The Simplest Query Language That Could Possibly Work

Richard A. O'Keefe
Department of Computer Science
University of Otago
Dunedin, New Zealand

ok@cs.otago.ac.nz

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand

andrew@cs.otago.ac.nz

ABSTRACT

The INEX'03 query language proved to be much too complicated for the INEX participants to use well, let alone anyone else. We need something simpler, but not too simple. Something which is basically a hybrid between Boolean IR queries and a stripped down CSS will do the job.

1. INEX NEEDS A QUERY LANGUAGE.

In the INEX conferences, we are trying to develop a data collection and a set of queries with known answers that can provide a solid basis for research and experimentation with XML information retrieval.

In order to communicate between researchers in the same year, we need a common query language. For INEX'02 there was such a language. In INEX'03 there was another. In order to communicate between the researchers who produce the queries in one year and the researchers who use them in later years, we need a stable, well-defined language.

The designer(s) of the INEX'03 query language had every reason to feel pleased. After the INEX'02 query language proved to need revision, surely this was the simplest thing that could possibly work: take an extremely well established XML structural query language (XPath) and add to it a minimal set of features for Information Retrieval.

It seems to be agreed that XPath is not a language for the casual user. But this paper is not concerned with user query languages. The query language we need is a query language for use by researchers who are expert in information retrieval and XML. What counts is whether the query language is suitable *for us*, not users.

Unfortunately, the production of this year's CAS queries proved conclusively that the INEX'03 query language is far too complicated *for us*:

- It proved too hard to use. Of the 30 CAS queries that were selected, 19 (nearly $\frac{2}{3}$), were either syntactically illegal or otherwise wrong. It took no fewer than 12 rounds of correction before we had a completed collection of queries.

- Like many W3C productions, XPath 1.0 is quirky, to put it kindly. It is very powerful in some respects, but there are queries that are very hard to express. For example, *//body//ip1//name | //body//ip2//name* is

legal, but *//body//(ip1|ip2)//name* is not.

- It proved to be hard to implement. Presumably everyone who submitted a query for consideration had already checked it with some XML IR engine; how else could they have known that the query had about the right number of relevant answers? Yet a large number of queries were syntactically or semantically wrong. That should have been noticed. At least one implementor switched the semantics of the / and // operators.

- It proved to be hard to implement for another reason. XPath is quite powerful, in ways that are not likely to be useful for information retrieval, and yet if XPath was not implemented in full, were we really implementing the INEX'03 query language? This year, it turned out that most of the power of XPath was not needed. It wasn't the simplest thing that could possibly have worked. For example, we[23] found that there were 198,041 nodes in the index after ignoring "noise" tags. Yet if ordinal position was also ignored, there were only 10,522 distinct paths. Not one of this year's selected CAS queries used the ordinal position ([n]) feature of XPath.

- XPath has a clear definition of the "string value" of a node; the definition is precise, but given the actual XML markup in the document collection we are working with, it's not the definition we want. For example, if there is one mention of Joe Bloggs in the collection, as ⟨au⟩⟨fnm⟩Joe⟨/fnm⟩⟨snm⟩Bloggs⟨/snm⟩⟨/au⟩, then the string value is "JoeBloggs" and a search for the word "Bloggs" is guaranteed to miss it.

  Worse, markup that is supposed to enclose numbers very commonly includes punctuation as well; the rules of XPath say that trying to convert such a string value to numeric form is an error. Yet we want to query it.

2. THE INEX'03 QUERY LANGUAGE WAS TOO HARD TO USE.

Every group had to submit 3 CAS and 3 CO queries. These submissions were supposed to have been tested, and known to have a reasonable number (not too high, not too low) of relevant answers. In fact, some answers were provided with each submission. So each submitted query should have been a legal INEX'03 query.

From this pool, 30 CAS and 36 CO queries were selected. Of the 30 CAS queries, 19 had either syntax errors or serious semantic errors. The most common semantic error was using the "child" operator / when the "descendant" operator // was intended.

This is a shocking error rate.

It wasn't just hard to get the queries right in the first place; it was hard to fix them. It took 12 rounds of corrections before we had a workable set of queries, starting from what were presumably the best queries in the first place.

Since a query language based on XPath 1.0 was too hard for us to us, it is impossible to believe that a query language based on the much more complicated XPath 2.0 could be usable by us.

3. WHAT SHOULD WE LOOK FOR IN A QUERY LANGUAGE?

3.1 We want something WE can use.

This paper is not about query interfaces or query languages for end users. This paper is solely concerned with query languages for *researchers* producing or using INEX data. Complexity is not necessarily a problem *for us*, as long as it is useful complexity. Requiring an intimate knowledge of XML or XML related technologies is not necessarily a problem *for us*. Requiring lots of punctuation in just the right places is not necessarily a problem *for us*.

While complexity need not be a problem, we need to take a step back and start with something much simpler than XPath, because it is an empirically established fact that it was too complicated *for us*. It is not likely that the query language we propose in this paper will serve for all time; what does matter is that it should be possible to automatically translate it into whatever richer language may be devised in the future. Simplicity now means easier conversion in the future. So one guiding rule is that nothing should be included in the query language unless it was actually used in this year's or last year's queries.

We do not want to limit INEX participation to experimenters following an "orthodox line" in query languages. Keeping the query language simple keeps the conference open to approaches with as yet unimagined index structures and retrieval techniques. XPath and XPath-like languages penalise such approaches.

3.2 Databases and information retrieval are different.

It is useful to distinguish between *database* query languages and *information retrieval* query languages. They have some similarities, but the differences are fundamental, and mean that an XML database query language is unlikely to be a good foundation for an XML information retrieval query language.

The CODASYL database language, "network" databases, the relational algebra, the relational calculus, SQL, the Object Query Language (OQL) in the ODMG Object Database Standard[4], and various spatial and temporal extensions

of relational databases, even the Smalltalk dialect used in Gemstone, all have these fundamental characteristics in common:

- To a large extent, as [9] puts it, this "data is primarily intended for computer, not human, consumption."

- A "database" is made up of elementary values (numbers, strings, dates, and so on) aggregated using a pre-defined set of container types with precise data structure semantics and labelled with user defined labels (column names, relation names, and so on).

- The user-defined labels have user-defined semantics which the database is aware of only to the extent that constraints are stated.

- Even when there are user-defined structures (classes in ODMG, Gemstone, and SQL3, for example), these may be seen as instances of one of a fixed set of meta-structures. For example, the ODMG standard provides an Object Interchange Format by means of which any object database may be dumped as a text stream; instances of classes all have a fixed format here and it is clear that "class" is a single meta-structure.

- There is a structured query language with a (more-or-less) formal definition which relates any legal query to a precise semantics, by appealing to the data structure semantics of the container types and meta-structures and to any stated constraints.

- A query processor is expected to obey the semantics of any query it accepts *precisely*; it may exploit known properties of the query language to transform a query into one with better performance, typically by using indexes.

- If a query has more than one answer, *all* of the answers are relevant. Someone who doesn't want all of the answers is expected to write a more specific query.

Database query languages are just like programming languages. (Very bad programming languages, some of them, notably SQL.) The person formulating the query is expected to understand the relevant user-defined labels and constraints and to "program" a query which expresses his or her needs. A database query engine is required to obey the query literally, just as a C compiler is required to translate C faithfully, even rubbish. If you ask an ODMG database the OQL query *select p from Persons p where p.address.city = "Dunedin"* and the answer includes a *p* for which *p.address.city = "Mosgiel"*, you will be seriously unhappy, even though Mosgiel is only 10 to 15 minutes' drive from Dunedin.

Since SGML was designed, the SGML slogan has been "a document is a database". For many years there have been SGML document database engines, notably SIM[16]. As XML is a special case of SGML, it is natural to view an XML document as a database.

- The elementary values are strings. The aggregates are labelled attributed tree structures. The data structure

semantics is provided by GROVEs, or the DOM. Element type names and attribute names are the user defined labels.

- Constraints are stated by means of DTDs or XML Schemas. XML Schemas in particular express the notion "a database is a document".

What you get, on that view, is a database query language for tree-structured databases.

Information retrieval is very different. Instead of saying "the programmer knows precisely what s/he wants and how that's represented, I must do exactly what s/he says", information retrieval engines say "the user wants to find out about something and has given me a hint about what it is, I must be helpful". If you ask an information retrieval system *"agricultural research Dunedin"* and it comes back with a web page about *"Invermay Agricultural Centre, Mosgiel"*, you are not angry with it for disobeying you but impressed with how clever it was to find something so helpful.

The fact that information retrieval systems regard the user's query as a clue about what the user wants instead of a precise specification has enormous consequences for the design of information retrieval languages. So does the fact that the text they search is itself *not* in a precisely defined language.

When you construct a DTD or Schema for a family of XML documents, you describe how the XML parts fit together. But if you have free text in some of the elements, it remains just as informal as free text on its own.

At one end, we have data without a known precise semantics. At the other end, we have queries that are regarded as clues rather than commands. As Shlomo Geva[13] pointed out in the INEX mailing list, even the Boolean operators are not taken all that seriously by some retrieval engines. If two relational or object database engines holding the same information give different answers to a single query, at least one of them is broken. If two information retrieval engines holding the same document collection give different answers to a query, one of them might be better, but each of them might find something useful that the other doesn't. It certainly doesn't mean that either of them is wrong. All of this makes it hard to design elaborate information retrieval query languages. What earthly use is elaborate precise syntax when you don't have, can't have, and wouldn't want, precise semantics?

Of course we can embed a database query language in an IR query language (find precisely this set of documents and use that as a clue combined with the other clues in the query to find what I really want instead), and we can embed an IR query language in a database query language (give me precisely the answers satisfying a bunch of tests one of which is this clue about what I have in mind). Confusion seems unavoidable; at least we should be clear about which parts are precise and which parts are fuzzy.

3.3 It's all about indexes.
The great strength of Information Retrieval systems is their indexes.

An information retrieval language for XML should exploit this. It should avoid "structural" queries that are hard to handle with plausible index structures. This suggests keeping XML "structure" and IR "content" parts of queries separate, rather than mingling them indiscriminately as XPath does.

This does not mean that we should always be limited to queries that can be expressed in terms of currently known index structures. On the contrary, if someone comes across a reasonable query that is not expressible in the INEX'04 query language, that's a *good* thing, because it suggests a research topic: what kind of index could support this kind of query?

3.4 "Descendant" is more useful than "child".
An extremely common mistake in the INEX'03 queries was using the "child" axis (/) when the "descendant" axis (//) was intended.

The designers of CSS recognised that "descendant" queries were more common when they used the invisible operator to mean "descendant", making "descendant" easier to say than "child".

Consider *//article/bdy/sec/ip1*. That may be what you want, but you might have wanted *//article/bdy/sec/bq/ip1* elements as well, had you known about them. The query *//article//bdy//sec/ip1* is more likely to be what you really mean.

It turns out that *none* of the INEX'03 queries needs "child" at all; in each case "descendant" will do. This frees us to use the simple spelling "/" for "descendant", as many INEX contributors expected.

4. SOME XML QUERY LANGUAGES
The world is awash in query languages for semistructured data, ranging from the complicated (CSS) to the mindbogglingly complicated (XQuery).

4.1 HyTime
HyTime[15, 14, 21] introduced many important things to SGML. One of them was a query language, HyQ[19].

However, the current standard says "HyTime recommends the use of the Standard Document Query Language (SDQL), defined in the DSSSL standard, ISO/IEC 10179:1996 Document Style Semantics and Specification Language, for the queryloc and nmquery element forms. The SDQL language includes equivalents of all the HyTime location address forms."

Early drafts of XPath looked like a stripped down HyQ.

HyQ is all about precise location of points and ranges both in trees and in multimedia coördinate systems. It is quite complicated. But it is worthy of note as one of the two ancestors of most XML query languages. (The other is SQL.)

Because the query language presented here is not semantically like XPath, it would be highly undesirable for it to resemble XPath too much in syntax.

4.2 DSSSL

DSSSL[17] is the SGML version of XSL and XSLT[6]. It contains a Scheme-based query and transformation language. It must be said that DSSSL is incomparably easier to read than XSLT. The Standard Document Query language is basically some datatypes for collections of nodes and some functions that manipulate them. It's a programming language, not an IR query language.

4.3 CSS

A CSS[3] ⟨selector⟩ is a collection of ⟨path⟩s or-ed together. In each ⟨path⟩, the focus is on the rightmost element; it is that element which the following style will be applied to. Working from right to left, an element must be a sibling ('+'), a child ('>'), or a descendant (invisible operator) of the element to its left.

An ⟨element⟩ test may check for an element ⟨name⟩ or not (⟨any⟩ or omitted). It may check whether an element is the ':first-child' of its parent. This means that XPath's */*[3 and p]* is expressible as **:first-child+*+p*. But XPath's */p[3]* is not quite expressible; *p:first-child+p+p* does not allow other elements between the *p* elements.

A ⟨filter⟩ may check whether an attribute is present, whether it is present and has normalised value exactly equal to a given text, whether it is present and contains a given white space delimited word, or whether it is present, looks like an xml:lang value, and has a given lang code as prefix. The grammar is given in Table 1.

There is no negation anywhere in CSS. You cannot test whether an attribute is present and *not* equal to a string. Paths cannot be negated. Within its limits, CSS seems quite usable.

4.4 XPath

This year's query language was based on XPath 1.0. XPath 1.0 has several uses in W3C standards. One of them is XPointer. XPointer provides a means of pointing *precisely* to a location or range in a document. That is, XPointer, and the underlying XPath, are *database* query languages for XML.

We can get an idea of the complexity of various extensions and relatives of XPath by looking at the sizes of the defining reports; to master any of them requires reading at least this much material. Since the reports are provided in HTML, the page count depends on how you display it. Therefore we normalise the number of screens by the number of screens for XPath 1.0 in Table 2.

The "all up" entries include the Data Model and Functions and Operators documents, which are essential parts of XPath 2.0, XSLT 2.0, and XQuery 1.0. To get page counts for the browser and paper size we used, multiply by left column by about 28.

If XPath 1.0 was too complex for us to master, can any of the other W3C query languages be easier? XML-QL looks as though it might be, but it is not a W3C recommendation, and [9] explicitly says that "...we take a database view, as

Table 2: Length of Specification (Normalised)

0.5	CSS 2.0 selectors[3]
1.0	XPath 1.0[7]
0.7	XML-QL[10]
1.5	XQL[22]
3.2	XSLT 1.0[6]
4.2	XSLT 1.0 + XPath 1.0 (XSLT includes XPath)
2.4	XQuery 1.0 and XPath 2.0 Data Model[11]
5.8	XQuery 1.0 and XPath 2.0 Functions&Operators[20]
3.1	XPath 2.0[1]
11.3	XPath 2.0 all up
9.0	XQuery 1.0[2]
17.3	XQuery 1.0 all up
10.1	XSLT 2.0[18]
18.3	XSLT 2.0 all up

opposed to document view, of XML. We consider an XML document to be a database ...".

In fact all of these languages take a database view, making them unsuitable as foundations for an information retrieval query language. Space does not permit thorough discussion of YATL[8], XQL[22], Quilt[5] (Quilt and XPath 1.0 are closely related), YATL[8], or others.

4.5 XIRQL

XIRQL[12] was designed as an "information retrieval" query language, not a "database" query language. However, it extends XQL, so parts of it resemble XPath, including the distinction between "child" and "descendant" which we failed to master. In the INEX collection, it was not clear to most of us what the root actually was, so the ability to refer to the root is not useful to us either.

The abstract of [12] tells us that XIRQL integrates "weighting and ranking, relevance-oriented search, datatypes with vague predicates, and semantic relativism ... by using ideas from logic-based probabilistic IR models." This means that important and attractive as XIRQL is, it is too closely tied to one particular approach to be suitable for INEX.

We propose a much simpler and less capable language, which can be seen as a very small sublanguage of XIRQL, and also of other query languages.

5. THE STRING-VALUE PROBLEM

Practically everything in XPath 1.0 that involves strings is defined in terms of the "string-value" of a node. The rules are spelled out in section 5 of the XPath 1.0 specification. Roughly speaking,

1. The string-value of a text item (parsed character data or CDATA) is the obvious text value.

2. The string-value of an element or of the entire document is the concatenation of the string-values of its text descendants in document order.

3. The string-value of an attribute is its normalised value as spelled out in the XML 1.0 specification. (An XML processor that does not validate cannot be used as the basis for an XPath implementation.)

Table 1: CSS grammar

⟨selector⟩	::=	⟨path⟩ (⟨or⟩ ⟨path⟩)*
⟨or⟩	::=	','
⟨path⟩	::=	(⟨siblings⟩ ⟨down⟩)* ⟨siblings⟩
⟨down⟩	::=	'>' | empty
⟨siblings⟩	::=	(⟨element⟩ ⟨followed-by⟩)* ⟨element⟩
⟨followed-by⟩	::=	'+'
⟨element⟩	::=	(⟨name⟩ | ⟨any⟩ | ⟨filter⟩) ⟨filter⟩*
⟨any⟩	::=	'*'
⟨filter⟩	::=	⟨exists⟩|⟨equals⟩|⟨word⟩|⟨prefix⟩|⟨first⟩|⟨lang⟩
⟨exists⟩	::=	'[' ⟨name⟩ ']'
⟨equals⟩	::=	'[' ⟨name⟩ '=' ⟨value⟩ ']'
⟨word⟩	::=	'[' ⟨name⟩ '~=' ⟨value⟩ ']'
⟨prefix⟩	::=	'[' ⟨name⟩ '|=' ⟨value⟩ ']'
⟨first⟩	::=	':first-child'
⟨lang⟩	::=	':lang('⟨value⟩')'

So ⟨au⟩⟨fnm⟩Joe⟨/fnm⟩⟨snm⟩Bloggs⟨/snm⟩⟨/au⟩ has string-value "*JoeBloggs*".

If you go looking for "*Bloggs*" in ⟨au⟩, XPath 1.0 guarantees you won't find it.

Of course, we don't have to follow XPath's definition of string-value. But if we don't do that, there isn't much point in following XPath's complex and limiting syntax either.

This definition of string value goes back to HyTime; every XML-related standard we've checked uses essentially the same definition. CSS and XSLT provide means for transforming a document by adding material at the beginning or end of an element's contents; the string value can be quite different in the transformed document. XPath was too hard; bringing XSLT into it would clearly be inadvisable.

There are three plausible ways around this problem.

- Add an extra space at the end of each text item. This gives the answer "Joe␣Bloggs␣", which will work. In rare cases like "⟨u⟩A⟨/u⟩ccelerator" this may break words up, but it will almost always help.

- For items which should be treated as having word breaks, add an attribute in the DTD:

  ```
  <!ATTLIST snm INEXword #FIXED "break">
  ```

  Ensure that there is at least one white space character at the boundaries of every element with INEX-word="break".

- Allow the indexing software to make the decision just as it does for stemming. Attributes like INEXword offer guidance, not rigid command.

The first approach is simpler. If we were seeking the precision of database queries, the second approach would be better. Examples like T⟨scp⟩itle⟨/scp⟩ W⟨scp⟩ords⟨/scp⟩ may make it essential even for us (although the INEXscan attribute should solve this problem). But whichever approach we take, we are divorcing ourselves from XPath.

5.1 Numbers

An XML document contains only strings. Many of this year's queries involved numeric comparisons. That requires converting strings to numbers. XPath specifies precisely how that is done. (The rules are somewhat different in XPath 2.0, but do not affect the present point.)

The problem is that the INEX'03 document collection is a realistic collection of sloppily marked up text. There are elements such as ⟨yr⟩ which are supposed to contain numbers, but also contain punctuation marks and other junk. Trying to convert such a string to a number is an error in XPath. If we want to know whether $yr > 1999$, we do not want our query to be derailed by ⟨yr⟩2000,␣⟨/yr⟩, as it *must* be in XPath.

Not only do we need rules for converting text to numbers that are different from the rules in XPath, we need to interpret comparisons fuzzily. If you ask a database for a record with $date > 1999$ and it reports a record with $date = 1999$, that's an error. If you ask an information retrieval system for documents with $yr > 1999$ and it returns one with $yr = 1999$, that's not an error, it's just somewhat less relevant than one that matches the clue precisely.

6. ARCHITECTURAL FORMS

HyTime was really several interesting standards packaged together. One of the key features presented was the idea of "architectural forms" and of architectural form processing.

Basically, the idea is that a document may be marked up (and validated) according to one DTD, yet processed according to another (traditionally but confusingly called a) meta-DTD. Attributes in the source DTD say how to map the elements and attributes physically present to the ones that ought to be present according to the target DTD. A processing instruction with a special form is used to tell an architectural-form-aware processor which attributes to use for this purpose.

This may sound like XSLT, or, if you are into arcana, like linkage declarations in SGML. In fact it is something much simpler. Elements and attributes may be dropped, renamed, or copied as they are.

Why would you parse in one DTD and process according to another? You might have a formatter that can handle many structures, and a specialised DTD that is only intended to use some of the features. You might have a meta-DTD written using English words for markup, and Swedish users who would like to use Swedish words, so they validate against a DTD which uses Swedish words, but which uses architectural form processing to map to the English version. You might wish to make fine distinctions; for example you might want to use ⟨species⟩ and ⟨foreign⟩ tags in your markup, but they might both be simply mapped to ⟨italic⟩.

With the INEX collection, we have a collection of documents marked up for printing. Some of the distinctions made in the DTD are not important for information retrieval purposes. The INEX'03 rules took this into account. For example, ⟨ip1⟩, ⟨ip2⟩, ⟨ip3⟩, ⟨ip4⟩ were all to be treated by the query engine as equivalent to ⟨p⟩.

That's the wrong time to do it. It had the unpleasant consequence that you asked for p[n] the element you got could be p[m] with $m \neq n$.

It is not the queries which determine which tags are equivalent, but the DTD designer and document collector. The replacement of tags by equivalents should be done before the documents are indexed, so that the index and the query agree about what elements are which. That is just what architectural form processing can do for you.

We may not want to index some elements, either because they do not contain text or because the text is never useful. (We yearned mightily for some way to get rid of ⟨ref⟩ elements during evaluation. They should never have been returned in the first place.)

Some elements may be presentation markup which it is useful to ignore (see Table 2 in [23]). This is especially useful because these are the tags which spoil the simple "add a space after each element" rule for modified string-value. For example, given ⟨st⟩V⟨scp⟩OICE⟨/scp⟩ XML⟨/st⟩ we would like this to be treated as ⟨st⟩VOICE XML⟨/st⟩. We want to ignore the tags of these elements, but not their contents.

In the spirit of architectural form processing, we can address these issues by adding attribute declarations in the DTD. XML allows us to add attribute declarations without changing the original ones, so xmlarticle.dtd could become

```
<!ENTITY old-dtd PUBLIC "..." "oldarticle.dtd">
%old-dtd;
<!ATTLIST ...>
...
<!ATTLIST ...>
```

with the original xmlarticle.dtd renamed to oldarticle.dtd. It is important that this can be done without touching the original DTD or the original XML files in any way.

The three attributes we want to add are

- INEXscan

nothing do not index this tag or its descendants

content do not index this tag; index its content

element index this tag; do not index its content

all index this tag and its content

The evaluation tool should heed this attribute; it would materially reduce the labour of judging.

- INEXname

if present, the name that is to be used in the index, and in queries, instead of the original element type.

- INEXatts

a list of pairs of names: *attr* - means "do not index @*attr*, *attr alt* means "index @*attr* under the name @*alt* instead". If an attribute is not in the list, it is indexed as itself.

For example, we might have

```
<!ATTLIST ip1 INEXname NMTOKEN #FIXED "p">
<!ATTLIST ip2 INEXname NMTOKEN #FIXED "p">
<!ATTLIST ip3 INEXname NMTOKEN #FIXED "p">
<!ATTLIST ip4 INEXname NMTOKEN #FIXED "p">
<!ATTLIST scp INEXscan NMTOKEN #FIXED "content">
<!ATTLIST ref INEXscan NMTOKEN #FIXED "nothing">
```

The mapping can be handled by a trivial post-parser.

7. THE SIMPLEST THING THAT COULD POSSIBLY WORK

The following query language was constructed to be just powerful enough to handle the queries people actually wrote. It clearly separates paths and text queries, allowing Boolean combinations of text queries but not of paths.

$$⟨topic⟩ ::= ⟨about⟩$$
$$| \quad ⟨filtered\text{-}path⟩ \ ⟨star⟩^? \ ⟨about⟩$$
$$| \quad ⟨filtered\text{-}path⟩ \ ⟨about⟩^?$$
$$⟨filtered\text{-}path⟩ \ ⟨star⟩^? \ ⟨about⟩$$

An ⟨about⟩ is basically a Boolean query plus context for the terms. A ⟨filtered-path⟩ describes a path in an XML document; the attributes of elements may be checked. There is no way of marking the "child" relation anywhere, or of specifying ordinal position.

If P and Q match ⟨filtered-path⟩ and A and B match ⟨about⟩, then A means "answer any elements that are about A"; PA means "answer any instances of P that are about A"; $PAQB$ means "for instances of P that are about A return instances of Q under that P which are about B"; and a missing A imposes no constraint.

$$⟨star⟩ ::= \text{'/'} \text{'*'}$$

A ⟨star⟩ may precede the final ⟨about⟩. This is to handle the queries which used //* in XPath. It means that once an instance of the preceding P or Q has been found, *any* descendant of that instance which fits the last ⟨about⟩ may

be reported. Such descendants are of course subject to ranking in the same way as any others, elements which are too "dilute" should not be a problem.

⟨filtered-path⟩ ::= ⟨filtered-elem⟩ ('/' ⟨filtered-elem⟩)*
⟨filtered-elem⟩ ::= XML-name ⟨filter⟩*

An XML-name is any XML identifier, possibly including colons. The time to deal with namespaces will be when we have to. The '/' operator means "descendant", not "child". This is what most people expected '/' to mean in the INEX'03 query language.

⟨filter⟩   ::= '[' ⟨attr-path⟩ ⟨range-list⟩ ']'
⟨range-list⟩   ::= ⟨range⟩ (',' ⟨range⟩)*
⟨range⟩   ::= number ('..' number?)?
                 | '..' number
⟨attr-path⟩   ::= ⟨attr⟩|⟨simple-path⟩
                 | ⟨simple-path⟩ ⟨attr⟩
⟨attr⟩   ::= '@' XML-name
⟨simple-path⟩ ::= XML-name ('/' XML-name)*

A filter compares text with a range of numbers. An ⟨attr-path⟩ is followed to find some text; the text may be the (modified) string value of an attribute or the (modified) string value of an element. Spaces and punctuation are trimmed from that modified string value; if the result can be converted to a number, the filter is satisfied to the degree that the number is in one of the ranges.

In a range $x..y$, x is the lower bound and y is the upper bound. It is an error if $x > y$. Missing x means $-\infty$; missing y means $+\infty$.

This query language does not use conventional notation like $<$ or $=$. There are two reasons for that. One is that these queries are supposed to be easy to express in XML, and XML makes it hard to use $<$. The second is that $<$ and $=$ are associated with precise meanings. But this is an information retrieval query language; a value which is not precisely in range may still be somewhat relevant. Since we don't intend the standard meaning of the mathematical signs, we shouldn't use them; it is important not to lie to the user.

⟨about⟩   ::= '(' ⟨or-query⟩ ')'
⟨or-query⟩   ::= ⟨and-query⟩ ('|' ⟨and-query⟩)*
⟨and-query⟩   ::= ⟨not-query⟩ ('&' ⟨not-query⟩)*
⟨not-query⟩   ::= ⟨text-query⟩ | '~' ⟨text-query⟩

An IR engine may interpret these Boolean operators the way it would normally interpret any Boolean operators. The conventional precedence of the Boolean operators is followed. They need not be "precise", and although it is tempting to define algebraic identities for this query language, it would be inappropriate. The ampersand is also awkward to express in XML; some other spelling such as ';' could be allowed.

⟨text-query⟩   ::= ⟨basic-query⟩
                    | ⟨basic-query⟩ ':' ⟨simple-path-list⟩
⟨basic-query⟩   ::= (⟨restriction⟩ ⟨term⟩)+ | ⟨about⟩
⟨term⟩   ::= word | '"' word+ '"' | ''' word+ '''
⟨restriction⟩   ::= empty | '+' | '-'
⟨simple-path-list⟩ ::= ⟨simple-path⟩(',' ⟨simple-path-list⟩)*

A text query may ask whether a basic query matches the current element, or whether it matches some descendant element. The commas in a simple path list mean "or" just as they do in CSS.

A word is an XML-name that doesn't include any dots, colons, or underscores, or is a pair of such names with an apostrophe in between, or is a number. A sequence of words between matching quotation marks is a phrase. The '+' and '-' restrictions have the same meaning as in the INEX'03 query language.

That's all there is to it. A parser for this language has been built using Lex and Yacc.

Several features that were considered but deliberately excluded:

- Filtering on anything other than a numeric range. In simple cases, this can be handled by the $PAQB$ pattern. Complex cases haven't arisen. When they do, it will be important to be clear about whether we want precise matches, so that XHTML documents making extensive use of the "class" attribute could be handled, or information retrieval matches, in which case we could simply have [⟨attr-path⟩ ⟨about⟩].

- Any kind of language sensitivity. This is what the CSS '|=' predicate is for, and its ':lang" predicate. When the INEX collection includes mixed-language documents, we could perhaps use [:lang word].

- Any kind of position checks. It is easy enough to add syntax for this, just copy XPath. What's hard is to interpret it. For example, as the XPath specification points out, "The location path //para[1] does *not mean the same as the location path /descendant::para[1]." Adapting* [:first-child] *from CSS would make more sense.*

- *Allowing any number of* ⟨path⟩⟨about⟩ *pairs. There's no difficulty in adding this, it just isn't needed.*

- *Allowing an axis other than "descendant". From a DTD, it is possible to compute a binary relation "can have child", the transitive closure of which is "can have proper descendant". This can be used to check the plausibility of queries. CSS also allows "child" and "sibling", which are similarly checkable. The complex mixing of axes in XPath makes it hard to check; we don't want to go there.*

8. SOME SAMPLE INEX'03 QUERIES

Query 61 //article[about(.,'clustering +distributed') and about(.//sec,'java')]

⇒ article(clustering +distributed & java:sec)

Query 64 //article[about(./, 'hollerith')] //sec[about(./, 'DEHOMAG')]

⇒ article(hollerith) sec(DEHOMAG)

Query 66 /article[./fm//yr < '2000']//sec[about(.,'"search engines"')]

⇒ article[fm/yr ..1999] sec("search engines")

Query 68 //article[about(., '+Smalltalk') or about(.,
'+Lisp') or about(.,'+Erlang') or about(., '+Java')]//
bdy//sec[about(., '+"garbage collection" +algorithm')]

⇒ article(+Smalltalk|+Lisp|+Erlang|+Java) bdy/sec(
+"garbage collection" +algorithm)

Query 71 //article[about(.,'formal methods verify correct-
ness aviation systems')]/bdy//*[about(.,'case study ap-
plication model checking theorem proving')]

⇒ article(formal methods verify correctness aviation
systems) bdy/*(case study application model checking
theorem proving)

Query 76 //article[(./fm//yr='2000' OR ./fm//yr='1999')
AND about(., '"intelligent transportation system"')]//
sec[about(., 'automation +vehicle')]

⇒ article[fm/yr 1999..2000]("intelligent transportation
system") sec(automation +vehicle)

Query 91 Internet traffic

⇒ (Internet traffic)

9. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) Version 2.0. W3C Working Draft, The World Wide Web Consortium, August 2003.

[2] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, and J. Robie. XQuery 1.0: An XML Query Language. W3C Working Draft, The World Wide Web Consortium, November 2003.

[3] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2; CSS2 Specification. W3C Recommendation, The World Wide Web Consortium, May 1998.

[4] R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, January 2000.

[5] D. Chamberlin, J. Robie, and D. Florescu. Quilt: an XML query language for heterogeneous data sources. In *The World Wide Web and Databases: Third International Workshop WebDB 2000, Dallas, TX, USA, May 2000. Selected Papers*, number 1997 in Lecture Notes in Computer Science. Springer-Verlag, January 2001.

[6] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, The World Wide Web Consortium, November 1999.

[7] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, The World Wide Web Consortium, November 1999.

[8] S. Cluet, S. Jacqmin, and J. Simeon. The New YATL: Design and Specifications. Technical report, INRIA, 1999.

[9] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. Technical report, AT&T Labs, 1998.

[10] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. A query Language for XML. W3C submission, The World Wide Web Consortium, August 1998.

[11] M. Fernándex, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, The World Wide Web Consortium, November 2003.

[12] N. Fuhr and K. Grosjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Research and Development in Information Retrieval*, pages 172–180, 2001.

[13] S. Geva. Re: Semantics of CAS Topics for the SCAS Task. E-mail to the INEX'03 participants, July 2003.

[14] C. F. Goldfarb. Hytime: A Standard for Structured Hypermedia Interchange. *IEEE Computer*, 24(8):81–84, August 1991.

[15] C. F. Goldfarb, S. R. Newcomb, W. E. Kimber, and P. J. Newcomb. *Information processing—Hypermedia/Time-based Structuring Language (HyTime).* Number 10744:1997 in ISO/IEC. International Organization for Standardization (ISO), second edition, 1997.

[16] T. R. M. D. S. Group. The Structured Information Manager. Web Site, 2003. viewed in November 2003.

[17] ISO. *Document Style Semantics and Specification Language (DSSSL).* Number 10179:1996 in ISO/IEC. International Organization for Standardization (ISO), 1996.

[18] M. Kay. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, The World Wide Web Consortium, November 2003.

[19] W. E. Kimber. HyTime and Sgml: Understanding the HyTime HyQ Query Language. internal report, IBM, August 1993. findable on the Web.

[20] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, The World Wide Web Consortium, November 2003.

[21] S. R. Newcomb, N. A. Kipp, and V. T. Newcomb. The HyTime Hypermedia/Time-Based Document Structuring Language. *Communications of the ACM*, November 1991.

[22] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). W3C submission, The World Wide Web Consortium, 1998.

[23] A. Trotman and R. A. O'Keefe. Identifying and Ranking Relevant Document Elements. In *INEX '03*, 2003.