Project Stage 1.5: XML Repository Management

Due date: Tuesday, October 29, 11:59pm (soft); Tuesday, November 5, 8:00am (hard).

Note: Stage 1 of the project consists of two tiers: Stage 1.0 and Stage 1.5. **Both** are due October 29. This document details Stage 1.5, the second half of the assignment.

Note 2: The soft deadline is determined by the official start of Stage 2. During the week between the soft and the hard deadline, we will schedule demos with each team (see **Deliverables and Submission** section).

Overview

Stage 1.0 of ArferDB incorporated three layers: Buffer Manager, Buffer Access (a.k.a. Page Read/Write), and DB file manager.

The DB file manager layer of ArferDB implements two types of generic database data files: sequential files with multiple page types (each page type forms its own sequence with its own key) and hashed files. For each file type, your implementation is responsible for the following operations:

- `create new data file`
- `delete data file from disk`
- `create new data page`
- `delete a data page`
- `record insertion`
- `record deletion`
- `record update`
- `record lookup by key` (direct record lookup)

Stage 1.5 builds upon the data file layer and creates the backbone of ArferDB. In particular, on Stage 1.5 you will implement the following functionality:

- Creation of a new XML repository.

- Deletion of an existing XML repository.
- Insertion of an XML document into an existing XML repository.

For the sake of completeness, we replicate some definitions from the Stage 1.0 handout.

XML repositories. In ArferDB, each query is run against a single XML repository(collection). An XML repository is a collection of XML documents, i.e., essentially, a forest of XML trees. XML documents belonging to the same repository need not share the same schema, origin etc — any group of XML documents can be treated as a repository.

XML repositories and tinyFS. tinyFS models a disk as a single file containing a preset number of disk blocks. Internally, tinyFS maintains a single directory with multiple files.

ArferDB stores information about a single XML repository in a collection of data files. Because tinyFS allows no directories, we assume that the name of a tinyFS file representing one of the data files for a specific XML repository follows roughly the following naming schema:

```
<Repository>-<FileName>.dat
```

where `<Repository>` is the name of the XML repository and `<FileName>` is the name of the data file in the repository. All files for all repositories will be stored in the "root" tinyFS directory.

XML Storage Layer

The lowest layer of Stage 1.5 uses the data file layer to create and manage persistent data structures for storing XML documents in XML repositories. XML storage in ArferDB consists of two parts:

- Primary data storage: each XML document is shredded into a collection of records that allows for fast construction of XML fragments from the data stored and managed by ArferDB.
- Secondary indexes: ArferDB implements inverted indexes with postings sa well as *inverted parent-child indexes* to index the structure of an XML document.

Requirements for XML Storage Layer

All layers of Stage 1.5 give each team the repsonsibility for appropriate design. Some design ideas and suggestions are given in this document, but each team has the right to change the design and implement the ArferDB functionality in a different way. However, all implementation must satisfy a number of requirements.

R1. Repository Creation. The repository creation function shall have the following declaration:

```
int createXMLRepository(Buffer buf, char * name);
```

The function shall create a collection of empty tinyFS files on the tinyFS disk associated with the specified buffer and prepare both the tinyFS disk and the buffer for further work with the newly created repository.

The function shall return a status code: 1 if the repository was successfully created; other value (you can have as many error codes as you need) if the repository could not be created. Typical reasons why repository cannot be created include:

- Duplicate repository name (repository with a given name already exists)
- Disk full (not enough disk space to host all data structures for a new repository)
- Buffer full (not enough unpinned space in the buffer to process repository creation)

See discussion of what files form an XML repository below.

R2. Repository Deletion. The repository deletion function shall have the following declaration:

```
int deleteMLRepository(Buffer buf, char * name);
```

The function shall remove all data files associated with the given repository as well as remove any support data structures and/or information about the given XML repository from any data structures supported by ArferDB. The function shall return an exit code: 1 is deletion was successful; another number (error code) if it was not. Typical reasons why repository cannot be deleted include:

- Incorrect repository name (repository with a given name does not exist)
- Buffer full (not enough unpinned space in the buffer to process repository creation)

R3. Insertion of an XML file. This is the **main** function of the XML management layer. The function shall have the fullowing definition:

```
int insertXML(Buffer buf, char * repository, char * filename);
```

This function finds the XML file with the specified name (`char * filename`) and coordinates the process of inserting the contents of the file in the specified (`char * repository`) XML repository. At the high level, the function is responsible for the following activities:

- Parsing of the XML document.
- Shredding of the XML document into the primary data storage.
- Indexing of the XML document in the secondary index structures.

See dicussion of each step below.

The function shall return an exit code: 1 if the insertion was successful; another number (error code) if it was not. Typical reasons for failure to insert an XML document are:

- Incorrect file name (XML file with a given name does not exist)
- Corrupt input file (file with a given name is not a well-formed XML document)
- Incorrect repository name (repository with a given name does not exist)
- Buffer full (not enough unpinned space in the buffer to process repository creation)
- Disk full (not enough disk space to store the file in the tinyFS disk)

Note: An insertion of an XML document shall be treated as an atomic transaction. If the insertion succeeds, the data stored in the repository becomes **persistent** and visible to other data requests. If the insertion fails, there shall be no data related to the attempted insertion left in ArferDB-controlled structures. This means that if your implementation observes inability to continue work after inserting some data into the XML repository, it is responsible for **graceful roll-back** of all changes to the database state.

R4. Primary XML data store. Each XML repository shall consist of two sets of data files: one for primary storage of XML data and one: for secondary indexing.

The specific design of the data files for primary XML store is left up to individual teams. We discuss some options in the next section, but each team can modify the internals of the data store to suit better its needs.

Regardless of specific implementation though, the primary data store needs to satisfy the following requirements.

1. The primary data store may consist of one or more data files of one or more data file types.
2. Implementation of each data file type shall be based on the data file management layer of ArferDB.
3. For each data file type used in the primary XML storage, you must implement the functionality that supports the following:

(a) formation of records stored in the data pages of the file: this needs to be done for each type of data page present in the data file.
(b) data access in a record: functions to retrieve infromation from a given data record.
(c) record management on a single data page.
(d) access to disk page headers and the information stored within them.
(e) insertion/deletion/updates of the data records: essentially a passthrough to the lower layer, but the record manipulation functions on this layer will know the specific structure of each type of record.
(f) record retrieval: also essentially a passthrough to the lower layer, but at this layer the functions will translate the retrieved byte arrays into actual data records.

The core operation on the primary data store, that needs to be implemented eventually, *but is not part of* Stage 1.5 is XML generation: given an XML node, output the XML document fragment rooted in that node. Your primary data store must allow for a relatively efficient XML reconstitution given a direct pointer to an XML element record.

R5. Secondary XML indexes. The secondary indexes serve as the main access point to the XML data stored in the repository. Most of the XPLite expressions will be resolved working only with the XML indexes, without touching the primary storage until XML-formatted output needs to be generated.

The specific design of the data files for the secondary index structures is left up to the teams. We discuss options in the next section, but each team can modify the design as suits its needs the best.

Regardless of specific implementation, the XML index structures implementation needs to satisfy the following requirements:

1. The index structures may consist of one or more data files of one or more data file types.
2. Implementation of each data file type shall be based on the data file management layer of ArferDB.
3. For each data file type used for XML indexes, you must implement the functionality that supports the following:
   (a) formation of records stored in the data pages of the file: this needs to be done for each type of data page present in the data file.
   (b) data access in a record: functions to retrieve infromation from a given data record.
   (c) record management on a single data page.
   (d) access to disk page headers and the information stored within them.

(e) insertion/deletion/updates of the data records: essentially a passthrough to the lower layer, but the record manipulation functions on this layer will know the specific structure of each type of record.

(f) record retrieval: also essentially a passthrough to the lower layer, but at this layer the functions will translate the retrieved byte arrays into actual data records.

The key operation on the existing index is fast lookup of data by a specific key. Therefore, regardless of implementation, your index structures shall be tailored for fast discovery of infromation.

Discussion of Options

This section of the document is non-normative. It contains suggestions for possible ways to implement the primary XML storage and the secondary XML indexes. Each team is allowed to modify any of the approaches discussed here and/or choose approaches of their own.

Primary XML Storage

Intuitively, the key functionality we want out of the primary storage boils down to this:

1. **Partition.** Each XML element can be accessed independently.
2. **Reconstruction.** Given direct access to a an XML element record, it should be a straightforward and efficient process to *reconstitute* an XML fragment with the given XML element as the root.

The partition requirement essentially bars storage of XML on disk as a single blob of text, as this would make discovery of individual XML data difficult[1]. A better way is to shred the XML document.

In class we discussed XML shredding. To simplify how we shred XML, we adopt the following limitations on the XML documents ArferDB will be working with:

1. No mixed content. The content of an XML element is either a single isntance of `PCDATA`, or a sequence of other XML elements, or the XML element is `EMPTY`.
2. Upper bound on element name length. Pick a reasonable upper bound on the names of XML elements. E.g., 20 bytes is most likely sufficient for everything we need.
3. Upper bound on attribute name length. Ditto.

[1]This is not quite true. It is possible to store an XML document as plain text on disk, and index the occurrence of each XML element by its byte position. This might solve immediate issues, but it makes anything other than direct access to XML fragments in the primary storage extremely inefficient.

4. No namespaces. All elements come from the same default namespace, there are no `<foo:bar>` elements.

5. XML documents are static. Once inserted into the database, XML documents won't undergo any modifications (this should make some indexing easier).

Notice that XML documents can still have arbitrary length information as (a) values of attributes and (b) content of XML elements.

One possible way to shred XML documents that are subject to limitations above is described below. The XML document is shredded into three types of records:

1. XML element records. These are going to be fixed-length records stored in a sequential file.

2. XML attribute records. Variable-length records (one variable length field). Stored keyed to a unique Id, with only direct access needed. Can be stored either in sequential or hashed files.

3. XML content records. Variable-length records (one variable length field). Can be stored both in a sequential or in a hashed file.

XML Element records. A simple XML element record can have the following structure:

```
 XMLElement:
        int XMLFileId     // Id of an XML file in the repository.
                          //      Could be filename, but may be easier to use an int
        int nodeId        // Depth-first search order Id of the XML node in the document
        int parent        // pointer to parent nodeId. could be replaced with  DiskAddress parent
        int isLeaf        // flag specifying if current node is a leaf
        int isEmpty       // if isLeaf == 1, isEmpty specifies if the node has any content
     String name          // name of the XML element  (could also be an int if needed)
        int ordinal       // position of the XML element among its siblings
DiskAddress attributes    // pointer to the first attribute record storing
                          //                information about its attributes
DiskAddress content       // for leaf nodes, points to the record storing its content,
                          //                                                otherwise NULL
```

These records have fixed length. A SAX parser will be able to generate XML record descriptions in the depth-first search order. Therefore, assuming increasing `XMLFileId`s (1 for the first inserted file, 2 for the next, and so on), XML element records can be stored in ArferDB's sequential file keyed by the pair `<XMLFileId, nodeId>`[2].

[2]This may require some retooling of **key exposure** to the sequential file and hashed file implementations. Essentially, a key may no longer be just a simple `int`. Alternatively, you can build unique keys from a pair of numbers.

XML Attribute records. A simple XML attribute record can have the following structure:

```
XMLAttribute:
    int XMLFileId   // Id of an XML file in the repository
    int XMLNode     // nodeId of the XML node the attribute belongs to
 String attName     // attribute name
 String attValue    // attribute value
```

According to our XML restrictions, only the `attValue` attribute in this record will have variable length. To store this record on disk, you can add a record header that specifies either the length of the `attValue` string, or the length of the entire record. Knowing the offset of a XML attribute record on disk allows for immediate access to the starting position of the `attValue` value.

Attribute records specifies as above can be stored in either sequential files, keyed by `<XMLFileId, XMLNode>`, or in a hashed file.

In a sequential file, the key is not unique, but all attribute records belonging to the same XML element will be clustered together, so the `attributes` pointed from the XML element record essentially serves as a sparse index on XML attributes. Parsing XML documents using the SAX parser will yield only "end-of-data-file" insertions of attributes - all attributes are parsed at the moment their host XML element is recognized.

In a hashed file, `<XMLFileId, XMLNode>` key is hashed, and all attributes for the same XML element are stored in the same hash bucket, but might be stored on different pages and possibly interleaved with attributes from other XML elements when there is a hash key collision. As such, using sequential file may be preferable here.

XML Content records. A simple XML content record can have the following structure:

```
Content:
    int recordLength   // length of record in bytes. Alternatively, can be length
                       //                                        of content field in bytes
    int XMLFileId      // Id of an XML file in the repository
    int XMLNode        // nodeId of the XML node the attribute belongs to
 String content        // content of the XML element
```

Here, `content` has variable length, so we include a record length field in the header of the record (or a content length field).

Unlike attribute records, there is exactly one content record that can be associated with an XML element. Therefore, XML content records can be stored in a straightforward manner both using sequential and hashed files. One caveat that must be observed is that the length of XML content can potentially exceed the size of a single disk page. We discount this possibility for attributes (attribute values are never that long), but we cannot ignore it for XML content. You must be able to handle this situation gracefully.

Inverted Indexes

There is a number ways in which the inverted index and the parent-child inverted index can be built. We discuss a few of these ways below.

Addresses. A `DiskAddress` data type used in this document can be built in a number of ways, but it needs to contain the following information:

1. Name of the data file
2. Block location (either absolute, or within the data file)
3. Record location on the block

A possible structure for this address is

```
typedeft struct {
   ArferDBAddress block;  /* FileId, BlockId pair      */
   int  offset;           /* byte offset of the record */
}  DiskAddress;
```

This structure allows for quick access to the data using lower layer functionality. You can use

```
DiskAdress address;
 ...
readBlock(buf, address.block);
```

to bring the block into the buffer, then

```
int recordLength;
unsigned char * blockHeader = calloc(SIZE_OF_DISK_BLOCK_HEADER, sizeof(char));
unsigned char * record;

blockHeader = read(block, 0);  /* read header */
/* discover record length here */
recordLength = .....
/***/
 record = calloc(recordLength, sizeof(char));
 read(buf, address.block, address.offest, recordLenghth, record);
```

(your mileage may vary, and somewhat different code is possibly needed to retrieve a record of variable length.)

You also could add the record length directly to the `DiskAddress` structure to make record retrieval more straightforward.

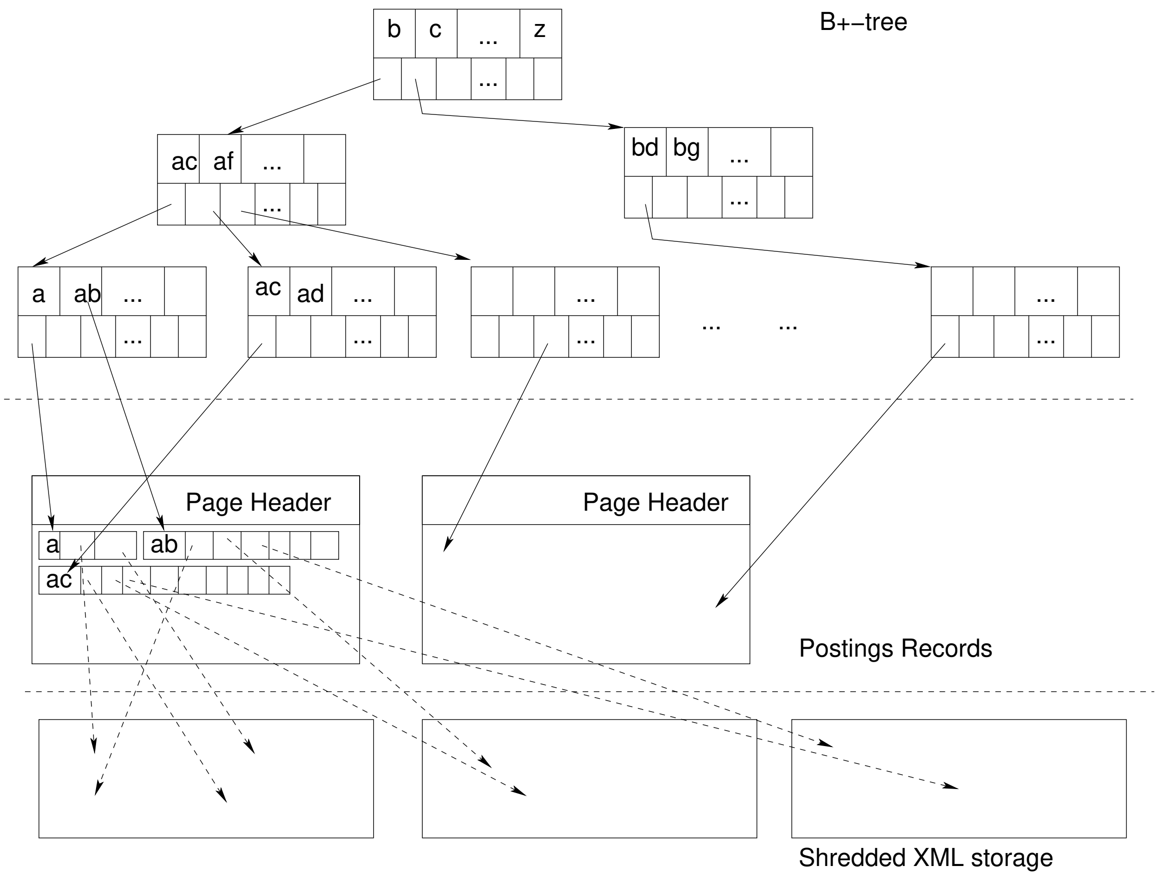

Figure 1: Possible organization of an inverted XML index. The top layer is a B+-tree of fixed-length keys (e.g., XML element names, XML attribute names). The middle layer is the sequential file of postings. The bottom layer shows the shredded XML document storage.

Inverted XML index. The inverted XML index consists of four componenets: XML Element Index, XML Attribute Index, XML Content Index (PCDATA Index) and XML Attribute Value Index (CDATA Index). Each can be built separately using the same principles. The four components of the index point to the three different parts of the shredded XML file:

- XML Element Index indexes `XMLElement` records in the primary storage.
- XML Attribute Index and CDATA Index both point to the `AXMLAttribute` records.
- XML PCDATA Index indexes `Content` records.

All key values for all four types of keys are strings. *Lexicographical order* allows us to introduce a total ordering on all such keys. This, in turn, allows us to use efficiently sequential structures to store the indexes.

Inverted Index for fixed-length keys. For the XML Element Index and XML Attribute Index, each key value (name of an XML element or of an XML attribute) has fixed length. This allows us to employ B+-trees to index all the keys. The actual postings records are of variable length due to the fact that each XML element/attribute can occur multiple times in the XML repository, so they cannot be stored directly in the B+-tree. Figure 1 shows one possible solution - a new layer of variable-length records with repeated attribute groups is stored in a sequential data file. Each record is pointed at by a pointer in a leaf node of the B+-tree. For a two layer B+-tree. the total of three (3) disk reads is needed to retrieve all locations of an XML element/XML attribute in the XML repository.

Inverted Index for variable-length keys. You have a number of options on how to index variable-length keys: attribte and content values.

- Use a special-purpose hash function that hashes the values into a fixed-length code. Use the architecture shown in Figure 1 to index the hashes.

  This approach may lead to *collisions*: same hash code is assigned to multiple different values. This can be addressed at the point of data retrieval: retrieve every data records for which a posting exists, but filter it at the retrieval time. If your hash function admits a large enough number of unique values, you can get away with relatively few collisions, and hence, this approach will be efficient, despite potential for wasted disk I/Os when retrieving records that will be filtered out.

- Use "elastic-key size" B+-tree, i.e., make the key fields of the B+-tree blocks to be variable length. This means that each B+-tree node may potentially store a different number of keys. However, while arithmetics might be hard, the actual management of "elastic" B+-trees is not very hard - they can be implemented as sequential files for records with variable length fields.

- Use a B+-tree in which each key value is a DiskAddress pointer the actual value, which is stored separately. This is not a very efficent method in theory - but if all key values are stored consecutively, you may get a lot of cache hits when trying to read key values (reading the first key value from a disk block brings the page to the buffer, reading the next key value leads to the same page - it is in the buffer, so we get a "cache hit").
- Use a simple dense index stored in a sequential file. Find appropriate values by full index scan, or via binary search.

Postings Record structure. We want to be able to perform a lot of query processing operations directly inside the index, without needing to access primary XML storage. At the same time, we also need to be able to access the full record for each indexed feature of the XML document. To support this, we need to store two different pieces of information about each indexed feature:

1. Address of the primary XML storage record.
2. Identity of the XML element which "houses" the specified feature.

We note that all indexed features: XML element names, XML attribute names, attribute values and content have a "host" XML element associate with them. This element is used to key each feature in the primary storage.

A simple structure of a posting record is therefore:

```
typedef struct {
  DiskAddress recordLocation;  /* location of the record in the primary XML storage */
  int DocId;  /* id of the XML document in the XML repository */
  int NodeId; /* id of the "host" XML node in the XML document */
} PostingRecord;
```

The entire index record then may look as follows (using pseudocode here for simplicity):

```
InvertedIndexRecord:
   int  nPostings;                 /* number of postings records */
   char[MAX_KEY_LENGTH] keyValue;  /* key value being indexed */
   List <PostingRecord> postings;  /* postings records */
```

This structure allows one to quickly identify the length of the postings record in bytes and retrieve the record from its location on a disk page.

Parent-Child Inverted Index. The parent-child inverted index can be built in a way similar to the simple inverted XML index. The key difference is that the parent-child index has a compound key the consists of two values. We can deal with this issue in two ways.

Suggestion 1: Simple B+-tree. The parent-child inverted index keys have the form `x/y`, where both `x` and `y` are strings. We note that if `y1,y2,y3,...,yN` is a lexicographic order on `y`s, and `x1,...xM` is a lexicographic order on `x`s, then

```
x1/, x1/y1,  x1/y2, ..., x1/yN, x2/, x2/y1, ..., x2/yN, ..., xM/, xM/y1, ...,xM/yN
```

is a proper lexicographical order as well. To complete it, we assume that `<root>`, i.e., the `""` value for `x` is the first one (i.e., `x1 = ""`. This allows us to construct a B+-tree indexing the pairs `x/y` of the keys in a single data structure.

Suggestion 2: Two-layer B+-tree. Here, we construct two tiers of B+-trees. The top-tier B+-tree indexes the values of the first key component. The leaf pointers point to the roots of the second tier B+-trees, each of which indexes the values of the second key component that are children of the parent key value. E.g., if `x` is a label of a leaf node in a top-tier B+-tree, and all `x`'s child key values found in the XML repository are `y1,..., yN`, then the lower-tier B+-tree associated with `x` will index values `y1,..., yN`. Figure 2 shows the architecture of such a data structure.

Parent-child index postings record. When indexing parent-child data, each posting essentially points to a pair of XML features: the parent feature and the child feature. You have the following options, when indexing them:

- Store only the pointer/NodeId of the parent. The child is discovered by retrieving the parent from primary storage and searching for its children/attributes/content. (this excludes *Attribute name/attribute value* pairs, as both are stored in the same record in shredded XML).
- Store only the point/NodeId of the child. Parent can be retrieved because all non-XML nodes store the nodeId of the XML element they belong to, and XML elements also store parent links.
- Store information on both the parent and the child. This way, you can access directly both parent and child records, which may be of interest depending on the context.

Deliverables and Submission

For the software, we take the "snapshot" approach. The snapshot of your Stage 1 implementation shall be submitted both to your team's repository on the course Github and using `handin`:

```
$ handin dekhtyar 468-stage01 <files>
```

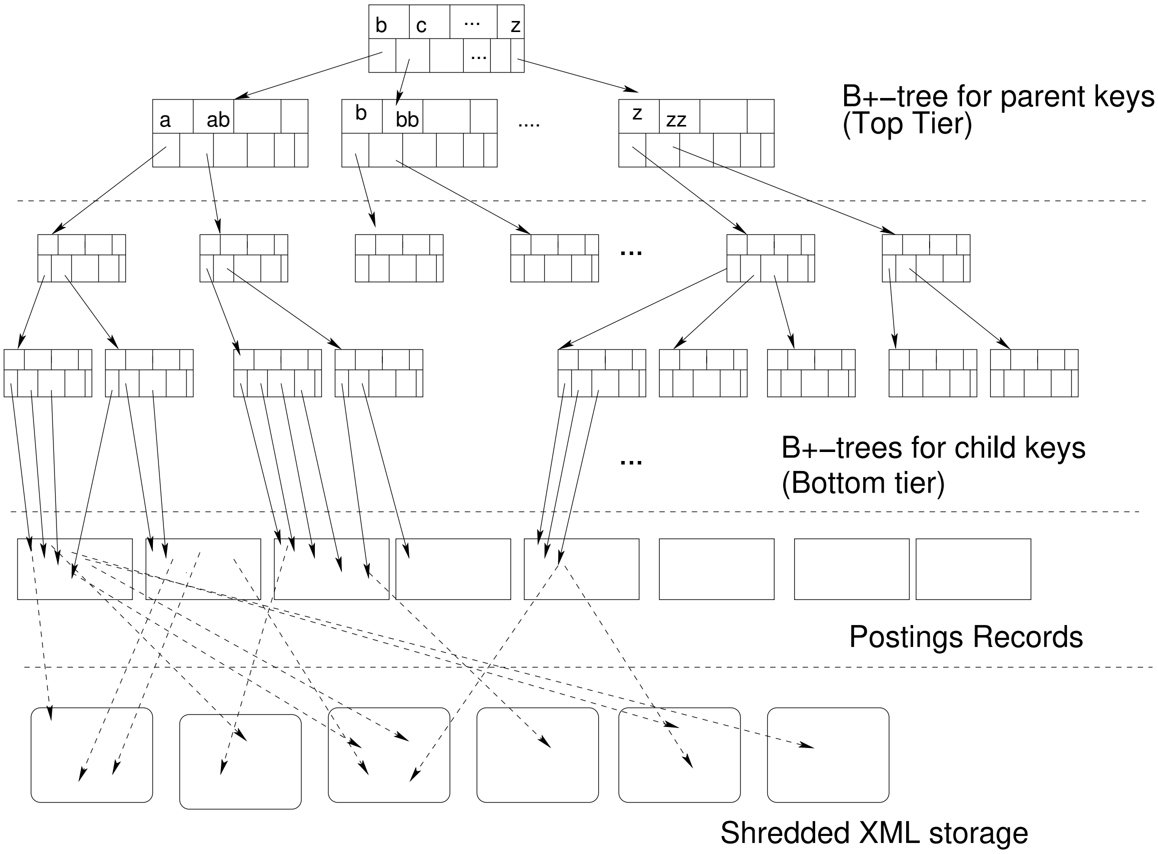

Figure 2: Possible organization of a 2-tier inverted parent-child XML index. The top layer is a B+-tree for parent keys. Next tier is a collection of B+-trees (one per parent key vlaue) indexing child keys. The bottom tier of the index structure is the postings records, which contain pointers to the primary XML storage.

The `handin` submission must contain all files necessary to compile and test your system (see below). It must also contain a `README` with the following information:

- Team name, and list of students (names, email addresses)
- Compilation and test running instructions.
- Breakdown of who did what.

Demo. The snapshot submission is for archival purposes. The real graded deliverable is a demo, each team must present to me. The demo must demonstrate that your Stage 1 functionality is sufficiently operational to successfully complete the core Stage 1 tasks. During the demo, you shall demonstrate the following abilities of your system:

1. XML Repository management: creation/deletion.
2. XML document insertion into an XML repository.
3. Examination of the contents of the data files stored in tinyFS partitions.

Basically I want to see you create an XML repository, insert one, and then at least one more XML document into the repository, examine the state of the tinyFS disk file, and specifically, examine any and all data files stored on the tinyFS partition, and then successfully delete the XML repository.

To facilitate the demo, you might want to create a number of simple test programs - one to insert XML data into the XML repository, another – to examine (dump) the contents of the disk pages and data files, the third – to delete repositories. Specifics are left to individual teams.

I only need 2 people from the team at each demo - we may do some during labs/lectures. We may do other demos outside of class time.

Good Luck!