

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 $c1, \dots, c7$. 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

