Project: Stage 2
XPLite Query Processing

Outline

This document outlines the ideas for ArferDB's query processor.

A separate document specifying the Stage 2 project deliverables and outlining submission instructions and deadlines will be handed out on November 19.

Stage 1 Overview

This document assumes that your version of ArferDB has the following Stage 1 components implemented and working.

XML Document Storage: primary. One Stage 1 each team needed to implement the process of shredding an input XML document into a collection of flat data records about the XML elements, content and attributes that form the document. In what follows we refer to the data file (files) storing the shredded XML information as XStore. If a specific component of XStore is of interest, we refer to them as XStore-Elements, textsfXStore-Attributes and XStore-Content.

XML Document Storage: inverted index. In addition to XStore, ArferDB indexes XML documents using two indexes: inverted index and inverted parent/child index. We refer to ArferDB's inverted index of XML documents as IIndex. IIndex has four components: IIndex-Elements, IIndex-Attributes, IIndex-AttValues and IIndex-Content, each indexing the occurrences in an XML file of, respectively, specific element names, attribute names, attribute values, and #PCDATA.

XML Document Storage: parent-child index. The parent-child index, referred to as PCIndex indexes all parent-child pairs found in an XML document. The following types of parent-child pairs can be found in an XML document:

1. Root/ XML Element
2. XML Element/ XML Element
3. XML Element/ Content (#PCDATA value)
4. XML Element/ Attribute
5. XML Element/ (empty XML elements)

6. Attribute/ Attribute Value

We assume that the following information is available to us in the primary and index records. The table below lists only the information within each record that we need in order to respond to XPlite queries. The records may contain additional information in them which is needed for other purposes.

XStore-Elements	IIndex	PCIndex
	Required	
Document Id *XML Node Id* (DFS order) *Parent NodeID* *Leaf flag* *XML Element name* *Pointer to attribute records* *Pointer to content record*	*Key Value*(Element name, attribute name, attribute value, #PCDATA value) *List of Postings* including: *Document Id* *Node Id* *Node address on disk*	*Parent Key Value* *Child Key Value* *List of Postings* including: *Document Id* *Parent Node Id* *Parent Address on disk* *Child Node Id* *Child Address on disk*
	Optional but Helfpul	
DFS In and DFS Out labels *Empty node flag* *Ordinal*	*DFS In and DFS Out labels*	

Stage 2: Query Processor

For full description of the XPLite query language, please see the XPLite handout. We will not repeat its contents here except for the bare minimum.

Recall that an XPlite query has the following format:

$$/\mathsf{Step_1}/\mathsf{Step_2}/\ldots/\mathsf{Step_k},$$

where each location step $\mathsf{Step_i}$ is:

$$\mathsf{Step_i} ::= \mathsf{Axis}' ::' \mathsf{NodeTest}\ [{}'['Predicate']'{}]^*$$

General approach. The XPLite query processor should consist of three core components:

1. Parser. The parser takes as input an XPLite query string and generates an XPLite query tree. The nodes of the tree are operations of the XPLite query algebra discussed below.
2. Evaluator. The evaluator traverses the XPLite query tree and executes each atomic operation of XPLite query algebra.
3. XPLite Query Algebra evaluation functions. Implementations of all atomic XPLite query algebra operations.

Each component is described below. Before proceeding though, we discuss XPLite query semantics.

XPLite Query Semantics

In ArferDB, each XPLite query Q is applied to a single XML repository R. Let $R = \{d_1, \ldots, d_N\}$ be the list of XML documents stored in the repository R. We define the formal *semantics* of an XPLite query Q on a repository instance R via the means of a function

$$Eval : Queries \times Repositories \longrightarrow \text{List of XML Nodes}.$$

Step 1: reducing repository queries to document queries. We start by defining the semantics of an XPlite query on a full repository *as the union of the semantics of the query on each document from the repository*:

$$Eval(Q, R) = \bigcup_{d_i \in R} Eval(Q, d_i).$$

Step 2: computing the semantics of a query on a single document. Given a single XML document $d \in R$, it is now our goal to define the semantics of the query Q on it.

Informally, given an XML document d with root element r, the semantics of Q is computed as follows:

- Start with a node list $L_0 = \{r\}$.
- For each location step $\mathsf{Step_i}$ do the following:
  1. Evaluate the axis $\mathsf{Axis_i}$ of $\mathsf{Step_i}$ on L_0.
  2. Evaluate the nodetest $\mathsf{NodeTest_i}$ on the results of the previous step.
  3. One by one, evaluate all predicates in the current location step.
  4. Set L_i to be the set of all nodes that passed the three steps above.
- Return $\mathsf{L_k}$.

Location Step Evaluation. On each location step, ArferDB query engine needs to evaluate the axis, the nodetest and any predicates found in it. Functions for evaluating these parts of the location steps form the core of the XPLite Query Algebra.

XPLite Query Algebra and Its Implementation

The core of the query engine lies in its ability to efficiently implement evaluation functions for each component of a location step. These functions **must take advantage** of the data structures we set up in ArferDB XML repositories.

We split the XPlite query algebra operations into two categories:

- Joint evaluations of axes and nodetests.
- Predicate evaluations.

Axis and nodetest evaluations

To take full advantage of ArferDB's index structures, we need to evaluate axes and nodetests in one step. XPlite has six axes (self, parent, child, ancestor, descendant, and attribute) and four different nodetests (Name nodetest, node() or *, attribute(), and text()). This leads to 24 possible axis-nodetest combinations, for which a separate evaluation strategy is needed.

Fortunately, (a) some of the axis-nodetest combinations are meaningless and always yield an empty set, and (b) in some cases multiple axis-nodetest combinations can be evaluated using exactly the same approach. This decreases the total number of *serious* evaluation functions you need to develop.

In the tables below, we enumerate all possible axis-nodetest pairs and identify the approach to evaluating them. While we are unable to provide the exact algorithms (it takes too much space), we specify which ArferDB data structures can be used in the evaluation. Before proceeding with the table, we make two important notes.

Note 1. Of the three core data structures, we prefer to do query processing on PCIndex when possible. Failing that, we try IIndex. Only when the query cannot be (completely) processed using the index structures, *or when using the primary data store is better* do we choose to evaluate the axis-nodetest pair using XStore.

Note 2. The key to proper understanding how evaluation functions work is the structure of the input to each function. The key input is the *XML Node List* consisting of the XML nodes that were selected on the previous location step. To make our life easy, we assume that the implementation of the *XML Node List* class allows you to sort/partition the contents of the node list by the following two keys:

- XML document Id
- XML element name

Given an XML node list $L = (n_1, \ldots, n_m)$, we denote as L_d, where d is an XML document, the sublist of L of all XML nodes belonging to document d. Similarly, given an XML element e, $L(e)$ denotes the sublist of L of all XML nodes for element e. You must implement both partitioning operations efficiently. Also, as $elements(L)$ we denote the list of all XML element names found among the nodes of L, and as $documents(L)$ we denote the set (list) of all XML documents whose nodes are in L.

We now proceed to consider each individual axis-nodetest pair. We assume that each evaluation function receives as input an XML node list L.

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
self	*ElementName*	IIndex	Search IIndex for all occurrences of *ElementName* Interserct with L
self	node(), *	none	Trivial. Return L.
self	attribute(),	none	Trivial. If L is a list of attributes, return L. Otherwise, return $\emptyset$.
self	text()	XStore-Elements	Retrieve XStore-Elements record for each node in L Check its *isLeaf* flag.

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
child	*ElementName*	PCIndex	For each $e \in elements(L)$ look for PCIndex records for $e/ElementName$. Intersect parent nodes with L.
child	node(), *	PCIndex	For each $e \in elements(L)$ look for all PCIndex e/x, where x is an XML element.
child	attribute(),	none	$\emptyset$.
child	text()	PCIndex XStore-Elements	equivalent to child::*/self::text() best done as a single piece of code though

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
parent	*ElementName*	PCIndex	For each $e \in elements(L)$ look for PCIndex records for $ElementName/e$. Intersect child nodes with L.
parent	node(), *	XStore-Elements	For each node $n \in L$, retrieve its XStore element record. Follow the parent link.
parent	attribute(), text()	none	$\emptyset$. (a node that is someone's parent cannot be a leaf node or an attribute)

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
descendant	*ElementName*	IIndex XStore-Elements	Search for IIndex records for $ElementName$. Retrieve their PCIndex-Elements records, and records for L. Compare *DFS In* and *DFS Out* values.
descendant	node(), *	XStore-Elements	For each node $n \in L$, retrieve its XStore element record. Scan XStore from each record to retrieve descendants.
descendant	attribute(),	none	$\emptyset$.
descendant	text()	XStore-Elements	Same as descendant:*, but return only nodes with *isLeaf* flag set.

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
ancestor	*ElementName*	IIndex XStore-Elements	Search for IIndex records for $ElementName$. Retrieve their PCIndex-Elements records, and records for L. Compare *DFS In* and *DFS Out* values.
ancerstor	node(), *	XStore-Elements	For each node $n \in L$, retrieve its XStore element record. Traverse the ancestors chain through parent pointers
ancestor	attribute(), text()	none	$\emptyset$.

Axis	NodeTest	ArferDB structures used	Evaluation Strategy
attribute	*AttributeName*	PCIndex	For each $e \in elements(L)$ search for $e/AttributeName$ PCIndex records. Intersect parent nodes with L.
attribute	node(), * attribute()	XStore-Elements XStore-Attributes	For each node $n \in L$, retrieve its XStore-Elements record. Jump to XStore-Attributes records, retrieve all. Alternatively, use PCIndex.
attribute	text()	none	$\emptyset$

Special Conditions. If L is a list of *attribute nodes* (i.e., the previous location step had a valid attribute axis), then child, descendant, ancestor and attribute axes shall return $\emptyset$. self and parent axes shall behave as follows:

- self axis is evaluated as in the table above, except, the searches are conducted through the list of attribute names, not element names, and text() nodetest returns $\emptyset$.
- parent axis is evaluated as in the table above, except, the searches are conducted on $ElementName/AttrbuteName$ types of parent-child pairs.

Predicate evaluation

Predicate evaluation can be broken into two parts: evaluation of individual built-in functions (bottom layer) and the overall predicate evaluation. We start with the built-in functions. Each built-in function is evaluated in the presence of the context set L of XML nodes. The actual function evaluation must happen **for each** node $n \in L$. In some cases, it is straightforward to compute all values of a given function on the entire context. In other cases, the values of the function are best computed individually for each $n \in L$.

position(). Split L into the contexts for individual XML documents: $L_{d_1}, L_{d_2}, \ldots, L_{d_M}$. For each context L_d, *sorted in document order*, associate with each node n its position in L_d. This can be done *on-the-fly*, without accessing any ArferDB data structures.

string(). If the function receives no arguments, the evaluation proceeds as follows. If the input node n is a leaf node (use XStore-Elements to check that if that information is not available from L itself), retrieve its content from XStore-Content.

If the input node n is an attribute node, retrieve its value from `XStore-Attributes`.

Otherwise, return "".

If the function receives as an argument an XPlite expression E, evaluate E (recursively), using the current node n as the input context node. Retrieve the first context node of the result. Evaluate string() (with no input parameters) on this node.

Note. In addition to direct computation of string(), because string values are actually indexed in IIndex, you may want to include in ArferDB an IIndex-backed implementation of the `[string() = <Text>` comparison, which might not require the direct computation of string().

contains(PathExpression, string) . Evaluate string(PathExpression, then check if `<string>` is a substring of the result.

last(). Break L into context nodes for individual documents: $L_{d_1}, \ldots, L_{d_M}$. With each document d associate $|L_d)$ as the answer.

count(PathExpression). Evaluate `PathExpression` with the given node $n \in L$ as the input. Retrieve the size of the resulting context.

not(PathExpression). Evaluate `PathExpression`. If empty, return `false`. Otherwise, return `true`.

true() and false(). Return constant values `true` and `false` respectively.

Overall predicate evaluation. A predicate subexpression of a location step has general syntax:

$$[\mathsf{Predicate}_1] \ldots [\mathsf{Predicate}_\mathsf{N}].$$

An XML node satisfies the predicate subexpression above, if it satisfies each individual predicate expression $\mathsf{Predicate}_1$, $[\mathsf{Predicate}_2], \ldots, [\mathsf{Predicate}_\mathsf{N}]$.

Each predicate expression is either a path expression or a comparison using built-in functions and constants.

Path expression predicates are evaluated by recursively calling the XPLite expression evaluation function, with a given input XML node as the input context set. The returning set of XML nodes is treated as `true` if it is not empty, and as `false` otherwise.

Comparison expressions are evaluated by evaluating the left side of the comparison expression, then evaluating the right side, and performing the appropriate comparison afterwards. Some special cases (e.g. string() = ¡Text¿) can be evaluated faster using ArferDB indexes.