Efficient Structural Joins with On-The-Fly Indexing

Kun-Lung Wu, Shyh-Kwei Chen and Philip S. Yu
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{klwu, skchen, psyu}@us.ibm.com

ABSTRACT

Previous work on structural joins mostly focuses on maintaining offline indexes on disks. Most of them also require the elements in both sets to be sorted. In this paper, we study an *on-the-fly*, in-memory indexing approach to structural joins. There is no need to sort the elements or maintain indexes on disks. We identify the similarity between the structural join problem and the stabbing query problem, and extend a main memory-based indexing technique for stabbing queries to structural joins.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Storage and Retrieval—*Information Search and Retrieval*

General Terms

Algorithms

Keywords

Structural Joins, Containment Queries, XML

1. INTRODUCTION

Structural joins have been identified as important operations for processing containment queries [1, 2, 3, 5]. A structural join is a set-at-a-time operation that finds all the ancestor-descendant relationships between two node elements 'in an XML document.

To process structural joins, each node element is typically labeled with a pair of numbers: (*start, end*). These two numbers are usually integers and represent the start and end positions of the element in the document tree [3, 7]. Namely, the interval encodes the region of the node element. With region-encoded intervals, a structural join can be formally defined as follows. Given two input lists, A and D, where A contains intervals representing ancestor node elements and D contains intervals representing descendant node elements, a structural join is to report all pairs (a, d), where $a \in A$ and $d \in D$, such that interval a contains interval d. Most of the previous work on structural joins assumes (i) offline indexes are maintained on disks for both input sets or (ii) the elements in both input sets are sorted or (iii) both.

In this paper, we study an on-the-fly indexing approach to structural joins. This is in contrast to prior offline indexing [2, 3] and non-indexed [1] approaches to structural joins. In order for on-the-fly indexing to be effective, the index storage cost must be low, and the index construction time and the index search time must be fast. Low index storage cost makes it possible to maintain the entire index in memory, avoiding degradation in structural join due to index I/O cost. Fast index construction and search makes structural joins efficient.

We extend a *containment-encoded interval* (CEI) indexing to perform structural joins, referred to as CEI indexing for structural joins, or CEI-SJ. CEI was originally proposed to index continual range queries, represented as intervals, for efficient stream processing [6]. It has low storage cost and fast insertion and search performance. It efficiently solves the *stabbing query* problem [4], which is to find all the intervals that are stabbed by any data point. We refer the original scheme as CEI indexing for stabbing query, or CEI-SQ. The insertion and search algorithms of CEI-SQ are simple and easy to implement in practice.

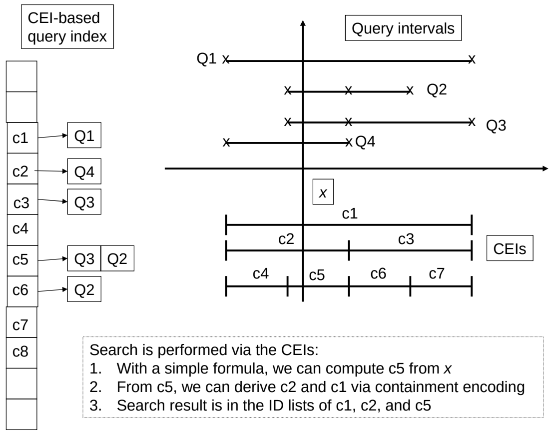
2. CEI INDEX FOR STABBING QUERY

The idea of CEI indexing centers around a set of predefined containment-coded intervals, called CEI's. These CEI's are virtual construct intervals used to decompose query intervals and store the IDs of the query intervals that use them in the decomposition. Data values in the stream are then used to search the CEI index. The containment relationships embedded in the CEI's makes insertion and search operations efficient.

Fig. 1 shows an example of CEI-SQ. It shows the decomposition of four query intervals: $Q1, Q2, Q3$ and $Q4$ within a specific segment containing CEI's of $c1, \cdots, c7$. CEI $c1$ contains $c2$ and $c3$; $c2 = 2 * c1$ and $c3 = 2 * c1 + 1$. $Q1$ completely covers the segment, and its ID is inserted into $c1$. $Q2$ lies within the segment and is decomposed into $c5$ and $c6$, the largest CEI's that can be used. $Q3$ also resides within the segment, but its right endpoint coincides with a guiding post. As a result, we can use $c3$, instead of $c7$ and $c8$ for decomposition. Similarly, $c2$ is used to decompose $Q4$. As shown in Fig. 1, query IDs are inserted into the ID lists associated with the decomposed CEI's.

The search algorithm is simple and efficient [6]. As an example, to search with a data value x in Fig. 1, the local ID of the unit-length CEI that contains it is first computed. In this case it is $c5$. Then, from $c5$, the local IDs of all its ancestors that contain $c5$ can be efficiently computed via

WWW 2005, May 10–14, 2005, Chiba, Japan.
ACM 1-59593-051-5/05/0005.



Search is performed via the CEIs:
1. With a simple formula, we can compute c5 from *x*
2. From c5, we can derive c2 and c1 via containment encoding
3. Search result is in the ID lists of c1, c2, and c5

Figure 1: Example of CEI-SQ.

containment encoding. Namely, the parent of a CEI with local ID l can be computed by $\lfloor l/2 \rfloor$, i.e., a logical right shift by 1 bit of l. In this case, they are $c2$ and $c1$. As a result, the search result is contained in the 3 ID lists associated with $c1, c2$ and $c5$. We can verify from Fig. 1 that the result indeed contains $Q1, Q2, Q3$ and $Q4$.

3. CEI INDEX FOR STRUCTURAL JOINS

With each node element encoded with a pair of integers, (*start, end*), the structural relationship between two elements can be easily determined [1, 2, 3, 7]. For any two distinct elements u and v in a tree-structured document, the following holds [3]: (1) The region of u is either completely before or completely after that of v; or (2) the region of u either contains that of v or is contained by the region of v. In other words, if there is any overlap between two intervals, the overlap is complete containment. In other words, two intervals never partially overlap with each other.

With this complete containment property between any two node elements, the problem of structural joins of two sets of intervals can be transformed into one that searches the CEI index of the ancestor intervals with the start, or end, points of the descendant intervals. In this transformation, we treat the ancestor elements as interval queries and the start, or end, points of all the descendant intervals as the data points as described in Section 2. At runtime, a CEI index is constructed on-demand for the ancestor intervals. Then, the start points of all the descendant intervals are used to search the CEI index. There is no need to sort the elements or maintain any indexes on disks.

Fig. 2 shows an example of structural joins viewed as stabbing the ancestor intervals with descendant start points. We draw each element interval as a horizontal line segment. Let A_d be the set of interval IDs stabbed by the vertical line at the start point of a descendant interval d. Because there is no partial overlapping between any two elements, each $a \in A_d$ must completely contain d. Since the result of searching the CEI index with the start point of a descendant interval d contains all the ancestor intervals that cover the point, such a search operation generates all the join output pairs involving d. Similar arguments can be made regarding the end point of a descendant interval.

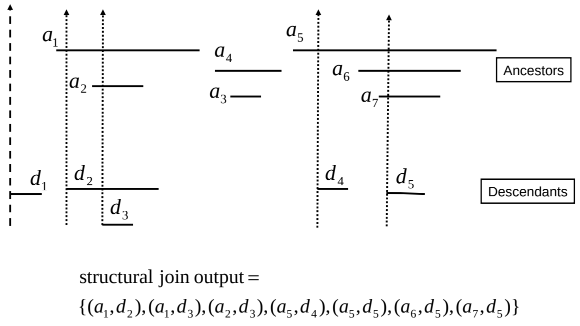


Figure 2: Structural joins viewed as stabbing ancestor intervals with descendant start points.

CEI-SJ can be further optimized. Because node elements are encoded with a pair of integers, representing the start and end positions of the element in the document, no two elements can share the same endpoint and the minimal length of an interval is 1. As a result, the start point of a descendant element would never stab at the portion of an ancestor element that corresponds to a unit-length CEI. Hence, unit-length CEI's can be eliminated.

4. SUMMARY

CEI-SJ is an on-the-fly, main memory-based indexing approach. Centering around a set of predefined virtual construct intervals whose IDs are encoded with containment relationships, CEI indexing has very fast insertion and search performance. Taking advantage of these desirable properties, our approach constructs on-the-fly a single in-memory CEI index on the ancestor set. Structural joins are efficiently carried out by searching the CEI index with the start (or end) points of the descendant elements. Simulations show that CEI-SJ substantially outperforms prior approaches.

5. REFERENCES

[1] S. Al-Khlifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of IEEE ICDE*, 2002.

[2] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, 2002.

[3] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proc. of IEEE ICDE*, 2003.

[4] H. Samet. *Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[5] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. In *Proc. of IEEE ICDE*, 2003.

[6] K.-L. Wu, S.-K. Chen, and P. S. Yu. Interval query indexing for efficient stream processing. In *Proc. of ACM CIKM*, Nov. 2004.

[7] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in Relational database management systems. In *Proc. of ACM SIGMOD*, 2001.