122–127, San Francisco, CA, United States, May 5-7 1982.

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

[13] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database Systems. Addison-Wesley, 3rd edition, 2000.

[14] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, and Vassilis J. Tsotras. Efficient structural joins on indexed xml documents. In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02), pages 263–274, Hong Kong, China, August 20-23 2002.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In SIGMOD, pages 322–331, 1990.

[16] D. Comer. The ubiquitous B-Tree. ACM Computing Surveys, 11(2):121–137, 1979.

[17] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In SIGMOD, pages 47–57, 1984.

[18] Haifeng Jiang, Hongjun Lu, Wei Wang, Beng Chin Ooi: XR-Tree: Indexing XML Data for Efficient Structural Joins: Proceedings of the 19th International Conference on Data Engineering (ICDE'03) 1063-6382/03 $ 17.00 © 2003 IEEE

[19] M. D. Berg, M. V. Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag, Berlin, Germany, 1997.

[20] L. Arge and J. S. Vitter. Optimal dynamic interval manage- ment in external memory. In FOCS, pages 560–569, 1996

[21] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. In VLDB, pages 1–12,1990.

.[22] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XMLdata for efficient structural joins. In Proceedings of the 19th Internati onal Conference on Data Engineering (ICDE'03), pages 253—263, Bangalore, India, March 5-8 2003.

[23] Hanyu Li, Mong-Li Lee, Wynne Hsu, and Chao Chen. An evaluation of xml indexes for structural join. SIGMOD Record, 33(3):28—33, September 2004.

[24] M. Zhang, J.T. Yao. The XML Algebra for Data Mining. In the

[25]Hanyu Li. Mong Li Lee. Wynne Hsu. Chao Chen. An Evaluation of XML Indexes for Structural Join . ACM 2002.

[26] S-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In VLDB, 2002.

[27] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In ICDE, 2003.

[28] S.-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In VLDB, pages 263.274, 2002.

[29] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In VLDB, pages 361.370, 2001.

[30] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In ICDE, pages 141. 152, 2002.

[31] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In SIGMOD, pages 425.436, 2001.

[32] S.-Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In VLDB, pages 263.274, 2002.

[33] IBM Corporation. XML data generator. http://www.alphaworks.ibm.com/tech/xmlgenerato.

[34] P. F. Dietz. Maintaining order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122.127, 1982.

[35] T. Grust. Accelerating XPath location steps. In SIGMOD, pages 109.120, 2002.

[36] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In SIGMOD, pages 47.57, 1984.