CS179g: Project in Computer Science: Databases

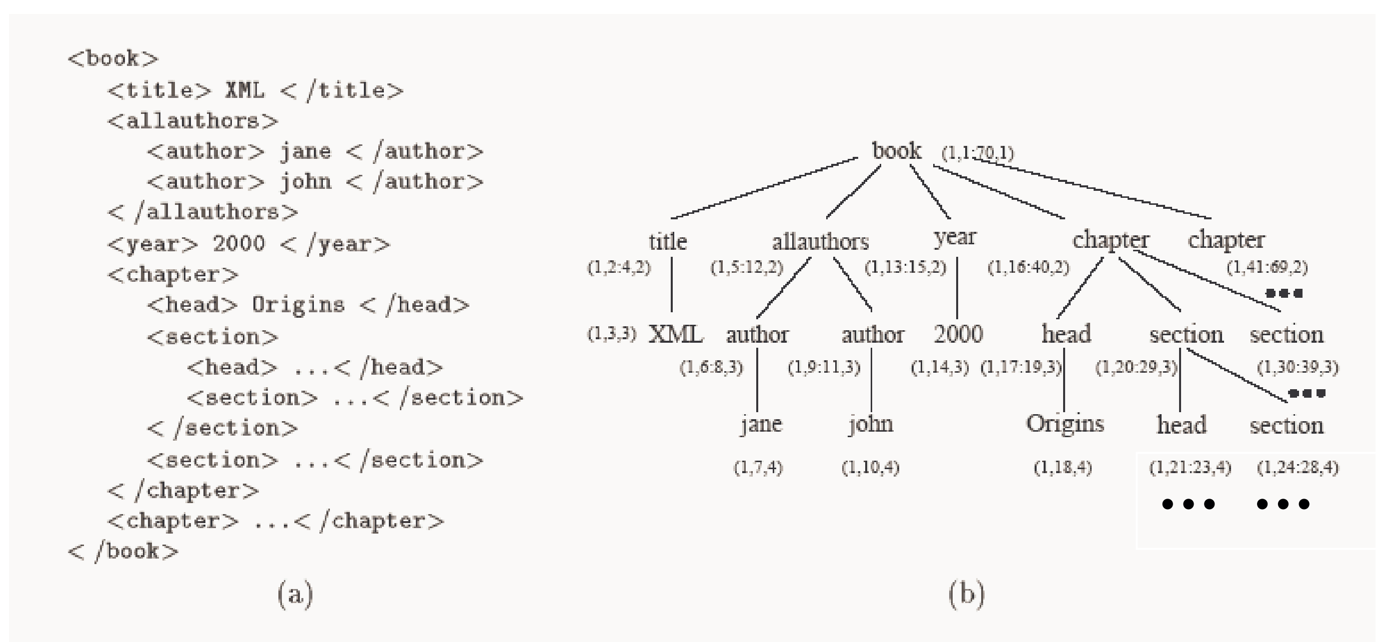
Department of Computer Science & Engineering
University of California - Riverside
Instructor: Vassilis Tsotras

1. Project Goal

The aim of the project is to make students familiar with XML data and XML data management from a DBMS perspective. The goal is to create a multi-version archiving system for XML documents.

2. Introduction

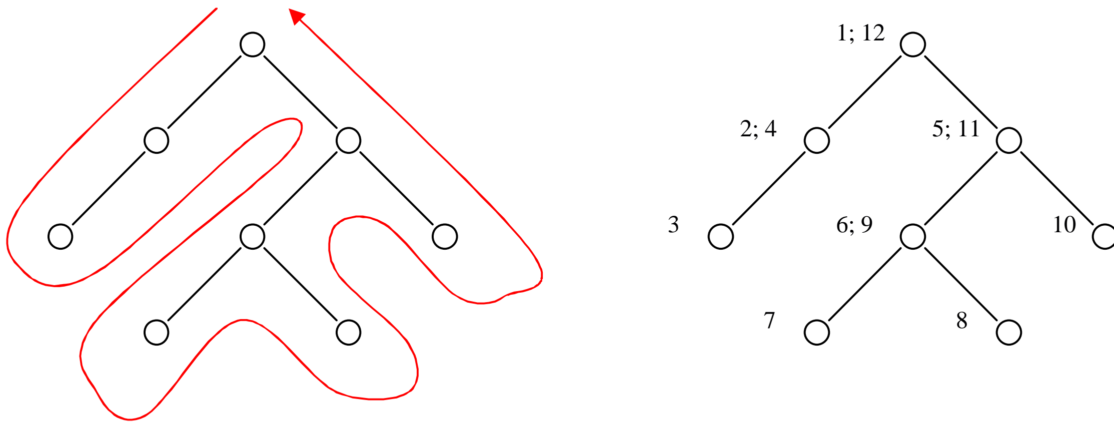
An XML database is a forest of rooted oriented labeled trees where each node is corresponding to an element and the edges represent element-subelement relationships. Node labels consist of set of attributes (attribute, value) pairs, which suffices for modeling tags, PCDATA content etc. Sibling nodes (children of the same parent node) are ordered. The ordering of sibling nodes implicitly defines a total order on the nodes in the tree obtained by the depth-first, left first traversal of the tree nodes. An example of an XML document appears below: (a) the document, (b) its tree representation (the numbers next to the nodes will be discussed shortly).



XML query languages like XQuery, make fundamental use of tree patterns for matching relevant portions of data in the XML database [3]. The query pattern node labels include element tags, attribute value comparisons and string values while the query pattern edges are typically parent-child edges or ancestor descendant edges. Path expressions are the building components for such queries and are thus at the core of the XML query processing. A path expression defines a series of XML tree node labels every two of which are related with ancestor –descendant or parent – child relationship. A path-expression can be thought as a collection of pair joins, called structural joins [1,2]. An example structural join in the above XML tree, is: “find all *head* elements that are under *chapter* elements”

2.1 Tree Numbering

There have been various research proposals recently for efficiently processing structural-join queries. One of the most popular approaches is to assign a numbering scheme on the XML tree so as to capture the structural characteristics of the tree nodes [1,2]. One such scheme is the range-based numbering. In this scheme, each node in the document tree is assigned a range of two numbers (*left*, *right*). The numbering is assigned through a depth-first traversal of the document. The *left* number is assigned when we first meet the node in the traversal down the tree while the *right* number is set when the node is reached for second time on the way up to the tree root. Since the leaf nodes in the XML tree do not have children they are assigned a single number instead of a range.



Given a node pair *(a,d)* node *d* is *descendant* of node *a* (while *a* is then an *ancestor* of *d*) if and only if

$$a.left < d.left < d.right < a.right$$

In the previous example node 2:4 is descendant on node 1:12 but is not descendant on node 5:11. Any two node ranges can be disjoint or one of them fully covers the other. Hence left numbers can be used to identify nodes in a static tree uniquely. The same is also true about the right value. To capture *parent – child* relationships, the tree level is also assigned to the nodes (a node *d* is then a child of node *a*, if *d* is descendant of *a* and *d*'s level number is one more than the level of *a*).

Assume that the elements of the XML document are organized in lists by tag name. For example, in the XML document in the previous page we would have separate lists for the tags *chapter, section, author*, etc. Assume that each tag list is ordered by the *left* position of its corresponding elements. Then a structural join query between two tags (say: *chapter//section*) would look into the elements of the two lists that participate in the query and try to identify *section* elements that are descendants of *chapter* elements (using the condition above). That is, the query can be solved by a single pass over the two lists (like a merge-join) assuming of course the lists are ordered.

2.2 Version management

In the simplest version management scheme a new version is created out of the currently last version. This results in a single line of consecutive versions; also called linear versioning (In practice most design applications follow the branched versioning scheme, where a new version can be delivered from any of the previous ones, thus resulting in a tree of versions. For the purposes of this project however a linear versioning scheme will be considered.)

In order to maintain all past versions of a given XML document, we need to be able to record the version lifetime for each element in the tree. Each element node in the document tree is thus stored as a record that contains:

- The node tag
- The left and right numbers (for ancestor/descendant relationships)
- The level of the node (for parent/child relationships)
- The version lifetime of the record (Vstart, Vend)
- A pointer to the content that the node had during its lifetime

When a node is created by a given version, say Vi, its Vstart is initialized to Vi (the Vend is empty). If at a later version Vj, this node is deleted, its Vend is updated to Vj. That is, node deletions in the XML tree are not physical deletions. Rather, we update the Vend for the node's record with the version that deleted this node. We do not delete records because they need to answer queries about previous versions of the document. We call a record "alive" for all versions within its version lifetime interval.

The next issue is how to store the element records so as to efficiently answer the two queries the project aims for. There are various ways to cluster the versions of a document, depending on which query the user needs to optimize. Consider first the structural join queries. As mentioned above, it is assumed that the document nodes are organized in tag lists. As the document evolves through subsequent versions, elements need to be added/updated in these lists.

Let's consider adding the elements in a tag list for version 1. These elements are stored sequentially in disk pages, in left position order. However, in subsequent versions new elements may be added, and elements may be updated as deleted. We call a disk page *useful* for all versions for which this page contains at least *U* alive records. When some page *p* is first created (by some version *Vc*) all its records are alive. Later, however, due to node deletions, the number of alive records in this page decreases. If it falls below *U*, page *p* is called *useless*. Assume, that a deletion during version *Vd* caused the number of alive elements in *p* to fall below the threshold. Then page *p* is usefull for all versions in the interval *(Vc, Vd)*. This means, that all versions within the page's usefulness interval can access this page and find enough records alive during that version.

Why this is important? Because, it allows for clustering the records in few pages. The problem of finding the alive records of a given tag list as of version Vi is then transformed to finding the useful pages during version Vi. Note that without version clustering, we may end up accessing a page (spend a whole I/O) and find even a single record. Consider an example where there where 100 alive elements for the version in question. Without clustering we may end up accessing 100 pages (too much I/O).

4 Project tasks

You will have to:

1. Find a way to adapt the numbering scheme to the case where there are deletes and updates in the XML tree structure. It is necessary to find a way to avoid re-numbering the tree if node deletions/additions occur.
2. Implement the XML versioning system in such a way that it is able to <u>efficiently</u>:
    - Give the whole document as of version Vi.(For example retrieve version 2 of the document)
    - Answer structural join queries for a given version of the document. (For example – for version 2 retrieve all chapters which belong to a given book)
3. Implement version clustering.
4. Perform evaluations and experiments and discuss how the value of coefficient *U* affects the system's performance.

By efficient implementation we mean an implementation that uses space proportional to the "log" approach while it has query performance like the "snapshot" approach discussed in class.

5 Resources

The project should be implemented in C++. The database system used for this project is Berkley DB. A reference is available at http://www.sleepycat.com/

We will also provide a C++ program for accessing/manipulating the disk pages in this system.

6 Related Papers

[1] Q. Li, B. Moon, “Indexing and Querying XML Data for Regular Path Expressions”, Proc. of VLDB Conference, 2001, pp: 361-370.
[2] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu, “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”, Proc. of ICDE Conf. 2002, pp: 141-154..
[3] N. Bruno, N. Koudas, D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching", Proc. of ACM of SIGMOD Conf. 2002, pp: 310-321.
[4] S-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, “Efficient Structural Joins on Indexed XML Documents”, Proc. of VLDB Conf. 2002, pp: 263-274.
[5] V.J. Tsotras, N. Kangerlaris, “The Snapshot Index: An I/O-optimal access method for timeslice queries”, Information Systems Journal, 20(3): 237-260 (1995)

7 References

BerkeleyDB, Available at http://www.sleepycat.com/