Big, Fast XQuery: Enabling Content Applications

Mary Holstege
Mark Logic Corporation

Abstract

Increasingly, companies recognize that most of their important information does not exist in relational stores but in documents. For a long time, textual information has been relatively inaccessible and unusable. Database applications allow relational data to be used and re-used; the architecture of relational database systems allow such applications to function even in the face of large amounts of data. XML [10] and XQuery [8] now allow the creation of a new kind of application that unlocks content in a similar way: a content application. In this paper, we examine the technologies that enable content applications to operate at scale in the context of MarkLogic Server [2].

1 Content Applications

Database applications built on top of relational database management systems use SQL to select specific pieces of data, join them against other data, and reassemble them into new views. It is this flexible, granular reuse of data makes relational databases powerful tools. Relational databases, however, are less useful for dealing with content which is arranged not in regular typed fields but in complex hierarchical documents consisting of running text.

Documents are often described as "unstructured" or "semi-structured" but the problem with documents, from a relational point of view, is not that there is too little structure, but that there is too *much*. Consider a medical document that describes the course of treatment for a patient, with procedures, observations, and actions indicated. Part of such a document, using XML markup, is shown in Figure 1.

The document has sections, which have titles and content. The content has running text which is interspersed with markup for things such as instruments, actions, and observations. Some of these are nested in other markup. The relation of an instrument, say, to the section's content is not simple: the order relative to other entities and other chunks of text is crucial and defines the narrative. While a relational model for this information is certainly possible, it is difficult and loses the narrative coherence of the original. It is easy to represent this narrative structure using XML markup, however. The more semantically rich and detailed the markup becomes, the harder it gets to map into a relational model, and, crucially, the harder it becomes to further enrich the mapped structure.

A content application is to textual content as a database application is to relational data. It uses a query language to select specific pieces of documents, join them against other document pieces, and reassemble them into new documents. A content application goes beyond simple text search ("get me the document that contains the phrase 'important symptom"') into fine-grained selection and assembly based both on full-text operators and

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
...
  <section>
    <section.title>Procedure</section.title>
     <section.content>
      The patient was taken to the operating room where she was placed
      in supine position and
      <anesthesia>induced under general anesthesia.</anesthesia>
      <prep>
        <action>A Foley catheter was placed to decompress the
        bladder</action> and the abdomen was then prepped and draped
        in sterile fashion.
      </prep>
      ...
      The fascia was identified and
      <action>#2 0 Maxon stay sutures were placed on each side of
       the midline.
      </action>
      <incision>
        The fascia was divided using
        <instrument>electrocautery</instrument>
        and the peritoneum was entered.
      </incision>
      <observation>The small bowel was identified.</observation>
'''
```

Figure 1: Simple Medical Document

operators over the hierarchical narrative of the document. A query such as "show me all procedures involving important symptoms where no anesthesia occurs before the first incision" entails full-text ("important symptoms, "procedure"), ordering (phrase "important and "symptoms", "occurs before"), and hierarchy (section's content's incisions). A relational model suited to answering such a question would be ill-suited to reconstructing the original narrative (the entire procedure): the reconstruction would be an immensely complex join.

With increasing amounts of content being created natively in XML or being readily convertible to XML the time is ripe for complex scalable content applications built on XML. What that calls for is a language designed for effective XML processing and a system architecture optimized for content.

1.1 Characteristics of Content Applications

Content applications vary widely, but some general trends can be identified:

- Individual documents may to be relatively large. Documents as large as 10 megabytes are common; those running to gigabytes are not unknown.
- The number of documents may also be large, and can run into tens of millions of documents or more. However, the number of documents is usually much smaller than this.
- Content bases are frequently created from large amounts of existing content which needs to be loaded and indexed in bulk.
- In general, update is less frequent than selection, but for Web 2.0 style content applications, being able to add annotations and metadata to the content base is also important. Frequently the content-loading and content-cleaning (update-intensive) and content-access (update-light) phases in the life-cycle of a content application are distinct, so optimization does not need to focus on maintaining fast query during periods of heavy update. This differs from on-line transactional database applications.
- It is important to be able to select small pieces of documents based on full-text, ordering, and hierarchical criteria. Full-text searching brings in linguistic knowledge for stemming, tokenization, and thesauri, as

well as relevance calculations to determine which matches are better than others. Multi-lingual content bases, documents, or even paragraphs are not uncommon.

- Document schemas may be complex, fluid, or unknown. As content applications evolve, new markup may be incrementally added to enrich the content and enable new kinds of queries.
- Content frequently arrives that does not adhere to the defined document structure, in need of clean-up. Clean-up transformations may be very complex, involving hierarchical restructuring.
- Content applications do evolve: to gain business advantages over competitors, to provide more accurate selections or better context for information, to integrate new content sources, and so on. Evolution of a content application often leads to an evolution of the content and vice versa: this in turn implies that flexible schemas and some ability to update content in place are important.
- Content applications are amenable to end-to-end XML processing: content can readily be encoded and stored in XML, XML (especially XHTML) can be directly rendered in a browser, and given an XML-oriented processing language, selected pieces of XML can be operated on to provide appropriate business logic.

Given these general characteristics of content applications, Section 2 reviews some aspects of XQuery that suit it to building content applications, and and Section 3 looks at how the architecture of the MarkLogic Server supports building scalable content applications.

2 Query Language

To enable content applications, the query language needs to understand large, hierarchically structured documents containing human text. It needs to permit highly granular selection based on both the hierarchy and the ordering of children (both of which are highly salient features of content). The XQuery family of specifications allows documents marked up with XML to do just this.

- XML-aware

  XQuery[8] was defined specifically as a query language for XML content. Simple path expressions can be used to select based on XML structure (both hierarchical and sequential) and XML result structures can be easily generated, often using an XML syntax.

- Full-text

  The query language for content applications needs to be able to perform full-text operations, not just string matching, to take into account language-sensitive operations such as stemming and tokenization. The proposed Full-Text extensions to XQuery[9] will provide such functionality.

- Update

  Although content applications typically focus on selection more than updating, the ability to update content or to add annotations or metadata is important to many content applications. Update operations also provide for performing content cleanup and augmentation in place. The proposed Update extensions to XQuery[7] will define update operations.

- Extension Functions

  Depending on the application, specialized capabilities may be required, such as security-related operations, or trigonometric functions. Fortunately, XQuery provides an extension mechanism through function libraries. XQuery itself defines a large selection of built-in functions and operators[6] for basic data manipulation.

- Optimization-friendly

  XQuery is a functional language without side-effects in which most functions are deterministic and stable within a single query, allowing optimizers to freely reorder expressions and avoid recomputing expressions. [1] The proposed update facility uses snapshot semantics which preserves this aspect of the language: the exact order in which updating operations are performed makes no difference to the result of the expression, so an optimizer is free to reorder expressions.

  The XQuery language design also enables lazy evaluation: if a path expression results in a million node sequence, only those nodes in the sequence that make a material difference to the final result need be fetched. Since full-text searches on large content bases can frequently produce such large results, the ability to notice that only the first ten results are being rendered and returned as a result of the overall query can lead to tremendous savings in effort.

- As Typed As You Want To Be

  Another interesting feature of XQuery that is particularly useful for content applications is that, while operations *may* be strongly typed, they *need* not be. An XQuery program can require a variable, function parameter, or return value to be of a specific named type that is declared in an XML Schema, or it can allow it to be anything at all. More flexibility is possible: a "type" constraint could be the requirement that the item be some kind of XML node, or a specific kind of XML node — an element for example — or an element with a particular name, whether defined in an XML Schema or not.

  Content frequently does not arrive perfectly conformant to some schema, and it may evolve over time. It is a great benefit to content applications to be able to use the same tools tools to perform the initial clean-up and evolution of content as are used to process normalized content. XQuery programs with different degrees of typing can be applied at different stages of the process. Alternatively, loosely typed XQuery programs can be used to process content without having to normalize it at all.

3 System Architecture

Query language functionality is only part of the puzzle for enabling content applications. Efficient execution of the query language at scale is important for real-world content applications. The architecture of the MarkLogic Server[2] enables such efficient execution at scale, by optimizing for the characteristics of content applications and taking advantage of the opportunities afforded by the features of XQuery. As [4] and [5] point out, special-purpose databases tuned for particular kinds of problems can easily out-perform general purpose relational databases in their problem domain by factors of 10 or more.

The MarkLogic Server architecture divides processing into two fundamental parts: evaluation and data access. Typically, scaling is accomplished by distributing the evaluation to one set of hosts called E-nodes ("evaluation nodes") and data access to another set of hosts called D-nodes ("data nodes"). The E-node and D-node functionality can be also be combined into a single host. Load balancers and caching proxies can be used to reduce and distribute the load across E-nodes.

3.1 D-nodes

D-nodes store the XML documents along with indexes to enable efficient access to those documents. D-nodes respond to requests from E-nodes to locate, fetch, or update documents under their control.

[1] There are some exceptions, such as the fn:trace() and fn:error() functions, as well as vendor extension functions to perform HTTP requests, compute random numbers, or report execution times, and the like. Optimizers are nevertheless left with a fairly free hand.

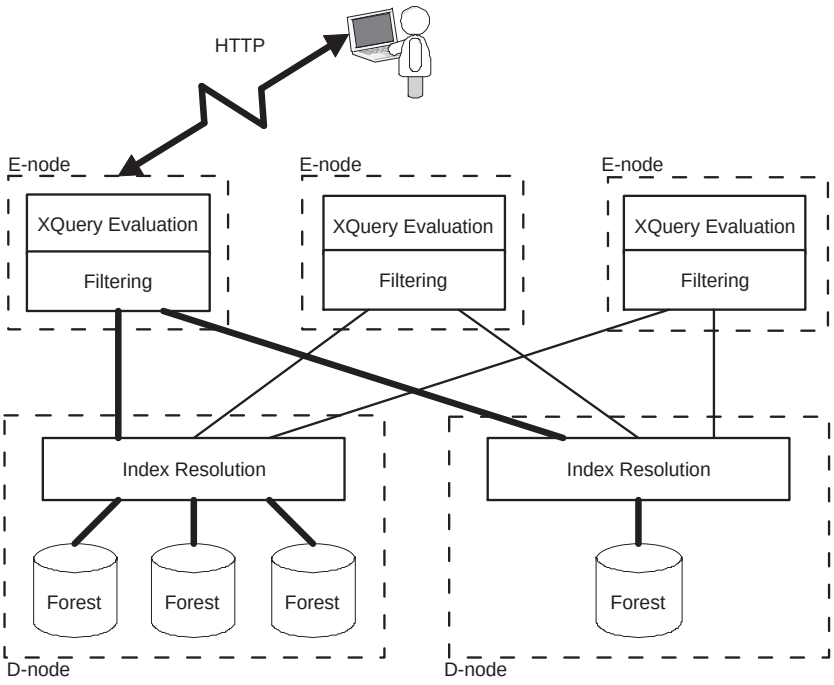

Figure 2: System Architecture

- Fragments

  Documents are broken into non-overlapping units of access called fragments. Fragments are the basic unit of operation in the system. Updates and selection from the databases occurs at the fragment level. Fragmentation choices involve making trade-offs between the expected number of fragments that may need to be fetched and processed to return the correct results of a query, and the size of fragments that must be filtered to produce the correct results or written to disk to process an update. These tradeoffs can be complex and how to balance them is beyond the scope of this paper.

- Forests

  A database may be broken into multiple forests, where each forest is under the control of a specific D-node. Distribution of data across multiple forests allows for greater concurrency and scaling.

- Inverted Indexes

  The forests managed by each D-node include universal indexes that map facts about documents to posting lists. Many kinds of posting lists exist: posting lists for each word, but also posting lists for structural facts, such as the presence of particular elements. The indexes are compressed inverted indexes[11]. Index settings control which specific kinds of posting lists are available in the indexes, and whether the lists record position details or just fragment identifiers.

  XQuery path expressions and full-text queries can be resolved against the indexes by intersections and unions of posting lists for the component facts. Such a result may not be accurate, however. For example, if only a simple word index without positions were available, the phrase query “simple example” could not be accurately resolved in the index. The best the index could do is return fragments that have postings for both “simple” and “example”. A secondary phase, called the filter, is responsible for weeding out the false matches. Index resolution can provide accurate answers in many cases: there is a tradeoff between

index resolution accuracy and index size. Again, details of how to balance such trade-offs is beyond the scope of this paper.

- Managing Updates

  The fragments themselves are stored in memory-mapped compressed representations of the document trees. This tree data and the indexes are stored in "stands" in accordance with the principals of a log-structured file system[3]: new content is initially held in memory and then written out in a sequential fashion when a sufficient amount has accumulated. Journalling allows for recovery in the event of a crash. Once written, the tree data and indexes are never updated. Fragments are updated by writing new versions of those fragments to new stands and noting that the fragment in the old stand has been deleted. The server uses multi-version concurrency control[1] to increase the throughput for read operations. Each fragment has a timestamp associated with it. Read operations obtain the most recent version of the fragment with a timestamp preceding the current transaction's timestamp, and therefore always obtain a consistent snapshot of the content base. A periodic merge process creates new stands with any deleted fragments eliminated and the indexes merged to optimize access. This allows both updates and selection to be fast under the normal expected conditions of relatively few active stands and a relatively modest update load once the bulk of the content has been added. Initial loading can also be fast because sequential writes of data in bulk is faster than piecemeal random writes.

3.2 E-nodes

E-nodes are responsible for communicating with clients and for XQuery evaluation: parsing, static analysis, dynamic evaluation, and assembly and serialization of results. E-nodes include HTTP listeners that service requests to execute XQuery modules and return the results.

- Query Processor

  The query processor performs static analysis and rewrite optimization of the query. Any operations that access data are converted into index requests and sent to the D-nodes, with the results being filtered as necessary. The query processor relies on lazy evaluation of node sequences to avoid fetching or processing content unless it is required by the ultimate result of the query.

- Filter

  The filter iterates through the postings returned by the D-nodes and applies the specific match criteria to the selected fragments and returns the requested nodes.

- Application Server

  An E-node also operates as an application server. It accepts HTTP requests on configured ports. In addition to performing conventional serving of documents, requests for XQuery modules are serviced by executing the indicated module and returning the results. Direct execution of XQuery modules enables a rapid development methodology for web applications: XHTML can be generated directly from XQuery for consumption in a browser.

3.3 Summary: Basic Query Flow

Consider a simple query for a phrase within an element as part of a larger query that only makes use of the first ten hits.

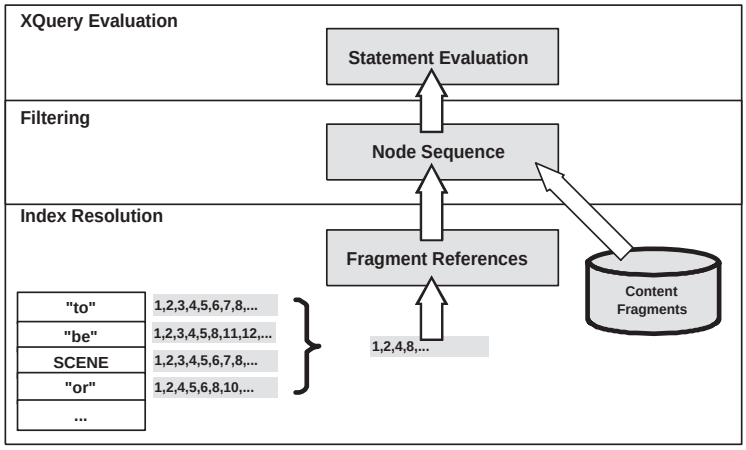

Figure 3: Query Evaluation

1. A client sends an HTTP request to an E-node. The application server accepts the request, locates, parses, and analyzes the appropriate XQuery module. For example: [2]

```
import module namespace my="http://marklogic.com/example"
   at "/MarkLogic/example.xqy";
for $result in cts:search( //SCENE, "to be or not to be" )[
    fn:position() = (1 to 10)
  ] return my:render-result($result)
```

2. The XQuery evaluator constructs an index query to be resolved by the indexes, based on the knowledge of available indexes. In this case, the query parser produces an index request such as:

```
AND(SCENE,"to","be","or","not")
```

3. The indexes combine posting lists to form a sequence of fragment references. Depending on the indexing options, index resolution may return "false positives", fragments identified by the index that do not match original criteria. Each D-node operates in parallel. Index resolution in this case examines the posting lists for the five terms, combines them into a single posting list that has references for all fragments that contain all five terms (fragments 1,2,4,8, etc. in the diagram).

4. The filter turns the sequence of fragment references into a sequence of nodes matching the original criteria. The first fetches each candidate fragment in turn and selects nodes in the fragment that actually meet the criteria (all the words in the phrase appearing in the appropriate order within a `SCENE` element). A fragment containing, for example, the phrase "not to be seen or heard" would be returned from the index resolution, but would not meet the original criteria and would be skipped by the filter.

5. XQuery is evaluated to render the result nodes. Lazy evaluation of the node sequence causes fragments to be fetched and filtered only as needed. In this case the filter only fetches as many candidate fragments are required to return ten `SCENE` element nodes to pass to the `my:render-result` function. If the `ACT` element were the root of the fragment, the entire act would be fetched for filtering, but only the matching

[2]The XQuery Full-Text extension defines the operator `ftcontains` which can be used to test whether a particular node matches some full-text criteria. A common case is to return the sequence of matching nodes, generally ordered by decreasing score. This is what the `cts:search` extension function does.

SCENE elements would be returned. If the ACT has ten matching SCENE elements, only that one fragment would be fetched.

6. The application server constructs an appropriate HTTP response and returns it to the client.

Caches at various level short-circuit some of these operations.

4 Conclusions

The divide between "content" and "data" is not a hard and fast one. However, content applications do tend to have different characteristics than relational database applications. Representing content with XML, operating on it with XQuery, and executing on an architecture optimized for such operations can open up the possibility manipulating large content bases at a fine-grained level to create new and interesting applications. It provides for a middle path between simply identifying documents that meet certain full-text criteria on the one hand, and losing the overall complex hierarchical and narrative flow of documents on the other.

References

[1] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[2] MarkLogic Server 4.0, 2008. http://marklogic.com/product/marklogic-server.html.

[3] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[4] Michael Stonebraker and Uğur Çetintemel. 'One Size Fits All': An Idea Whose Time Has Come and Gone. In *International Conference on Data Engineering (IDCE)*, 2005.

[5] Michael Stonebraker *et al.* One Size Fits All? — Part 2: Benchmarking Results. In *Conference on Innovative Data Systems (CIDR)*, 2007.

[6] Ashok Malhotra *et al.* (editors). XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath-functions-20070123/.

[7] Don Chamberlin *et al.* (editors). XQuery Update Facility 1.0. Candidate Recommendation, W3C, March 2008. http://www.w3.org/TR/2008/CR-xquery-update-10-20080314/.

[8] Jérôme Siméon *et al.* (editors). XQuery 1.0: An XML Query Language. W3C Recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[9] Sihem Amer-Yahia *et al.* (editors). XQuery and XPath Full Text 1.0. Candidate Recommendation, W3C, May 2008. http://www.w3.org/TR/2008/CR-xpath-full-text-10-20080516/.

[10] Tim Bray *et al.* (editors). XML 1.0 Recommendation. Fourth Edition, W3C, August 2006. http://www.w3.org/TR/1998/REC-xml-20060816.

[11] Justin Zobel and Alistair Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38, July 2006.