On Region Algebras, XML Databases, and Information Retrieval

Vojkan Mihajlović, Djoerd Hiemstra, and Peter Apers*

Database Group, University of Twente, The Netherlands
{vojkan,hiemstra,apers}@cs.utwente.nl

Abstract This paper describes some new ideas on developing a logical algebra for databases that manage textual data and support information retrieval functionality. We describe a first prototype of such a system.

1. INTRODUCTION

In this document we will share some thoughts on building a database management system for textual, semi-structured, data. A simple, but often adequate way to manage textual data is to take an of-the-shelf information retrieval (IR) system. Typically, the system administrator has to tell the system what the documents are, and what fields, if any, can be used to search on additionally. We will use the word document here to denote the 'unit of retrieval': Documents are the only data type that can be returned by the system (usually the document's metadata and a reference to the actual location of the document). For instance, to search Web data, it is customary to consider a single web page as 'the document' and to index a fixed number of additional fields like domain name and title. Such a set-up has proven to be very useful on the Web, but once the decision is made to index Web *pages*, it usually impossible to search for complete Web *sites* that contain the query contents (assuming there is a way to infer what a web site is). Building a system that supports searching of complete web sites, or even both searching of single pages and web sites, would require a serious amount of reprogramming. Similarly, if we would like to add search services, like e.g. the option to search for Dublin Core metadata fields in web documents, then again some serious reprogramming is needed, as well as a change to the system's query language.

So, IR systems support the retrieval of documents, but whatever constitutes a document has to be determined by the application programmer at development time. Furthermore, many IR systems support the search and retrieval of some small fixed number of fields in a document (e.g. title, author). However, they lack the notion of *data independence*: any change in what constitutes a document, or any change in which fields need to be searchable, will lead to system developers having to get back at the programming code, or at least at rebuilding the entire database. In this paper we will show some ideas and approaches to building a textual retrieval system that *does* provide this kind of data independence. The system can insert documents that are specified in some standard format (i.e. XML), and support querying of any part of the data. Documents (i.e. the unit of retrieval) and additional search fields do not have to be determined at application development time, but can be specified by some declarative query language (e.g. XQuery with full-text search extensions [3]).

```
<SCENE>
  <STAGEDIR>Enter HAMLET</STAGEDIR>
  <SPEECH>
    <SPEAKER>HAMLET</SPEAKER>
    <LINE>So, let me say the famous quote
     again: <LINE>to be, or not to be</LINE>
     that's the question.
    </LINE>
    <LINE>Whether 'tis nobler in the mind</LINE>
  </SPEECH>
  <SPEECH>
    <SPEAKER>OPHELIA</SPEAKER>
    <LINE>Good my lord,</LINE>
  </SPEECH>
  <SPEECH>
    <SPEAKER>HAMLET</SPEAKER>
    <LINE>I humbly thank you;
      <LINE>well, well, well.</LINE>
    </LINE>
  </SPEECH>
   :
```

Figure 1: Example XML data

Figure 1 contains some (rather freely interpreted) example data from Shakespeare's plays [19]. Depending on the application, we might want to search for complete plays using some textual queries, we might want to search for scenes referring to speakers, we might want to retrieve speeches by some speaker, we might want to search for single lines using quotations and referring to speakers, etc., etc. The database system should operate relatively independent from the data and the kind of searches we want to do. A single database should support all kinds of queries – and more – without the need to rebuild it, or the need to have several copies of the data that each support a different kind of query.

The paper is organised as follows. In Section 2, we introduce region algebras. In Section 3 we extend the algebra to support requirements from XPath and XQuery. Finally,

*Thanks to the Netherlands Organisation for Scientific Research (NWO) for funding the research (grant 612.061.210). We'd like to thank Torsten Grust of U. Konstanz for making their XML loader available, Maurice van Keulen, Maarten Fokkinga and Mila Boldareva for helpful discussions.

DIR 2003, 4th Dutch-Belgian Information Retrieval Workshop.

Section 4 describes a prototype implementation of a system that supports XPath with extensions for full-text querying and ranking.

2. ALGEBRAS

An important aspect of developing systems that manage textual data is the definition of an algebra to manipulate data and mark-up. An algebra is a formal framework for manipulation based on operators and a domain of values. The operators map values taken from the domain into other domain values: every expression involving operators and values produces again a value in the domain. If we take as our domain all integers, and as the operators the sum $+$ and product $*$, then examples of algebraic expressions are: $(3+1)*(2+4)$, or: $(3*2)+(3*4)+(1*2)+(1*4)$, or: $4*6$, etc. Algebras might have certain properties. For instance, the arithmetic operator $*$ distributes over $+$.

A well-known algebra for data manipulation is the relational algebra [7] where the domain consists of all relations and examples of operators are selection σ, projection π, product $\times$, etc. Expressions are referred to as queries.

2.1 Boolean algebra for information retrieval

In the old days, of-the-shelf retrieval systems used the Boolean model of IR (see e.g. [15]). As said in the introduction, IR systems reason about documents. Not surprisingly, the domain of Boolean algebra for IR is sets of documents. Suppose we want to use a Boolean IR system to manage Shakespeare's plays. We might decide that useful units of retrieval (i.e. the documents) are the speeches, so in the example of Figure 1 everything between the first occurrences of `<SPEECH>` and `</SPEECH>` would be document 1, the speech by Ophelia would be document 2, etc. The operators of the Boolean algebra are `AND`, `OR` and `NOT`, and example Boolean expressions (queries) are: `honour AND (hamlet OR ophelia)`, or: `famous AND quote`.

Here we will use a mathematical notation that is close to the relational algebra notation, using set intersection $\cap$ for `AND`, set union $\cup$ for `OR` and set difference $\setminus$ for `NOT`. As in relational algebra, we will make select statements explicit, so the query `hamlet` which means "select all documents about hamlet", will get an explicit select operator σ: $\sigma(\texttt{'hamlet'})$.

Some properties of Boolean algebra

Equations 1 and 2 show some properties of Boolean algebra. It is good to realise that there are many expressions/queries that give exactly the same results. However, some expressions might require more processing power and more memory requirements for intermediate results when executed in a system. Avoiding expressions that require a lot from the system is called *query optimization* in relational database systems. For Boolean IR systems however, query optimization is hardly an issue.

$$(\sigma(\texttt{'a'}) \cap \sigma(\texttt{'b'})) \cup \sigma(\texttt{'c'}) \equiv (\sigma(\texttt{'a'}) \cup \sigma(\texttt{'c'})) \cap (\sigma(\texttt{'b'}) \cup \sigma(\texttt{'c'})) \quad (1)$$

$$(\sigma(\texttt{'a'}) \cup \sigma(\texttt{'b'})) \cap \sigma(\texttt{'c'}) \equiv (\sigma(\texttt{'a'}) \cap \sigma(\texttt{'c'})) \cup (\sigma(\texttt{'b'}) \cap \sigma(\texttt{'c'})) \quad (2)$$

Properties like the above might be important for another reason. When extending algebras to enable ranked retrieval, it is problematic that expressions that give exactly the same matching results, are producing different document rankings. If we base a query optimizer on the properties of the original algebra, then the system would produce different rankings depending on the query plan chosen. Well-known extended Boolean models [16] and fuzzy set models [14] show exactly this kind of unpredictable behaviour. We will continue this discussion in Section 2.3, but first we will introduce an algebra that provides us with the kind of data independence we described above.

2.2 Region algebras for information retrieval

Region algebras [2, 5, 6, 12] are an extension of Boolean algebras that reason about arbitrary parts (regions) of textual data. Region algebras model textual data as a linearized string of tokens: Instead of assigning nominal document identifiers to documents (as done by the Boolean algebra), region algebras assign ordinal identifiers to each token in the database. This is shown in Figure 2 for the first part of our example database.

```
<SCENE>1
  <STAGEDIR>2Enter3 HAMLET4</STAGEDIR>5
  <SPEECH>6
    <SPEAKER>7HAMLET8</SPEAKER>9
    <LINE>10So,11 let12 me13 say14the15famous16quote17
     again.18<LINE>19to20be,21or22not23to24be25</LINE>26
     that's27 the28 question.29
    </LINE>30
    <LINE>31
      :
```

Figure 2: Position numbering of example data

Numbering schemes for XML data have been studied extensively (see e.g. [8, 9, 17]). The above scheme combines a so-called stretched pre-post numbering scheme [9] with a positional IR index [18]. A region is defined by a starting position and an ending position. For instance, region 10,30 defines the first line, and region 20,25 specifies the text "to be, or not to be". In the remainder, the starting position is called *pre* and the ending position is called *post*.

The operators `containing` and `contained_by`, for which we will use the symbols $>$ and $<$, are essential for reasoning about regions. The expression $A > B$ returns all regions from A that contain at least one region from B. The expression $A < B$ returns all regions from A that are contained by a region from B. Both operators can easily be defined by conditions on the region's *pre* and *post*. Section 3 will define them more formally.

Some properties of region algebras

Equation 3 shows the region algebra expression of the Boolean query `a AND b`, that is, select all documents (`<doc>`) that contain both `'a'` and `'b'`. Note that instead of the `<doc>` tag, we could have specified a query on any other XML element. Equation 4 shows the region algebra expression of the Boolean query `a OR b`.

$$(\sigma(\texttt{'<doc>'}) > \sigma(\texttt{'a'})) \cap (\sigma(\texttt{'<doc>'}) > \sigma(\texttt{'b'})) \equiv \sigma(\texttt{'<doc>'}) > \sigma(\texttt{'a'}) > \sigma(\texttt{'b'}) \quad (3)$$

$$(\sigma(\texttt{'<doc>'}) > \sigma(\texttt{'a'})) \cup (\sigma(\texttt{'<doc>'}) > \sigma(\texttt{'b'})) \equiv \sigma(\texttt{'<doc>'}) > (\sigma(\texttt{'a'}) \cup \sigma(\texttt{'b'})) \quad (4)$$

Both expression can be simplified as shown. It is interesting to note that properties of the union $\cup$ are no longer similar to those of the intersection $\cap$, as is the case in Eq. 1 and 2.

Related work

The algebra described above is similar to the region algebra proposed by Burkowski [2]. Burkowski implicitly distinguishes mark-up from content. In later publications Clark

et al. [5] and Jaakkola et al. [12] describe region algebras that do *not* distinguish mark-up from content. In their algebra, σ('<doc>') returns all regions that start and stop on the position of the opening `doc` tags, and σ('</doc>') returns regions that start and stop on the position of the closing `doc` tags (i.e. tags and words are not treated differently). In order to get all speeches from the database, their algebras use the `followed_by` operator (denoted here as $\rightarrow$). The expression $A \rightarrow B$ returns all regions that start with some region from A and end with some region from B. The following query would retrieve all speeches that contain the word 'hamlet'.

$$\sigma(\texttt{'<speech>'}) \rightarrow \sigma(\texttt{'</speech>'}) > \sigma(\texttt{'hamlet'}) \quad (5)$$

It is easy to show that, by the definition of the $\rightarrow$ operator given above, the expression $A \rightarrow B$ will return a large number of regions (there are 6 regions that satisfy σ('<speech>') $\rightarrow$ σ('</speech>' in Figure 1). Clark et al. [5] limit the number of regions by putting an additional constraint on sets of regions. Each set of regions must be a "generalized concordance list", that is, a set of regions for which no region is nested in another region in the same set. Their approach cannot handle nested data very well. For instance, because of nested `<line>` tags, the following query will not match any of the lines in Figure 1, although the first line (region 10,30) obviously matches the query.

$$\sigma(\texttt{'<line>'}) \rightarrow \sigma(\texttt{'</line>'}) > \sigma(\texttt{'quote'}) \quad (6)$$

Jaakkola et al. [12] limit the number of regions by defining $A \rightarrow B$ such that A and B define nested regions. This solves the problem above, but introduces a similar problem if A and B are arbitrary text regions. For instance, one would expect the following query to match region 20,25 ("to be, or not to be", see Fig. 2), but their system will not retrieve it.

$$\sigma(\texttt{'to'}) \rightarrow \sigma(\texttt{'be'}) > \sigma(\texttt{'not'}) \quad (7)$$

We believe that the examples above show it is important to distinguish mark-up from content explicitly, that is, the system should know which tokens in the linearized string (see Figure 2) are mark-up, and which are textual units.

Another problem of region algebras is the fact that they cannot deal with direct inclusion [6], that is, the the `parent` and `child` access steps of XPath are not easily defined as manipulation of regions. Section 3 shows some extensions of the algebra that deal with this.

2.3 Requirements for ranked retrieval

Ranking of search results is important for systems that manage textual data. Users are interested in text containing certain *information*, which is not necessarily text containing a specific word or textual data item. In order to reason about ranking of search results, we extend the region algebra to a probabilistic region algebra, where a probabilistic region is defined by the triple *pre*, *post* and *probability*. When a ranking is needed, regions are sorted by their probability (or score). The probability/score reflects how well the found element (region) matches the query.

We distinguish three types of approaches to extending region algebras with score functions for ranked retrieval. These three types of approaches are also applicable to e.g. extending XQuery and XPath with score functions for ranked retrieval as described by Buxton and Rys [3].

1. **Orthogonal** Ranking and matching are defined (and possibly also implemented) as black box functions, where one does not affect the other.
2. **Matching Consistent** The properties of the original algebra are partially preserved. Some matching semantics of the original operators are preserved, e.g. a score of 0 being the same as no match, but other properties are not. Rewriting a query expression might produce a different ranking of regions (nodes).
3. **Ranking Consistent** The properties of the original algebra (see e.g. Equations 3 and 4) are fully preserved, and the semantics of the score are completely defined. If a query expression can be rewritten to another query that retrieves the same region sets (or node sets), then it will also define the same ranking of regions (nodes).

We aim at developing a Ranking Consistent extension of XQuery / XPath. This will allow us to benefit from algebraic properties for query optimization, without the danger of producing different rankings for different query plans. Furthermore, we believe that a Ranking Consistent extension will produce the best search quality achievable on the basis of the data, but of course, this has to be evaluated empirically. We will follow a language modeling approach to IR [10] for developing extensions, because this approach defines the semantics of document scores quite clearly.

Note that the World Wide Web consortium aims at extensions of XQuery / XPath that are Orthogonal or possibly Matching Consistent [3]. Algorithms for ranking will be mostly implementation-defined. These are probably realistic goals considering the current status of theory in information retrieval and database research, and considering the wish of industry to include their own (proprietary) ranking algorithms.

3. XML STORAGE AND QUERY ALGEBRA

The logical query algebra developed with the aim of preserving the ranked retrieval properties is described in this section. We begin with the explanation of the relational model used for storing XML collection in a database. The information about the tree structure of XML documents is stored in Region Index (RI) relational table, shown in Table 1, where we used information obtained from Figure 2. This table is formed using the XML mark-up information. The key of the relational table used for binding this basic relational table to all other relational tables describing XML structure and content (see text below), is a unique *pre* index.

In Table 1, *pre* and *post* attributes are positive integer values assigned to an XML node using the stretched pre-post indexing scheme explained in subsection 2.2. The column identified as *par* is used for fast access to parent and child nodes of a context nodes, thus enabling the direct inclusion expressions. It serves as a foreign key and represents the key *pre* index of a parent node.

In addition to the three columns which represent indexes obtained during the XML tree traversal, the columns *type* and *name* represent the information about XML mark-up. Therefore, *type* information denotes whether the entity is an element node (*'node'*), a text node (*'text'*), a processing instruction node (*'proc_ins'*), or a comment node (*'comment'*). The *name* column is used for element and processing instruction nodes only, and is defined by element name or processing instruction name, as specified by mark-up.

The information about element attributes is stored in a table with columns for the pre (*attr_pre*) and post index (*attr_pst*), a pointer to the element node which has an attribute (*attr_own*), the attribute name (*attr_name*), and the attribute value (*attr_val*). For the storage of textual information an extra relational table is introduced. The table

pre	post	par	type	name
1	3924	-	node	SCENE
2	5	1	node	STAGEDIR
3	4	2	text	-
6	62	1	node	SPEECH
7	9	6	node	SPEAKER
:				

Table 1: Region index relational table

pre	word
3	enter
4	HAMLET
8	HAMLET
11	So
:	

word	prob
again	0.000866
and	0.00528
arms	0.000328
be	0.000456
:	

Table 2: Word location index and word statistics

describes Location Indexes (LI) and is depicted in in Table 2. Here the *pre* identifier represents the unique location index of a word.

Additional tables (not shown here) are used to store the information about processing instruction and comment nodes. In these tables we introduce two columns, namely *p_i* and *comm*, which represent the target values used in processing instructions and the textual information that defines comments' content, respectively. Thus, the result is the existence of two relational tables of following type: *(pre,p_i)* for processing instruction nodes, and *(pre,comm)* for comment nodes. Note that the exact name of processing instruction is stored in the region index table.

Furthermore, since our goal is to enable IR on XML collections we introduced a relational table that stores word background probabilities. It is formed during the XML collection loading procedure, and is stored along with all the other information in the XML database. The background probability relational table has the form of *(word,prob)* and is depicted in Table 2.

For the actual storage of XML data we have chosen the Monet database kernel [1] since it gives promising opportunities for efficient XML storage and querying. It is a fact that main memory-oriented database systems, like Monet, outperform traditional DBMSs by using CPU and cache optimized execution. Furthermore, Monet can be easily extended with new data types and operators. A good example of the power of Monet is the staircase join algorithm aimed at speeding-up the execution of descendant and ancestor XPath axis steps which is implemented as a Monet extension (for details see [9]).

The Monet database kernel stores the data in Binary Association Tables (BATs), and all the operations are executed on BATs. BATs are relational tables with a head (key) and a tail column only. This fits very well in our relational structure, since almost all tables have *pre* as a key value (except *(word,prob)* table). Thus, we are able to split all tables that have more than two columns into a number of tables that have the form $BAT(pre, tail)$, where *tail* can be one of:

tail = {*post, par, type, name, word, attr_pre, attr_pst, attr_own, attr_name, attr_val, p_i, comm, prob* }

3.1 Query Algebra Domain

For defining query algebra operators on the logical level of an XML database we will first set the domain. The domain is defined following the relational model on physical level, explained in previous subsection. Based on relational model we identified four different sets on the logical level: the region index set - R, the Basic Index (BI) set - B, the word location set - L, and the background probability set - P. Each of these sets contains a specific type of information about XML elements or XML elements' content, with respect to the data stored in relational tables. Thus, the information about XML elements ($r \in R$) stored in the region index table with additional *prob* information entity that stores the probability of a current region in a probabilistic database model defines the RI set: r = *(pre,post,par,type,word,prob)*. Word location indexes ($l \in L$) and word background probabilities ($p \in P$) are defined based on their relational counterparts: $l = (pre, word)$ and $p = (word, prob)$. Basic indexes ($b \in B$) are an abstraction of LIs used to define text regions with a certain probabilistic values assigned to them: b = *(pre,post,prob)*.

In these definitions, each attribute of the index structure has its specific domain. Thus, attributes *pre*, *post*, and *par* are integer numbers. The *type* attribute is a subset of a character set that represents the type of a node in XML (e.g. *'a'* for attribute). Attribute *word* is a set of strings representing words or processing instruction, attribute, and tag names. The probabilities (foreground and background) are stored as real numbers ranging from 0.0 to 1.0 using the *prob* attribute.

Considering all the information present in the tables from the previous subsection, we can consider the region index to be the basic entity for our new region algebra. Using the RIs as a basic construct in our algebra together with BIs and LIs as its subsets, we are able to address arbitrary parts (elements) of an XML tree (collection) or arbitrary terms in each text node. Furthermore, various operators can be introduced that will stay in the domain of region or basic indexes with respect to the operand types and used operators. These operators will be defined in next subsection, while the core implementation of algebraic operators will be explained using an example in the next section.

3.2 CIRQAl

The four sets that define the domain of the algebra are used for the definition of all the operators in the Complex Information Retrieval Algebra (CIRQAl) on the logical level of a database. Algebraic operators are defined to enable execution of CIRQuL queries whose syntax is described in [13]. Given the high complexity of CIRQuL queries, here we will explain only the most important operators that in the same time form the core of the logical algebra and enable the execution of a great variety of queries over XML documents.

The basic framework for describing CIRQAl will be introduced using the XPath properties and specifications according to the XPath 1.0 standard [4]. In general, the XPath language can be defined as a sequence of consecutive steps starting from the root node, which guide us through the XML tree structure. Furthermore, after introducing context elements, i.e. the resulting set of context nodes obtained after the execution of the previous XPath step, we can define each XPath step independently. We will use C to denote the set of RIs that represents the whole XML collection, and R_c to denote the set of current context elements. Thus, we can define each XPath expression in a functional manner using the following abstract step operations:

$XPath(root, C) := root \circ step_1(C) \circ step_2(C) \circ ... \circ step_n(C)$, where each step operation is executed on the result set from the previous one ($step_i(R_c, C)$). According to the XPath specification [4] each XPath step can be subdivided into *axis*, *node-test*, and *predicate* part. Therefore, the step operation can be further decomposed into basic operations used for

axis operator	result
self(Rc,C)	R_c
child(Rc,C)	$\{s:C; r:R_c \mid s.pre = r.par \bullet s\}$
parent(Rc,C)	$\{s:C; r:R_c \mid s.par = r.pre \bullet s\}$
descendant(Rc,C)	$\{s:C; r:R_c \mid r.pre < s.pre < r.post \bullet s\}$
descendant_or_self(Rc,C)	$\{s:C; r:R_c \mid r.pre \leq s.pre \leq r.post \bullet s\}$
ancestor(Rc,C)	$\{s:C; r:R_c \mid s.pre < r.pre \wedge s.post > r.post \bullet s\}$
ancestor_or_self(Rc,C)	$\{s:C; r:R_c \mid s.pre \leq r.pre \wedge s.post \geq r.post\}$
preceding(Rc,C)	$\{s:C; r:R_c \mid s.post < r.pre \bullet s\}$
preceding_sibling(Rc,C)	$\{s:C; r:R_c \mid s.post < r.pre \wedge s.pre = r.par \bullet s\}$
following(Rc,C)	$\{s:C; r:R_c \mid s.pre > r.post \bullet s\}$
following_sibling(Rc,C)	$\{s:C; r:R_c \mid s.pre > r.post \wedge s.par = r.pre \bullet s\}$
attribute(Rc,C)	$\{s:C; r:R_c \mid s.type = \text{'a'} \wedge r.pre = s.pre \bullet s\}$
namespace(Rc,C)	$\{s:C; r:R_c \mid s.type = \text{'s'} \wedge r.pre = s.pre \bullet s\}$

node-test operator	result
qname(Rc,word)	$\{s:R_c \mid s.word = word\}$
star(Rc,type)	$\{s:R_c \mid s.type = type\}$
text(Rc)	$\{s:R_c \mid s.type = \text{'t'}\}$
comment(Rc)	$\{s:R_c \mid s.type = \text{'c'}\}$
p_i(Rc)	$\{s:R_c \mid s.type = \text{'p'}\}$
p_i(Rc,target)	$\{s:R_c \mid s.type = \text{'p'} \wedge s.word = target\}$
node(Rc)	$\{s:R_c \mid s.type = \text{'n'}\}$

Table 3: Axis and node-test operators

defining *axis*, *node-test*, and *predicate* parts:

$step_i(R_c, C) := R_c \circ axis_i(C) \circ node\_test_i \circ predicate_i(C)$

Axis and node test steps can be defined using the introduced domain sets and the operator notation, and we will specify them in the logical algebra using the operators with the same names as in the XPath standard. Axis and node test operators are described in Table 3, where R_c represents the *context node-set* ($R_c \subset C$), and s ($\in C$) is the element of the *result node-set* after the selection[1]. We can exert that the operators *descendant* and *ancestor* have the same functionality as the operators that define the containment relation in region algebra. Therefore, CIRQAl can be considered as a more general algebra than region algebra, since it contains information about the tree structure organization of documents (not just on two dimensions like in region algebra approaches). Moreover, CIRQAl includes the probabilistic calculus in defining some of the operators, as we will see below.

Along with the 13 standard XPath operators used for XML tree traversal (axis operators) we introduced seven operators for type and name node tests (defined based on the XPath specification [4]). Two of them are used for node tests by name (*qname* and *star*) and five for node tests by type (*text*, *comment* *p_i*, and *node*). All these operators are defined in Table 3. Here we will not define operators for predicate due to their number and complexity, we will only introduce operators for the IR extension to XPath instead.

IR operators are introduced to enable ranking and ranked retrieval of XML elements or documents. As a starting point for defining IR operators we considered the proposal for a new query language for IR on XML databases, called CIRQuL [13]. As we already stated we will explain only the basic operators, which are defined in Table 4. These operators are explained in more details in next section's example.

In Table 4 we used *ind* as a shorthand notation for region index attributes, defined as follows:

$ind = \{pre, post, par, type, word\}$,

and two additional operators, namely $\triangleright$ and $|\ |$ (size of a region), which are defined in next equations:

$$r \triangleright B := \{b:B \mid b.pre > r.pre \wedge b.post < r.post \bullet b\} \quad (8)$$

$$|x| := x.post - x.pre + 1 \quad (9)$$

Using the operators defined in Table 4 we are in position to express numerous IR tasks. Together with the algebraic operators for axis and node test steps we are able to form the foundation for the complex information retrieval on the XML database consisting of an arbitrary number of indexed XML documents.

[1] A notation like $\{s:C; r:R|Pred \bullet s\}$ denotes the set of all s, when s ranges over C and r over R, constrained by the predicate $Pred$ that must hold [20].

4. EXECUTING COMPLEX QUERIES

In this section we will explain the transformation from CIRQuL query to core database functions using the execution plan on logical level. The XML data presented in the first section will be used for describing the execution of a query on top of the physical level of a database (Monet). To query the data we will take the next query example:

```
//speech[IR(('famous' adj 'quote')[0.8] and be[0.2])]
```

If we consider logical algebra operators given in the previous section we can come up with the execution plan that is given in Figure 3. In defining the execution plan, we distinguished between four different expression levels, where each level corresponds to an expression of different complexity in CIRQuL. Furthermore, we can distinguish between the XPath expression part, which is a part responsible for the execution of plain XPath steps (axis and node test steps), and the IR expression that describes the execution of IR functions, for which we differentiate between three layers: basic, region, and complex expression layer.

The basic expression layer is responsible for finding the location (the *pre* index) of a selected word (*interval* operator), a group of words (*or_term* operator), or the phrase region formed using the adjacent (*adj* operator) or near (*near* operator) operation on terms. These operators work on the domain of BI sets, where input parameters for *interval* operators are the term location set (L) and the set of background probabilities of terms (P). The resulting set is again in the domain of BIs.

Region expression operators are aimed at computing the probability of a term (or phrase expression) in a complex region, as defined with the *probab* operator in Table 4. Furthermore, the 'importance' parameter is resolved here using the *scale* operator, which scales the significance (importance) of a single basic expression in a complex one. All these operators work on the RI domain, while *probab* operator takes as an input two sets, one RI set that is a result of XPath expression operators, and one BI set that is a result of application of basic expression operators.

Finally, complex expression operators combine the probabilities on regions in region expression and give the result which is ranked according to the probabilistic values of RIs. Thus as a resulting set we obtain a set of RIs which is ordered in descending order with respect to the *prob* attribute.

Since we used Monet database system as a database platform, the prototype implementation of execution engine over XML database was done in Monet Interpreter Language - MIL. The MIL algebra consists of numerous operators which

operator	Definition
interval(term)	$\{b : B; l : L; p : P \mid l.word = term \wedge p.word =' term' \wedge b.(pre, post) = l.pre \ \wedge b.prob = p.prob \bullet b\}$
or_term(B1,B2)	$\{b : B; b_1 : B_1; b_2 : B_2 \mid (b.(pre, post) = b_1.(pre, post) \vee b.(pre, post) = b_2.(pre, post)) \wedge$ $b.prob = b_1.prob + b_2.prob \bullet b\}$
adj(B1,B2)	$\{b_1 : B_1; b_2 : B_2 \mid b_2.pre = b_1.post + 1 \bullet (b_1.pre, b_2.post, b_1.prob \times b_2.prob)\}$
near(B1,B2,n)	$\{b : B; b_1 : B_1; b_2 : B_2 \mid b_2.pre \geq b_1.pre - n \wedge b_2.post \leq b_1.post + n \wedge$ $b.pre = min(b_1.pre, b_2.pre) \wedge b.post = max(b_1.post, b_2.post) \wedge b.prob = n \times b_1.prob \times b_2.prob \bullet b\}$
probab(R,B)	$\{s : R; r : R; b : B \mid (s.ind = r.ind \wedge s.prob = \lambda \frac{\#(r \triangleright B)}{\lvert r \rvert} + (1 - \lambda) b.prob \bullet s\}$
scale(R,imp)	$\{s : R; r : R \mid (s.ind = r.ind \ \wedge s.prob = r.prob \times imp \bullet s\}$
or(R1,R2)	$\{s : R; r_1 : R_1; r_2 : R_2 \mid (s.ind = r_1.ind \vee s.ind = r_2.ind) \wedge s.prob = r_1.prob + r_2.prob \bullet s\}$
and(R1,R2)	$\{s : R; r_1 : R_1; r_2 : R_2 \mid s.ind = r_1.ind \wedge s.ind = r_2.ind \wedge s.prob = r_1.prob \times r_2.prob \bullet s\}$
order(R)	for any $r_1, r_2 \in R : r_1 \prec r_2 \Leftrightarrow r_1.prob > r_2.prob$

Table 4: CIRQAl operator definitions

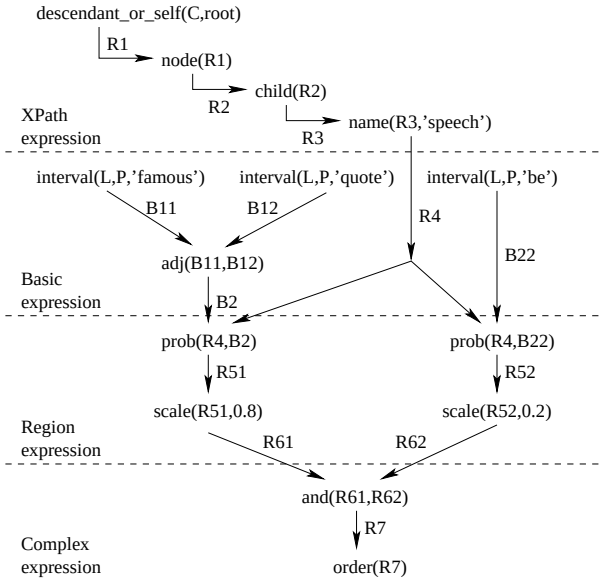

Figure 3: Query execution plan

are defined over simple variables and over sets. Furthermore, it can be easily extended with new user-defined operators. Using built-in and user defined operators we were able to perform straightforward mapping from logical algebra to MIL operators. Moreover, we used the string notation for defining operators in query execution plan given in Figure 3 to exert the simplicity of transforming algebraic operators on logical level on MIL operators that work on Monet sets and tuples. Thus, for each operator in the logical algebra we made a corresponding MIL operator (or two in some cases) that perform the same operation as defined by logical operators. The result of transforming execution plan for previous query into MIL code is depicted in Figure 4.

```
R4:=root.descendant_or_self.node.child.qname('speech');
R4.init_IR;
  R51:=R4.prob(interval('famous').adj(interval('quote')));
  R61:=R51.scale(0.8);
  R62:=R4.prob(interval('be')).scale(0.2);
  R7:=R61.and(R62);
RR:=end_IR(R7);
```

Figure 4: MIL operations for the query plan

5. DISCUSSION

The current prototype system supports most functionality of the query language proposed in [13] and the W3C requirements for full-text search [3]. The system will be tested on the INEX 2003 data [11]. At the moment, the system is not fast enough for on-line, interactive, applications. This is partly due to inefficient query plans, and partly due to inefficient definition/implementation of single operators. In the near future, we will speed-up the system, and we will redefine (some) operators such that the probabilistic algebra will form a Ranking Consistent extension of the XPath version of the algebra.

6. REFERENCES

[1] P.A. Boncz. Monet - A Next-Generation DBMS Kernel for Query-Intensive Applications. PhD thesis, U. of Amsterdam, 2002.

[2] F.J. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In *Proceedings of ACM SIGIR'92*, pp. 112–124, 1992.

[3] S. Buxton and M. Rys. XQuery and XPath full-text requirements. Technical report, W3C, 2003. http://www.w3.org/TR/

[4] J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0 TR, W3C, 1999, http://www.w3.org/TR/xpath.

[5] C.L.A. Clarke, G.V. Cormack and F.J. Burkowski. An algebra for structured text search and a framework for its implementation. *Computer Journal* 38:43–56, 1995.

[6] M.P. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of ACM PODS'95*, pp. 11–22, 1995.

[7] C.J. Date. *An Introduction to Database Systems, 6th edition.* Addison-Wesley, 1995.

[8] D. Florescu and D. Kossmann. A performance evaluation of mapping schemes for storing XML data in a relational database. Technical report, INRIA, 1999.

[9] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems (TODS)*, 2003.

[10] D. Hiemstra. *Using language models for information retrieval.* PhD thesis, University of Twente, 2001.

[11] Initiative for the Evaluation of XML Retrieval. April 2003 - December 2003, http://www.is.informatik.uni-duisburg.de/projects/inex03/.

[12] J. Jaakkola and P. Kilpeläinen. Nested text-region algebra. Technical report, University of Helsinki, 1999.

[13] V. Mihajlović, D. Hiemstra, P.M.G. Apers. CIRQuL - Complex Information Retrieval Query Language. In *Proc. of the VLDB 2003 PhD Workshop*, 2003.

[14] C.P. Paice. Soft evaluation of Boolean search queries in information retrieval systems. *Information Technology: Research and Development*, 3(1):33–42, 1984.

[15] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, 1979.

[16] G. Salton, E.A. Fox and H. Wu. Extended Boolean information retrieval. *Communications of the ACM*, 26:1022–1036, 1983.

[17] A.R. Schmidt, M.L. Kersten, M.A. Windhouwer and F. Waas. Efficient relational storage and retrieval of XML documents. In *The WWW and Databases*, Springer-Verlag, pp. 137–150, 2000.

[18] F. Scholer, H. Williams, J. Yiannis and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of ACM SIGIR'02*, pp. 222–229, 2002.

[19] W. Shakespeare. *Hamlet.* 1600. http://www.ibiblio.org/bosak

[20] J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, 1992.