Quark: An Efficient XQuery Full-Text Implementation

[Demonstration Paper]

Anand Bhaskar, Chavdar Botev, Muthiah M Muthaia Chettiar, Lin Guo,
Jayavel Shanmugasundaram, Feng Shao, Fan Yang
Cornell University
Ithaca, New York
{ab394, cbotev, mm376, guolin, jai, fshao, yangf}@cs.cornell.edu

1. INTRODUCTION

XQuery 1.0 and XPath 2.0 Full-text (XQFT) has been recently developed and proposed for extending XQuery and XPath with full-text search. While there have been many previous proposals to integrate XML queries and full-text search, an interesting aspect of XQFT is that it integrates full-text search with a full-function XML query language (XQuery). Consequently, it presents new challenges and opportunities for integrating sophisticated queries with full-text search. In this demonstration, we focus on two such challenges: (a) performing keyword search over XML views, and (b) scoring arbitrarily complex combinations of XQuery and XQFT queries. We illustrate the power of these two features using an example movie database that stores XML information about movies and their reviews (Figure 1).

Keyword Search Over Views. Given a movie database, users may wish to create personalized views over the data. For instance, some users may only be interested in 'thriller' movies, while others may only be interested in movies made after the year 2000. An even more interesting example is a user who wishes to see the reviews of each movie *nested* under the movie element, instead of being listed separately. XQuery is well suited to creating such personalized views of XML data. For instance, we can create a view in XQuery for nesting reviews under their corresponding movies using the following Query 1:

```
declare function AllMoviesData() {
  for $facts in doc("facts/*.xml")/facts
  for $reviews in doc("reviews/*.xml")/reviews
  where $facts/@movieid = $reviews/@movieid
  return <movie movieid="{$facts/@movieid}">
           {$facts/*} {$reviews/*}
         </movie>                }
```

Given the above personalized view, a user may wish to search this view for "excellent greek" movies. This can be expressed in XQFT using the following Query 2:

```
AllMoviesData()/movie[. ftcontains
                        {"Greek", "excellent"}]
```


SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

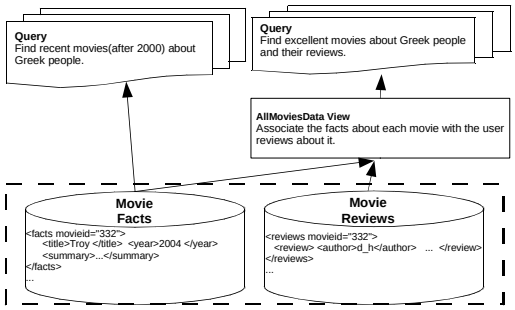

Figure 1: Movies Database Application

However, this query raises an interesting challenge: how do we efficiently evaluate keyword search queries over views? In traditional information retrieval (IR), keyword search is performed over static documents that are pre-indexed using inverted lists. But how can we efficiently evaluate keyword search queries over dynamically generated content; over virtual views that have not been materialized and indexed?

Scoring Arbitrarily Complex XQuery/XQFT Queries. A common user task is querying for movies of interest, and browsing the ranked results. A typical query can include a mix of structured and full-text predicates. For example, a user might query for "excellent Greek" movies that were created after the year 2000, in rank order. The above query can be expressed in XQFT as the following Query 3.

```
for $m score $s in
    doc("facts/*.xml")/facts[./year > 2000 and
           ./summary ftcontains
   {"Greek","excellent"}]
order by $s desc
return $m
```

In the XQFT query, the "score" clause binds a score to each movie that satisfies the user predicates. The query then orders the results and returns them in scored order. The interesting aspect of the above query is that it involves both structured ($year > 2000$) and full-text predicates ('Greek', 'excellent'). Consequently, it makes sense to score the results based on a combination of structured and full-text predicates. Specifically, a movie should be ranked higher if it is more recent *and* if it is also more relevant to the full-text search predicate. This query again raises an interesting question: how can we score arbitrarily complex XQuery and XQFT queries, which may contain any combination of structured and full-text predicates?

In this demonstration, we will present the Quark open-

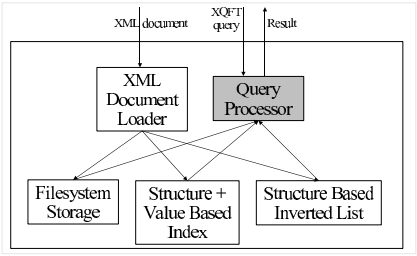

Figure 2: Quark System Architecture

source data management system (http://www.cs.cornell.edu /database/quark) that is architected to effectively address the above questions. In particular, we will demonstrate the following features of Quark:

- An efficient technique for evaluating keyword search queries over XML views by exploiting inverted list structures on the *base* XML data.
- A general and efficient framework for scoring and ranking arbitrary XQuery and XQFT expressions. To the best of our knowledge this is the first scoring implementation for the complete XQuery language, including XQFT.

For the demonstration, we have created and indexed a data set with movies and reviews extracted from Internet movie sites. We also have other XML data sets including INEX (*http://inex.is.informatik.uni-duisburg.de:2004/*), DBLP in XML (*http://dblp.uni-trier.de/xml/*), and SIGMOD Record in XML (*http://www.dia.uniroma3.it/Araneus/Sigmod/*). The total size of all collections is over 900MB, and Quark provides interactive response time for most user queries.

2. QUARK SYSTEM ARCHITECTURE

Figure 2 shows the architecture of Quark. When an XML document is loaded into the system, it is processed by the Document Loader. The Document Loader first assigns unique Dewey IDs [8] to the nodes of the document (a Dewey ID is a hierarchical numbering scheme where the ID of the node contains the ID of its parent node as a prefix). The document is then stored/indexed in three different formats[1], as described below.

The document is stored in a compressed binary format in the Filesystem Storage – this format uses techniques similar to the XML binary representation techniques described in [1], and is used as the primary storage format for the document. The Structure+Value Based Index (SVBI) is similar to RootPaths [3] and can efficiently support branching XPath queries, including those with structured predicates (such as $year > 2000$). Finally, the Structure Based Inverted List Index (SBILI) is similar to the XML inverted list organization proposed in [2]. For each keyword occurring in the stored documents, the inverted list stores the Dewey IDs of the elements that *directly* contain the keyword. Each keyword inverted list also has a B+-tree index built on the Dewey ID to allow for fast access to different parts of the list.

[1]One of the design decisions of Quark is to deal with "read and append-mostly" applications, such as document repositories and content management systems. Consequently, storing/indexing each document in three different formats is not expected to adversely affect performance.

3. KEYWORD SEARCH OVER VIEWS

As mentioned in the introduction, the ability to create views over data is one of the powerful features of XQuery (and other database-style query languages). Views have many advantages, including logical data independence, access control and the presentation of tailored views of data to users as illustrated in the movie example. Given such views, users may then wish to evaluate full-text search queries over the views. As in our example Query 2, a user may wish to issue a keyword search query over the view containing movies and its reviews. An interesting question thus arises: what is the most efficient way to support full-text queries over views?

At a high level, there are three possible solutions to this problem. The first solution is to compute the view at the time of loading XML documents, materialize and index the view as a separate XML document, and then use regular query processing techniques to evaluate queries (structured or full-text) over this view. However, the main disadvantage of this method is that the view may not be known until query time, and hence cannot be materialized and indexed before query execution time.

A second solution is to materialize the view *during query processing*, and then process full-text queries over the view. The main disadvantage of this approach is that there are no indexes built over the view (since it is not known until query time), and hence evaluating queries over the view can be inefficient.

To address the above issues, we propose a third solution where the entire view is not materialized during document loading or query processing. Rather, the indices on the base data (facts and reviews, in our example) are used at query time to identify the view elements that satisfy the full-text query, and only the relevant view elements are materialized. An interesting aspect of our solution is that it can produce the same score for the relevant view elements as the other two techniques, using a variant of TF-IDF scoring.

The above solution is similar to the problem of answering database queries over views by rewriting the queries over the base data (e.g., see [4, 7]). The main difference between these approaches and our proposed approach is that we consider keyword search queries – the different semantics of keyword queries and the presence of scoring gives rise to new challenges.

We now describe the main idea behind our technique for processing keyword search over views. Figure 3 presents the original query evaluation plan for Query 2. Intuitively, the bottom part of the plan evaluates the *AllMoviesData* view, and the top part evaluates the keyword search query over the view. We propose to rewrite the above query plan by pushing down the keyword search query over the various XQuery operators until it is evaluated directly over the base data. Since the base data is indexed using Structured Inverted Lists, the query can then be evaluated efficiently.

The result of rewriting the query plan in Figure 3 is shown in Figure 4. Essentially, the keyword search for "Greek" and "excellent" over items in the *AllMoviesData* view has been rewritten into eight sub-queries. Each sub-query tries to match the keywords in one or two of the underlying data sets, the facts data set and the reviews data set, and then assembles these intermediate results to determine the result of the keyword search query. For instance, the sub-query in Figure 4 matches "excellent" in the facts and "Greek" in the reviews. The rewritten plans have significant benefits over the original plan because the keyword search query is eval-

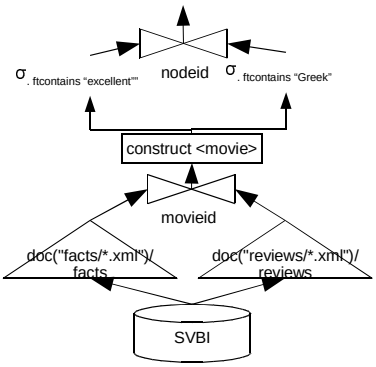

Figure 3: Original Plan

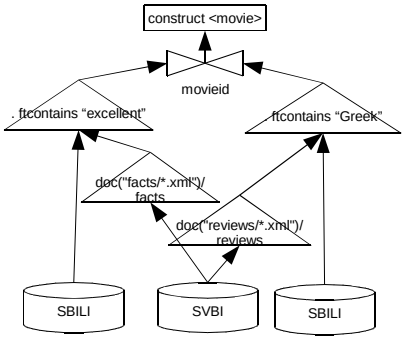

Figure 4: Rewritten Plan

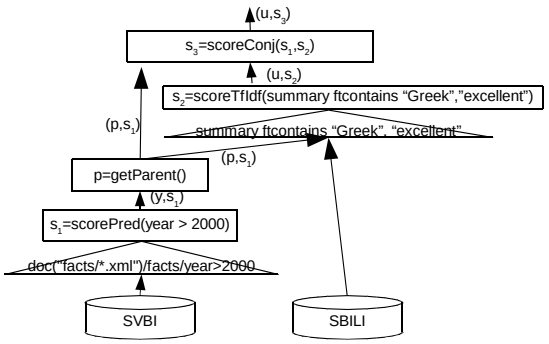

Figure 5: Score Computation and Propagation

uated over the base data, which is indexed using Structured Inverted Lists. It should be noted that the lookups in the Structured Inverted Lists are based on the context-sensitive methods described in [2]. Therefore, the cardinality of the inputs at the movieid join operator on Figures 4 will never be worse than the inputs of the movieid join operator on Figure 3.

4. SCORING ARBITRARILY COMPLEX XQUERY / XQFT QUERIES

The second challenge outlined in the introduction stems from the fact that users can score arbitrarily complex XQuery and XQFT queries. While there has been some previous work on scoring a mix of structured and full-text queries [6, 5, 9], these techniques focus on the relational model or restricted XML query languages. In contrast, we focus on scoring the *entire* XQuery language, including XQFT.

We support such a general scoring mechanism as follows. First, we extend the basic XQuery evaluation model with scores. Traditionally, each XQuery expression produces a sequence of items. To capture scores, each XQuery expression now produces a sequence of *scored* items. Second, we extend each XQuery expression to propagate scores. The score propagation is based on a probabilistic interpretation of scores [6], and is extended to work with any XQuery expression. Third, we extend each XQuery function to propagate scores, again preserving the probabilistic interpretation. Finally, we integrate TF-IDF scores for full-text search with the scores of other XQuery expressions.

$Expr : ScoredSequence^k \rightarrow ScoredSequence$
$ScoredSequence := (Item, Score)*$

Figure 5 demonstrates how the scores are computed and propagated for Query 3. At the bottom of the plan, we have the access to the index structures, SVBI for the structured predicate and SBILI for the full-text predicate. The access methods for these indexes generate resulting (item, score) pairs using index-dependent scoring methods. Then, we aggregate the scores from the index lookups and generate the score for the conjunction s_3, which in our implementation is the product of s_1 and s_2. Further, the score is associated with the result of the conjunction and is further propagated up the expression tree.

5. QUARK DEMONSTRATION GUI

For the purpose of the demonstration, we have created a Java-based GUI, which provides a user-friendly interface to

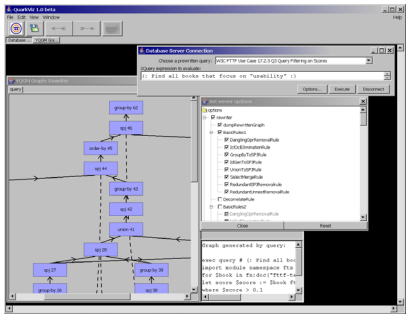

Figure 6: The Quark GUI

the Quark server. The GUI allows the user to enter queries, view their results, and control the internal aspects of the server such as the rewriting and optimization techniques used. The GUI also provides a visualization of the query graph at different stages of query processing - such as after the parsing, rewriting or optimization phases (Figure 6).

We will demonstrate queries similar to those described in this paper. We will also demonstrate the implementation of the XQuery 1.0 and XPath 2.0 Full-text Use Cases (*http://www.w3.org/TR/xmlquery-full-text-use-cases/*).

6. REFERENCES

[1] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An evaluation of binary xml encoding optimizations for fast stream based xml processing. In *WWW'04*.

[2] C. Botev and J. Shanmugasundaram. Context-sensitive keyword search and ranking for xml. In *WebDB'2005 Poster*.

[3] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava. Index structures for matching xml twigs using relational query processors. In *XSDM'2005*.

[4] M. Fernandez, W.-C. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *WWW'1999*.

[5] N. Fuhr and K. Groβjohann. Xirql: a query language for information retrieval in xml documents. In *SIGIR'2001*.

[6] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[7] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying xml views of relational data. In *VLDB'2001*.

[8] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD'2002*.

[9] A. Theobald and G. Weikum. The index-based xxl search engine for querying xml data with relevance ranking. In *EDBT'2002*.