Project: Stage 3
Query Processing and XML generation

Due: 11:59pm, Thursday, June 5, 2008.

Outline

This is the final stage of the project. On this stage, you have a single task: build a query processor for an XML query language XPlite.

As part of the query processing task, you will also have to generate XML fragments corresponding to the nodes retrieved by your queries.

The full description of *XPlite* can be found in a supplemental handout. Please study it first, before proceeding.

Xpath-plus Modifications

As in Stage 2, your submission will consist of two parts: the database client XpathPlus and the DBMS server MiniNXBase. In addition to the commands implemented in Stage 2, XpathPlus shall implement one more family of commands:

- **XPLite Query**. The syntax of the this command is as follows:

  RETURN document("RepositoryName")XPLiteQuery;

  Here RepositoryName is the name of the repository to which the query is to be applied, and XPLiteQuery is the actual query written in XPLite. (Note, all XPLite expressions start with a "/" and are considered absolute expressions in XPath terminology).

  For example to submit XPLite query /child::a/child::b/following::*[pos()=1] to the repository `"test"`, the following command will be issued:

  RETURN document("test")/child::a/child::b/following::*[pos()=1]

Once the query is submitted, XPath shall wait until it is executed by the DBMS server and shall output the result sent back by the server.

MiniXNBase Modifications

When the RETURN command is passed to MiniNXBase server, the following shall happen:

1. As is the case with all other commands, the server shall create a new thread to handle the command.
2. The command shall be recognized; the repository name and the the query shall be extracted.
3. The query shall be parsed.
4. The query shall be processed over the repository specified in the command.
5. The result of the query, a list of XML nodes shall be obtained.
6. For each node in the result, its XML representation shall be generated.
7. XML representations of all result nodes shall be returned back to Xpath-Plus.

XML Generation

The result of an XPLite expression is a list of XML nodes sorted in the *document order*. Each node appears no more than once in that list. MiniNXBase has to generate XML representations for all the nodes in the result.

XML Representation of an XML element node: If a node in a result set is an XML element node, then its representation is an XML fragment whose root is this XML element, and whose contants are the subtree of this element in the original XML document.

XML Representation of an XML attrubute node: If a node in a result set is an XML attribute node, then its representation is the **opening tag** for its parent (i.e., the XML element node which has the current attribute). If current XML attribute has name *attr* and value *"val"*, and its parent element is `<x>`, then the XML representation of this node is

```
<x attr="val">
```

Note, that `<x>` may have other attributes — they are to be ignored in creating the representation for *attr*. Note also, that if two resulting nodes represent two attributes of the same XML element, the representations for these two nodes need to be generated intependently.

Query Processing

The query processor for *XPLite* should follow the basic architecture of the a query processor for the relational DBMS:

- Parser;
- Preprocessor;
- Logical Query plan generator;
- **[Optional]** Physical query plan generator/plan optimizer.
- Implementations of atomic operations of XPLite.
- XML generation (see above).

You are responsible for the specifics of the implementation of all stages, and the exact APIs for all classes you build for this stage of the project. The section below offers some hints and suggestions, but it is NOT normative (i.e., you do not have to follow the suggestions in it).

Notes on Implementation

Parser

XPLite queries have the form (see *XPLite* handout for more instructions)

“/”LocationStep1“/”LocationStep2“/”. . . /LocationStepN

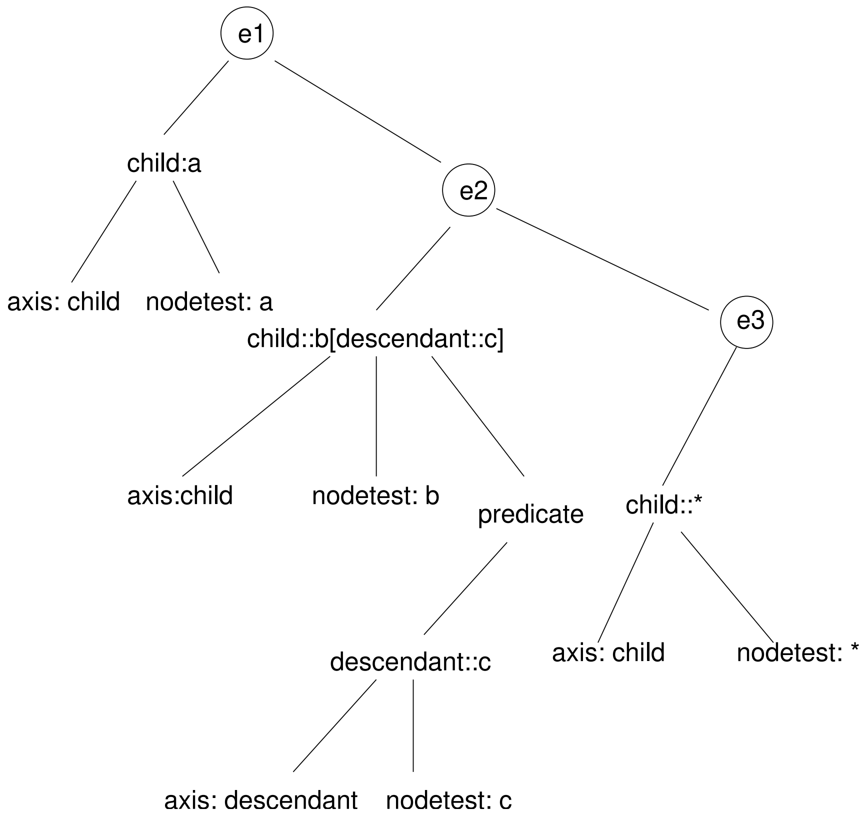
Each location step is a triple: `Axis"::"NodeTest"["predicate"]"` (with the predicate part being optional, or, possibly, repreating multiple times). The goal of your parser is to recognize individual location steps, and break each location step into the (axis, node test, predicate triple). There is enough of “syntactic sugar” in the query to make is easy to determine the boundaries of each part of the location step, as well as the boundaries of the location steps.

There is a relatively straightforward tree-like data structure that can be used to represent parsed *XPLite* expressions. The root of the tree is a node representing the the entire expression. It has two children. It’s first (leftmost) child, represents the first location step, and has three children: axis, nodetest and predicate. The predicate node may be a leaf, or, it may be a root node for a nested *XPLite* expression. The second child of the root, represents all of the expression except for the first location step, and so on.

For example, *XPLite* expression

```
/child::a/child:b[descendant::c]/child::*
```

will have the following representation.



(a simpler representation exists, which would omit the circled nodes).

Preprocessing

Once the expression is parsed, you must ensure that it can be successfully processed. This includes making certain that all axis names and all nodetests are correct. Most of these checks can actually be done during parsing, so, it is possible that you can avoid a standalone preprocessing stage.

Logical plan generation

XPath query evaluation is an ongoing research area for the database community. At the same time, several straightforward methods are not difficult to implement.

The most straightforward and naive query plan is to evaluate the XPLite query in the order of appearance of location steps. Basically, you follow the three evaluation rules presented below:

evaluateLC(Axis::NodeTest[Predicate], Input) = evaluatePredicate(Predicate, evaluateNodeTest(NodeTest, evaluateAxis(Axis, Input))).

evaluate(LocationStep/Expression, Input) = evaluate(Expression, evaluateLC(LocationStep, Input)).

evaluate(/Expression) = evaluate(Expression, getRoot()).

These three rules describe exactly the behavior of the *evaluate* function (method), which takes as input an XPLite expression. *Input* parameter is the current list of nodes, which, for absolute expressions is always the root of the tree.

The evaluation of the XPLite expression here is reduced to implementing three base functions: *evaluateAxis()*, *evaluateNodeTest()* and *evaluatePredicate()*. These functions will be addressed below.

Alternative ways of Logical Plan Generation

The main disadvantage of the plan generation strategy described above is its inefficiency: each call to evaluation function will yield one or more scan of the underlying data structures (implemented in Stage 1 of the project), whereas, in many cases, the scans for consecutive operations (e.g., axis and node test evaluation) can be combined.

We observe that XPLite has 10 axes, two distinctly different types of node tests (name node test and type node test) and two distinctly different types of predicates (nested XPLite expression and a comparison). Additionally, predicates are optional. This gives rise to $10 \cdot 2 \cdot 3 = 60$ possible combinations for a given location step. One can replace three top-level *evaluate* functions with the a collection of more specific functions, that evaluate the axis, the node test, and, applicable, the predicate in one pass. We note the following:

- Node name nodetest can be evaluated "on-the-fly" while **any** XPLite axis is being evaluated. As long as the evaluation function is "aware" of the node name, against which the test needs to be performed, this operation, this can be incorporated directly into the axis evaluation process. Thus, it may make sense to create two version of *evaluateAxis* method: one, for evaluation of the axis only, and one for evaluation of the axis, given the name of the XML nodes to be returned. From the pure coding point of view, the second method requires only minor extension of the code for the first method.
- Of all node type nodetests, only `text()` is non-trivial. In particular,
  - $*$, a.k.a., `node()` nodetest given a node, always returns `true`, therefore, these nodetests can be all but ignored.
  - `attribute()` nodetest will fail on any XML node from `StructureIndex` and will succeed on any XML node from `AttributeIndex`. Since the only way to "get" to attributes is via the `attribute` axis, the following describes evalution of `Axis::attrbiute()` pair:

    ```
    if (Axis == attribute)
       return evaluateAxis(attribute, InputNodeSet)
    else return EmptySet;
    ```
  - `text()` nodetest succeeds on any XML element node which does not have any child nodes (save for content). The `IsLeaf` field of the `StructureIndex` record contains this information. Thus, `text()` nodetest can be evaluated without any extra effort in the same scan as

the axis evaluation: all you need to do is check the `IsLeaf` attribute of any node, retrieved by axis evaluation procedure.

- Comparison predicates can be evaluated without additional accesses to the `StructureIndex`. Indeed, `pos()` and `last()` require only the information about the context node set. Evaluation of `string()` **does** require extra disk reads, but you need to access `XMLContent` structure. In particular, the following needs to be noted:

  – `string()` can be evaluated on the fly: as soon as the XML node (element or attribute) passes the axis+node test evaluation, its `string()` value can be computed. This allows you to streamline processing of comparisons involving `string`.

  – `pos()` and `last()` functions require full context to be generated prior to their evaluation. However, we can note that (i) the value of `last()` depends only on the entire context nodeset, and won't change from individual node under consideration. This value can be computed and conveniently stored for further use at the end of the axis+nodetest evaluation; (ii) `pos()` will be evaluated on all nodes from the context nodeset. It is possible to precompute the position of each XML node in the context nodeset during the axis+nodetest evaluation procedure, although one has to be careful about it: *remember that the final order of the nodes is based on their location in the document, NOT on how they were added to the nodeset, so, any time a new node is added to the context, it has to be put in its precise position and the positions of other nodes have to be adjusted.*

- A brief discussion of evaluation of individual axes is below.

Physical plan generation

If you include more than one version of each (or some) of the evaluation functions for atomic operations, you need a physical plan generation stage, at which your code will have to select the actual implementation to be used when executing the query plan.

This part is optional, and its implementation will yield up to 20% extra credit (depending on how much is implemented).

Implementation of XPLite Axes

There are two basic ways in which XPLite axes can be evaluated. The first way involves implementing four DOM methods: `getFirstChild()`,`getNextSibling()`, `getParent()` and `getPeviousSibling()`, and then implementing each axis using only these four methods. In this approach, the four DOM methods will access the index structures and scan the database for information, while the axis implementations will be abstracted from the direct data access.

The second way involves direct implementation of XPath axes using Stage 1 API, via scans of the XML index structures.

Implementing XPLite Axes via DOM

Each XPLite axis can be represented as a sequence (possibly recursive) of calls to the four DOM methods: `getFirstChild()`,`getNextSibling()`,`getParent()` and `getPeviousSibling()`. E.g., we can represent the descendant axis as

getFirstChild()$^+$.getNextSibling()*;

i.e., as all nodes reachable from the current node, via first child and next sibling traversal.

The expressions for each axis can be found in the supplemental XPath handout available at the following URL:

http://www.csc.calpoly.edu/∼dekhtyar/560-Fall2007/lectures/lec11.560.pdf

The DOM methods can be implemented as follows:

- `getParent()`: the description of each XML node in `StructureIndex` contains `Parent` field, which stores the node ID of the parent node. The parent node then, can be retrieved directly using Stage 1 API.
- `gefFirstChild()`: `StructureIndex` has the following nice property: *if an XML element node N is children, then the record for its first child is the next record after the record for N*. Checking if a node has children can be done using the `IsLeaf` field of the `StructureIndex` record.
- `getNextSibling()`: Next sibling of a node N can be found by a forward scan of `StructureIndex` starting at the position of the record for N until a node M whose $Preorder$ is greater than the $Postorder$ of N is reached. Comparing the parents of N and M will reveal whether M is the next sibling of N.
- getPreviousSibling() Previous sibling of a node N can be found by a backwards scan of `StructureIndex` starting at the position of the record for N until a node M whose $Preorder$ and $Postorder$ are smaller than the $Preorder$ of N is reached. Comparing the parents of N and M will reveal whether M is the previous sibling of N.

Direct Computation of Axes

DOM-based approach is relatively easy to implement — only four methods ”touch” `StructureIndex`, and has the advantage of making the query processor implementation-independent. But it has a key disadvantage: it is slow. Here we discuss briefly how axes can be evaluated using Stage 1 API directly.

- `self`: trivial.
- `parent`: see `getParent()` above.
- `child`: scan `StructureIndex` starting with the context node, and until a node whose $Preorder$ is greater than the $Postorder$ of the current node is reached. For each node scanned, check if the context node is its parent.
- `descendant`: same as child, except return all nodes retrieved.
- `ancestor`: starting from the beginning of `StructureIndex` perform a scan until you reach context node, retrieve all nodes whose $[Preorder, Postorder]$ interval is a super-interval of the $[Preorder, Postorder]$ interval of the context node.
- `following`: Scan `StructureIndex` starting with the context node, do not output any nodes that match the `descendant` condition, retrieve all other nodes.
- `preceding`: Scan `StructureIndex` from its beginning and until you reach the context node, report all nodes, for which both $Preorder$ and $Postorder$ are less than the $Preorder$ of the context node.
- `following-sibling`: Scan `StructureIndex` starting at the context node. Report all nodes whose $Preorder$ is greater than the $Preorder$ of the context node, which have the same parent as the context node. Stop when you reach a node whose level is smaller than the level of the context node.
- `preceding-sibling`: Scan `StructureIndex` forward starting at the parent of the context node. Report all discovered siblings of the context node. Stop when you reach the context node.
- `attribute`: Use Stage 1 API to find the first record in the `AttrbiuteIndex` for the context node. Scan `AttrbitueIndex` for the remaining records. Note: if you know the name of the attribute (from the name node test that follows the axis), you only need to retrieve the attribute with that name.

Grading

This is the ultimate stage of your project, and the success of the project will depend on how well you have implemented the entire system.

We will release the tests at least 10 days prior to the submission deadline. All tests will be run through the XpathPlus client – there will be no Java wrappers testing individual APIs.

Your grade for the entire project will be the grade you recieve for Stage 3, **unless** your Stage 3 submission earns less than **50%**. In the latter case, your grade will be computed as:

$$0.35 \cdot Grade(Stage1) + 0.35 \cdot Grade(Stage2) + 0.3 \cdot Grade(Stage3).$$

Note, that in the second scenario, you lose any credit for fixing bugs found on Stages 1 and 2, but, if your submissions for those stages were at around 80-90% level, you can get around 70-75% of the project grade.

Coding and Submission

Your submission must be accomanied by a README file with compilation instructions. It would be highly desirable to make sure that your project compiles by simply running `javac` on `MiniXBase.java` and `XpathPlus.java` (with whatever necessary CLASSPATH settings). Your submission MUST include ALL files necessary to run the server and the client (this includes any NEUSTORE classes you need).

While code is not graded, I reserve the right to examine it. Please comment adequately and make effort to make your code readable (your teammates will be the first to benefit from this).

You should have one submission per team. One person should email a **.zip** or a **gzipped tar file** containing your entire submission to dekhtyar@csc.calpoly.edu before the deadline. Please include the name of the group in the subject line of your email.

This is the final submission. No Late Submissions Will Be Allowed!