Project
XML Indexing Scheme

1 Overview of the XML Indexing Scheme

In this project, you will have to implement a specific scheme for indexing incoming XML data. This scheme, loosely based on the architecture of XISS [1].

The indexing scheme described here is designed with the following tasks in mind:

1. Indexing of an entire XML document from a DOM tree, or a SAX stream.
2. Retrieval of XML elements by element name.
3. Retrieval of XML elements by relationship (ancestor, descendant, parent, child, etc. . .).
4. Determination of relationship between two elements.
5. Retrieval of content of an element.
6. Retrieval of attributes of an element.
7. Generation of XML for an element.

Except for the last task, the index structures described below can address the tasks specified above in a straightforward manner (XML generation requires a somewhat more involved procedure).

[1] Quanzhong Li, Bongki Moon, Indexing and Querying XML Data for Regular Path Expressions, in *Proc., International Conference on Very Large Databases (VLDB)*, 2001, pp. 361 – 370, http://www.vldb.org/conf/2001/P361.pdf

Simplyfying Assumptions and Constraints

Storage of XML-related data verbatim would lead to the need for variable-length records in a number of different places. XISS avoids it by storing all text data (element names, attribute names, content, values) in a special *names* index, and using fixed-length records elsewhere.

Our approach to data storage is somewhat different. To facilitate it, we use the following simplifying assumptions:

1. **Size of element names.** The length of an element name is less than or equal to **16 bytes**.
2. **Size of element content.** The length of the content of a leaf node is less than the size of a disk block. I.e., any content record will fit onto a single disk page.
3. **Size of attribute names.** The length of an attribute name is less than or equal to **16 bytes**.
4. **Size of attribute values.** The length of an attribute value is less than or equal to **64 bytes**.
5. **Number of unique element names.** You can assume that the list of unique element names can be stored on a single disk block.
6. **Uniqueness of attributes.** For each element node, only one occurrence of a specfic attribute is allowed (this is a *well-formedness* of XML requirement, but it makes sense to stress it here).
7. **Record Addresses.** Because you will use NEUStore as the back-end for your code, we can adopt a simple procedure for referencing data stored on disk. Each record (in a data file, or an index file) can be characterized using two parameters.

   The first addressing schema, used most commonly is (PageId, RecordNumber), where PageId is the page Id stored in the first four bytes of the NEUStore disk block, and RecordNumber is the slot on the page in which the desired record resides.

   The second schema is (PageId, Offset). Here, PageId is as above and Offset is the byte position at which the record/data start on the page. This can be used to address variable length records and content.

Index Structures Overview

The XML indexing scheme you will be implementing consists of four structures used to store all data from the XML document. These structures are:

ElementIndex indexes XML elements in a document by name. The internal organization of this structure is left for the groups to decide. Externally, this structure, given an element name should be able to find all nodes with this name in the **StructureIndex** structure, described below.

StructureIndex stores information about the XML nodes in the document. Each node is represented as a record, storing all necessary indexing information (see specifications below). The default assumption is that the nodes in this index are stored in the depth-first-search order of traversal of the DOM tree of the XML document (a.k.a, in the order of appearance of their opening tags in SAX stream).

Content stores all text content of the XML document in concatenated form. **StructureIndex** records will point to specific records in the **Content** index. Content is the **only** *variable-length record* index structure that you have to implement in this project.

AttributeIndex stores all information about attributes. The data in this structure should be indexed by the node information (each attribute belongs to a specific node).

The indexing schema is shown in Figure 1

We provide details about each structure below.

ElementIndex

ElementIndex structure is responsible for (a) *specifying a unique integer id* for each unique element name in the XML document, (b) serving as a secondary index over **StructureIndex**, based on element name/element id. This can be implemented in a number of different ways. A non-exhaustive list of ways is below:

- Use a hash-table approach to store the data. Element names are hashed (and in addition to it are assigned unique ids), and information about locations of all elements with a certain hash key is stored in the hash table buckets and overflow pages as needed.
- Index element names lexicographically. Information on occurrences of each element is stored in a single page (with overflow pages if needed), the header page contains the list of element names and pointers to the pages.
- Store element names on first-come-first serve basis (given that the element name table fits a single disk page, a scan of this table is a cheap operation), use dense index to store pointers to occurrences of elements.

The indexing portion of this structure must contain records that have the following structure:

```
(ElementName, ElementId, PageId);
```

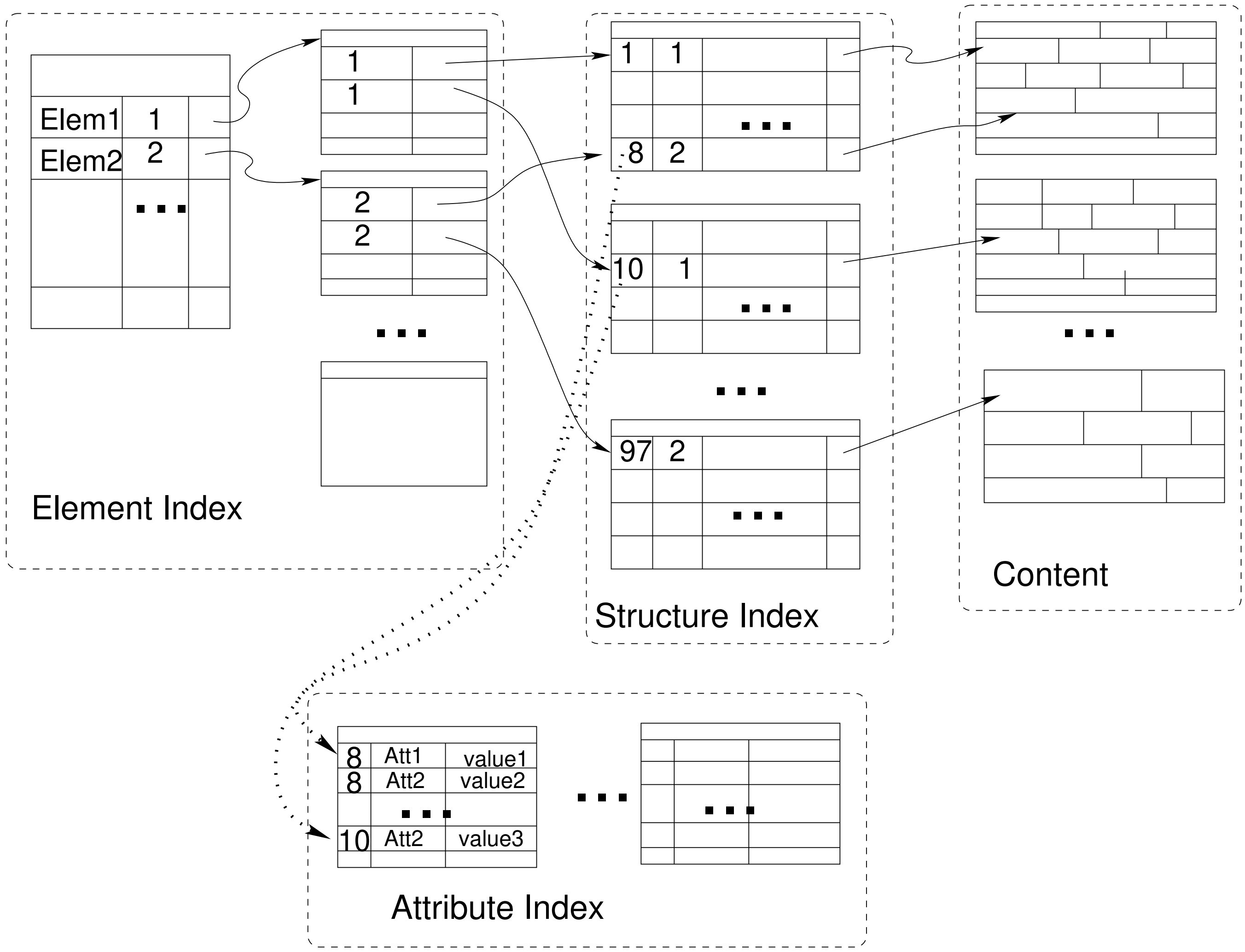


Figure 1: The XML indexing schema for the project.

with an optional RecordNumber attribute (needed for dense indexes, but not for hash-tables).

The records specifying locations in the **StructureIndex** should have the following structure:

(ElementId, PageId, RecordNumber).

Here (PageId,RecordNumber) pair is an address in the **StructureIndex** structure.

StructureIndex

StructureIndex is the *data file* storing information about all element nodes in the XML document. This is the core index structure in this schema, as it links to or is linked from all other structures.

The **StructureIndex** stores one record per element node in the XML document. The record structure is as follows:

StructureIndex(NodeId, ElementId, PreOrder, PostOrder, Ordinal, Layer, Parent, IsLeaf, Content).

We explain each attribute in turn:

- NodeId: unique Id of the node, primary key of **StructureIndex**. Integers can be used.
- ElementId: the Id of the element name (same as in the **ElementIndex** structure).
- PreOrder, PostOrder. The key indexing parameters for each node.

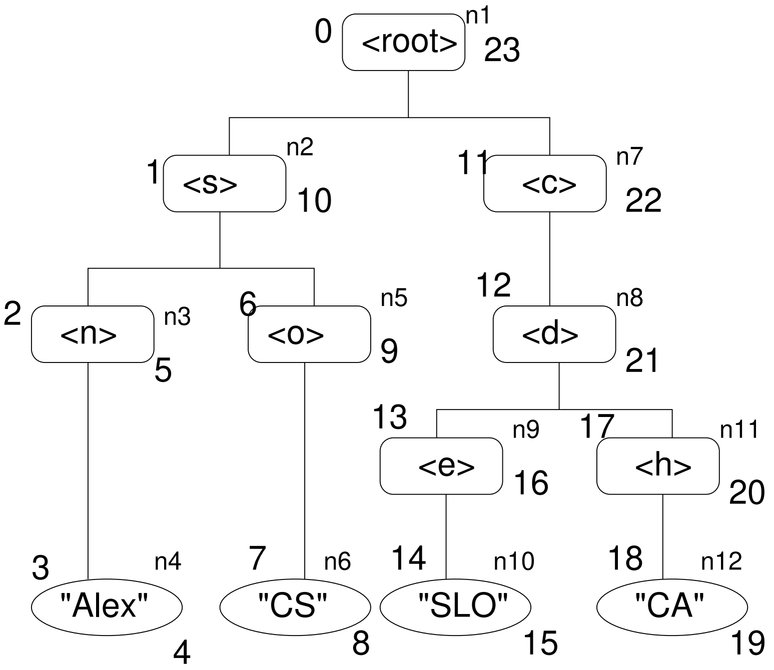
  Given an XML document, the PreOrder and PostOrder numbers for each XML element in it are defined as follows:

  - The XML document (in a from of an XML/DOM tree) is traversed in depth-first search (DFS) order.
  - A counter is kept. It is set to 0, when the traversal starts at the root.
  - The counter is incremented by 1 *each time DFS traversal enters a new node, and each time DFS traversal is about to leave a node forever (i.e., upon completely exploring its subtree).*
  - For each XML element node, the PreOrder value is the value of the Counter when DFS traversal enters the node.
  - For each XML element node, the PostOrder value is the value of the Counter when DFS traversal leaves the node forever.

  For example, consider the following XML document:

```
<root>
   <s>
      <n>Alex</n>
      <o>CS</o>
    </s>
    <c>
       <d> <e>SLO</e>
           <h>CA</h>
       </d>
    </c>
</root>
```

The figure below shows the PreOrder and PostOrder for each XML node in the document.



The PreOrder values are shown above and to the left of each node. The PostOrder values are shown below and to the right of each node. Above and to the right of each node is the node's unique id, which follows the DFS traversal order in the XML tree.

- Ordinal. The position of the current node in the list of its siblings. For the first child, Ordinal=1, and so on.
- Layer. The depth of the node in the DOM tree. You can start with Layer=0 for the root node, or Layer=1 for the root node, but you have to be consistent in the further assignments of depth.
- Parent. Pointer to the parent node. Can be just the NodeId of the parent. Can also be a triple (ParentNodeId, ParentPage, ParentRecNo)

which contains the unique identifier of the parent node, and its location in the StructureIndex.

- IsLeaf: boolean value, **true** if current node has no children (its only content is text), false otherwise.
- Content. The address in the **Content** data structure of the start of the content (scope) of the current node. The most straightforward way to represent is a (ContentPage, ContentRecNo) or (ContentPage,ContentOffset) pair. The specifics would depend on how you choose to implement the Content index structure.

In general, it is useful to keep **StructureIndex** sequential. The default order is the order of appearance of start tags in the XML document, which also happens to be the DFS traversal order. This order is easily constructed both via DOM tree traversal, and through SAX stream analysis.

Content

The Content structure will store the content of the XML document. Because the length of the content string for each leaf node is limited only by our simplifying assumptions (it is less than the size of one disk page[2]), different leaf nodes in an XML document will have content of different length.

The most straightforward way to deal with this, is to make Content a variable-length record index. Each record must contain only one field: the content itself, although you may elect to add more information to the record if it suits your purposes.

AttributeIndex

The AttributeIndex data structure stores, and indexes the XML attributes and their values. The restrictions on attribute name and attribute value sizes mean that the attributes can be stored in fixed-length records. The structure of the records is as follows:

AttributeIndex(NodeId, AttributeName, AttributeValue).

Here, NodeId is the node id (same as in StructureIndex) of the XML node, containing the attribute, AttributeName and AttributeValue are self-explanatory.

The pair NodeId, AttributeName forms the primary key for this data.

It is useful to store **AttributeIndex** records sequentially, ordered by NodeId.

You will also need to build an index structure on top of this storage. The choice of the structure is up to you. A B+tree for a dense/sparse index, or a simle dense/sparse index would work. Alternatively, you may also change the structure

[2]To be more exact, it should be less than the amount of free space on a single disk block - i.e., the size of the disk block minus the size of the block header,

of the **StructureIndex** records to point to the first appropriate attribute record in **AttributeIndex**.

EXample

Consider the following XML document d:

```
<root>
   <a x="123"><b top="2">This is</b><b> a test.</b></a>
   <name>John</name>
   <age>25</age>
</root>
```

This document will be indexed as shown in Figure 2. In this example, we assumed the following:

- All PageIds (top left corner of each page) are numbered from 0 and up. Essentially, it suggests that all these structures are stored in a single file. (Generally speaking this does not have to be the case, but it makes our example simple).
- The **ElementIndex** uses a simple lookup table for element names, filled in the order of occurrence of first instance of an element in the document. Each element name gets its "bucket" (pages 1 – 5) and the strucutre of the record on the bucket is (ElementId, PageId, RecordId). Note that ElementId is redundant here - it is enough to store it once on the page.
- **StructureIndex** is a sequential file with no extra indexes defined on top of it. Each page contains four full records, record structure as described above in this handout. Layers are counted from 0, Parent=-1 means "root", i.e., no parent, and root also gets the ordinal of 0.
- **Content** is shown approximately. In reality, there are no breaks between records in this structure.
- **AttributeIndex** is a dense index on a sequential file, sorted by the NodeId.
- Arrows on the diagram show where the address stored in the record is pointing to.

Modifications to the Basic Design

The above description of the four structures forming the XML indexing schema is sufficient to store and access information about XML. You may, however, choose to make some modifications to this design, and expand it, according to what you find convenient.

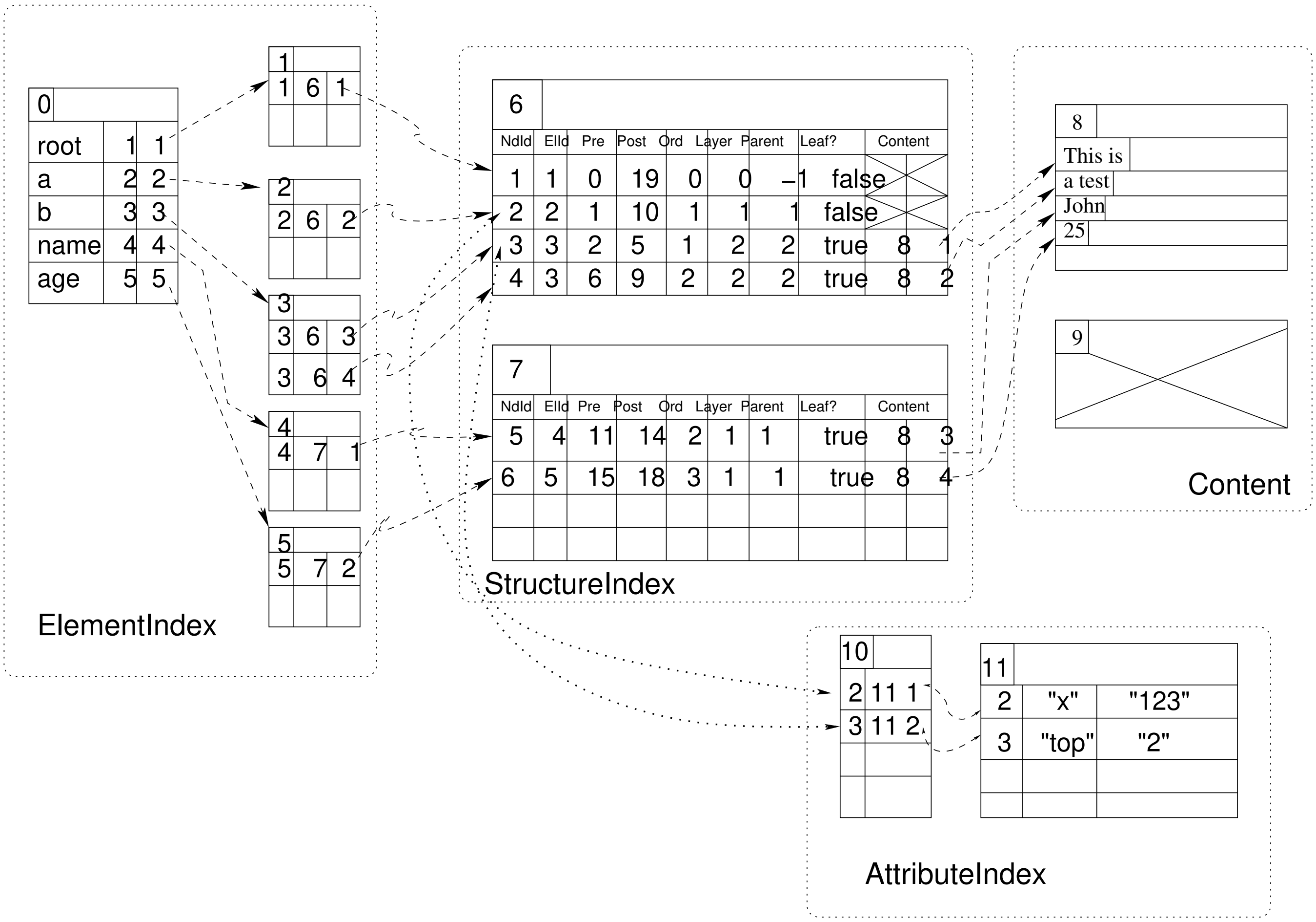


Figure 2: Indexing XML documents.

Some decisions discussed below are at a relatively low level (add another attribute to the record), some others - at a higher level (use a specific index structure).

- **Disk Page Size.** You have certain degree of freedom when deciding on the sizes of your disk pages. The key restricting assumption is that the list of all XML elements (to be more exact, the records indexing XML element names) can be stored in a single disk block. Your page sizes should not be outrageously large (no more than 16K), and should not be too small. I suggest considering 2Kb, or 4Kb pages.
- **File Headers.** The design of File header pages is left up to you. Note that there is certain information that NEUStore stores in File Header pages, you need to ensure you do not overwrite it.
- **Block Headers.** The design of block headers for all page types is left to you. Note that NEUStore stores PageId and PageType in the first 8 bytes of the page. Feel free to design the page header to your liking. Note that it may be useful to keep some space in the page header unoccupied at the beginning stage of your design. You may need to add more information to the headers later, during the second/third stages of the project.
- **Record Structure.** The design of each individual record structure given to you should be treated as an outline. The information above states what I think *must* be present in the record. You may choose to add more information to the record, and/or omit some of the fields I used. If you are omitting the fields, please make sure that it does not damage the functionality of the index structure.

  Examples of possible changes to record structure may include: (a) adding dense index pointers from **StructureIndex** to **AttributeIndex**; (b) adding pointer to the end of the content/scope to the StructureIndex, etc....
- **Index Structures.** The choice of specific ways of indexing element names, attributes and XML tree nodes is left up to you. It is pretty clear that some sort of indexing has to take place for element names and for attributes, because we should be able to quickly answer the following questions:
  - given a name of an XML element, find all occurrences of this element in the XML document.
  - given an XML node (element node), find all its attributes.

  Less evident is the need to build extra indexes on StructureIndex. You may, however, decide that you want to build a dense or sparse index on StructureIndex on its main search key (PreOrder,PostOrder pair). In this case, you may implement either simple or B+tree index.

  Similarly, you may want to build a secondary index on AttributeIndex, indexing AttributeName. This, again, is not necessary, and is left up to you.
- **Index Structure Implementations.** As mentioned above, if you choose a specific index for your data, you can also choose whether to use simple indexing techniques, B+-trees, or hash tables, where appropriate.