Answering XQuery Using Cached XPath Views

Bhushan Mandhani

June 2, 2004

1 Introduction

Query answering using views and semantic caching are well-studied problems for relational databases. In this paper, we explore these ideas in the context of answering XQuery queries posed to XML databases. Our focus is on client-side caching of data. We will be looking to store XML fragments corresponding to XPath expressions locally, and then reuse these fragments to answer queries posed by the client. Our motivation for doing this is two-fold. Firstly, by making queries run on small XML fragments instead of the original huge XML document, we hope to achieve a speedup in query execution. Secondly, consider a distributed setting in which a client is repeatedly posing queries to a remote XML data source. Here, when a client query can be answered from the local cache, we don't need to ship a query to the remote source, thereby achieving a speedup in query execution as well as savings in the amount of data the client has to retrieve over the network. Our experiments demonstrate the success of our approach in achieving these goals.

This paper is organised as follows. In Section 2, we describe the underlying theory which helps us determine when an XPath view can be used to answer an XPath query. In Section 3, we describe our semantic caching based XQuery answering system. Finally, in Section 4, we show our experimental results.

2 XPath Queries and Views

We first look at what it means to be able to answer XPath expression q using p. It means, we can find an expression r, which when applied to the data of p gives us q. We write this as $p \circ r = q$. Note that this is an inherent property of the expressions themselves, and holds for all data trees.

XPath expressions are naturally represented as pattern trees. We denote by P the pattern tree for p, and so on. Note that there will be a unique result node in P which corresponds to the elements being returned by p. We call the downward path from the root to the result node the axial path of P, and all the nodes on this path, the axial nodes of P. Thus if $p = a/b[@c]/d[e]$, then d is the result node, and a, b, d are the axial nodes.

For $p \circ r$ to be defined, the root node of r must match the result node of p. Further, the pattern tree of $p \circ r$ is obtained simply by stitching together P and R by merging the result node of P with the root node of R, so that R becomes a newly added subtree. The axial path of this new pattern tree is simply the axial path of P concatenated with the axial path of R.

Now, suppose $p \circ r = q$. It means the pattern tree Q is equivalent to the pattern tree P $\circ$ R. Thus, P $\circ$ R $\subseteq$ Q, and vice versa. The condition we use to check containment in our work is homomorphism between pattern trees. Though this condition is sufficient, but not necessary for containment, we assume containment does not hold if we can't find a homomorphism i.e., we "pretend" the condition is necessary too. Thus, equivalence of expressions a and b is checked by looking for homomorphisms both ways. Note that checking containment and equivalence of XPath expressions is known to be coNP-complete.

We now take a brief detour. Define the number of an axial node as the number of nodes in the path from the root to that node. We then have the following result.

Lemma: If expressions a and b are equivalent according to the condition described above, then in both homomorphisms, each axial node maps to an axial node in the other tree, and moreover, the two axial nodes have the same number.

Proof: Consider the homomorphism from A to B. The root must go to the root, and the result node must go to the result node. Since child edges in A go to child edges in B, and descendant edges in A go to downward paths in B, it follows that the axial path in A goes to a path from root to result node in B. But B is a tree, and has a unique path from root to result node, which is exactly its axial path. Thus, the axial path of A goes to the axial path of B. Let l(A) denote the number of nodes in the axial path of A. Then it immediately follows that l(A) $\leq l$(B). Similarly, considering the homomorphism from B to A, we get l(B) $\leq l$(A). Thus, l(A) = l(B), the two axial paths have the same number of nodes. Further, since each axial node must map to a distinct axial node, the Lemma follows.

The above Lemma is very useful for us. It is incorporated in our implementation of the dynamic programming algorithm for finding homomorphisms. It significantly reduces the space of possibilities the algorithm needs to consider, and thus speeds it up.

We now have the machinery in place to address the problem of determining, given arbitrary expressionss p and q, if p can be used to answer q. Let nodes $x_1, x_2, \ldots, x_n$ form the axial path in Q, so that x_1 and x_n are respectively the root and the result node. Let k be the number of nodes in the axial path of P. Let S denote the set of predicates for node x_k in Q. For any $T \subseteq S$, let Q_T denote the pattern tree obtained from Q by deleting all the descendants of x_k in Q except those which correspond to predicates in T. We then have the following result.

Theorem: P can be used to answer Q iff

$$\mathrm{Q}_S \subseteq \mathrm{P} \subseteq \mathrm{Q}_\phi$$

Further, if the above holds, the expression R such that P $\circ$ R = Q is simply the subtree of Q rooted at x_k.

Proof: Suppose P can be used to answer Q, using R, so that P $\circ$ R = Q. By our modified definition of equivalence, homomorphisms must exist both ways. Consider the homomorphism from Q to P $\circ$ R. By our earlier Lemma, $x_1, x_2, \ldots, x_k$ map to the axial path of P. Thus, a subset of this homomorphism is a mapping from Q_ϕ to P, and thus P $\subseteq \mathrm{Q}_\phi$. Similarly, a subset of the homomorphism from P $\circ$ R to Q gives us a mapping from P to Q_S, and thus $\mathrm{Q}_S \subseteq$ P.
We now prove the converse. Suppose $\mathrm{Q}_S \subseteq \mathrm{P} \subseteq \mathrm{Q}_\phi$. Let R be the subtree of Q rooted at x_k. Then P $\circ$ R $\subseteq \mathrm{Q}_\phi \circ$ R. But the right hand side is simply Q. Thus,P $\circ$ R $\subseteq$ Q. Similarly, $\mathrm{Q}_S \circ$ R $\subseteq$ P $\circ$ R. But the left hand side is simply Q. Thus, Q $\subseteq$ P $\circ$ R. Thus, Q = P $\circ$ R, and P can be used to answer Q. This completes the proof.

The above Theorem establishes the algorithm we use to determine when we can use P to answer Q, and if that is the case, what expression R to apply to P to get Q. Now, we are ready to build or caching based XQuery answering system.

3 Caching Based XQuery System

We now describe the architecture of our system. Each cache entry represents one cached XPath expression. It stores the URL of the XML document, the cached expression, the name of the local file on disk storing the corresponding XML data, the size of this file, and a timestamp. The cache is organised as a linked list of these entries in main memory, sorted by decreasing timestamps (so that the most recently used entries are at the head of the list). The maximum allowed cache size bounds the total size of all the cache files on disk.

The system scans a given query for the core XPath expressions in it i.e., those expressions which begin with "document" and which are responsible for determining how the XML data is accessed. Currently, the system doesn't support queries with variables inside predicates in these core expressions, since these are rewritten statically before executing the query.

For each core expression scanned, the system traverses the list of cached entries to determine if one of them can be used to answer the expression. If that is the case, and we have a cache hit, this expression will be replaced by a substitute expression which does the same job and which reads from a cache file on disk. The timestamp of the corresponding entry is updated, and it is moved to the head of the list, since we maintain the list in decreasing order of timestamps.

If no cache entry can be used, and we have a cache miss, we get the XML data for this expression from the data source, and store it in a local file. We then create a new cache entry for this expression, and insert it at the head of our list. Before we do that however, we first check whether enough space is available in the cache for this new entry. If that is not the case, we delete list entries, and their corresponding disk files, using LRU replacement, until enough space becomes available. Since our list is in decreasing order of timestamps, LRU is easily implemented by always deleting the last entry in the list. Once we have added the new entry, we appropriately replace the corresponding XPath expression in the query. Thus, the user query is rewritten and executed. This rewritten query uses only cache files.

4 Experiments

In our experiments, we used a 50 MB document generated using the data generator of the XMark benchmark. The query workloads we used each consisted of 400 queries, generated from the following six base queries.

- **Q1:** Find the name of the person with id "person0".
  for $b in document("XMark/auction0.33.xml")
  /site/people/person[@id="person0"]
  return $b/name/text()

- **Q2:** Give the names and addresses of all people watching the open auction "open_auction0".
  for $p in document("XMark/auction0.33.xml")
  /site/people/person[watches/watch[@open_auction = "open_auction0"]]
  return <person>{$p/name,$p/address}</person>

- **Q3:** How many open auctions has person "person0" placed bids in.
  count(document("XMark/auction0.33.xml")
  /site/open_auctions/open_auction[bidder/
  personref[@person = "person0"]])

- **Q4:** List the initial and current prices of all open auctions where the seller is person "person0".
  for $i in document("XMark/auction0.33.xml")
  /site/open_auctions/open_auction[seller[@person = "person0"]]
  return <open_auction_prices>{$i/initial,$i/current}
  </open_auction_prices>

- **Q5:** List all cosed auctions for the item with id "Item0".

for $i in document(”XMark/auction0.33.xml”)
/site/closed_auctions/closed_auction[Itemref[@Item
= ”Item0”]]
return $i

- **Q6:** Find the prices of all items bought by person “person0”.
for $i in document(”XMark/auction0.33.xml”)
/site/closed_auctions/closed_auction[buyer[@person
= ”person0”]]
return $i/price

The first of these is from the set of 20 benchmark queries. We created the remaining queries ourselves. The reason we didn’t use more benchmark queries is that most of them require scanning large fractions of the source document, and are ill-suited to taking advantage of a cache containing small fragments of the document. In future work, we plan to include support for identifying such queries, and shipping them directly, instead of bringing in huge data fragments into the cache to answer them.

Observe that in each of these queries, the core XPath expression involves a predicate in which an attribute (which is typically an ID or IDREF attribute) is required to have a particular constant value. For example, in Q1, the constant is “person0”, in Q2 it is “open_auction0”, and so on. Note that we have a fixed XML document. Thus, for each such query, we have a fixed set of valid values that the constant can take, which is just the set of values that occur in our source document. For example, for Q1 we could choose from a set of 8414 person constants, for Q2 we could choose from a set of 3959 open auction constants, and so on.

We now describe how we generate our query workloads. For writing each query, we first randomly choose one of the six base queries. We now need to choose one from a set of some N constants for that query. We do that using a Gaussian distribution. The idea is that, the set of constants around the mean form the “hot region” for that query. In the query workload, the query constants will largely come from the respective hot regions, and the cache will exploit precisely this locality in the query workload.

Recall that the parameters of a Gaussian are its mean μ and standard deviation σ. Each base query has its own parameters. The mean is randomly set to some fixed value in the range [1,N] where N is the number of constants we are choosing from for that query. The standard deviation is set to (SD * N) where SD is a parameter in our experiments. SD is basically the standard deviation expressed as a fraction of the length of the interval we are sampling. To choose a constant for a query, we sample a real number using the corresponding Gaussian, and then round it to the nearest integer. Intuitively, the larger SD is, the larger the variation in the set of constants chosen, and the lesser the locality in our query workload.

We used query workloads of 400 queries each, out of which the first 100 were used for warming up the cache i.e., we began measurement of cache hit rate after the first 100 queries. We used Galax as our XQuery engine. Figure 1 shows the variation of cache hit rate with SD (with the maximum cache size fixed at 1 MB). As expected, as SD increases, the cache hit rate drops, since the locality in the query workload decreases. Another quantity of interest to us is the total amount of data brought into the cache from the data source. In a distributed setting, this is the amount of data retrieved over the network. We are interested in comparing this with the total size of the query results, which is the amount of data that would have been retrieved in a pure query shipping architecture. These are plotted in Figure 2, which shows that we do get savings when SD is small.

We now look at measurements made for varying values of the maximum cache size, with SD fixed at 0.001. Figure 3 shows that the cache hit rate in-

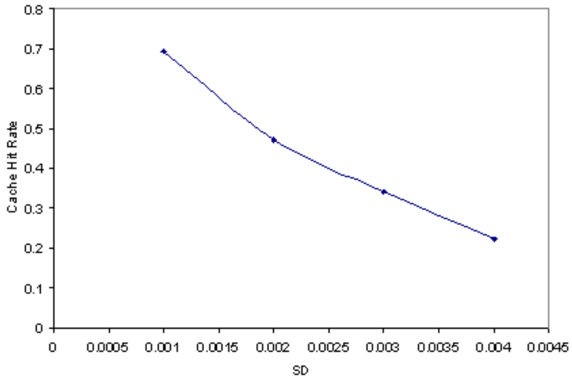

Figure 1: Variation of Cache Hit Rate with Standard Deviation

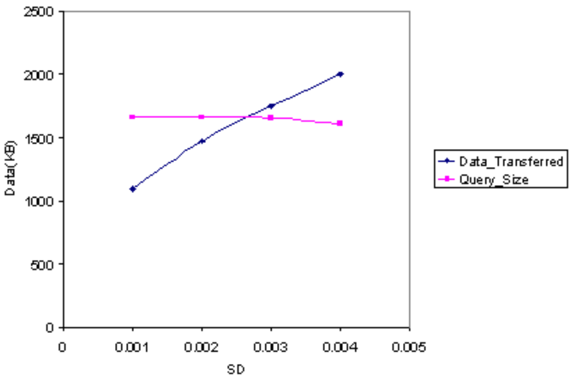

Figure 2:

creases from around 0.3 to 0.7 before settling down. Thus, 0.7 is the best that can be achieved for this particular workload, and a 1 MB cache is enough for this. Figure 4 shows the total data brought in for different cache sizes. The total query result size is constant here since the workload is fixed.

Figure 5 plots the total processing time for the entire query workload. The horizontal line on the top shows the time taken when caching is not used (the

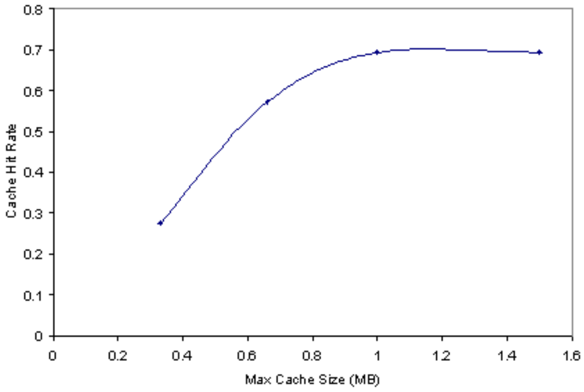

Figure 3: Variation of Cache Hit Rate with Cache Size

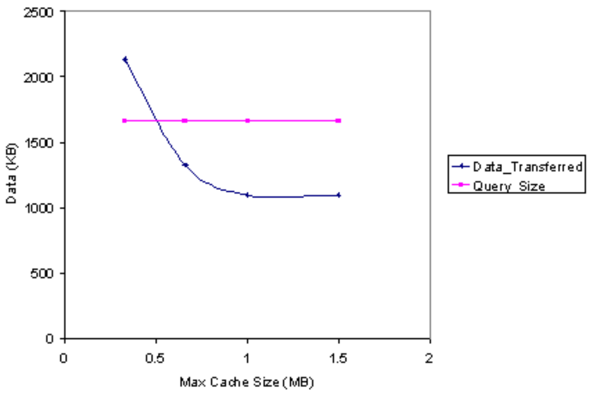

Figure 4:

"conventional" system), and each query is executed directly on the source document. The curve below it plots the processing time for our caching-based system. Observe the huge speedup achieved by our system. For a 1 MB cache, our system takes around 7000 s, compared to around 19000 s for the conventional system. Significant benefits are obtained even when we are just using a small 0.33 MB cache. Recall that our source document is 50 MB. The conventional system runs each query on this complete document. On the other hand, whenever we get a cache hit in our system, the rewritten query runs on an XML fragment whose size is in KB, and thus runs much faster. This explains the huge speedup obtained. The slowness of Galax, which is a main memory XQuery engine, makes this difference even more drastic. Finally, Figure 6 plots the number of entries in the cache at the end of processing.

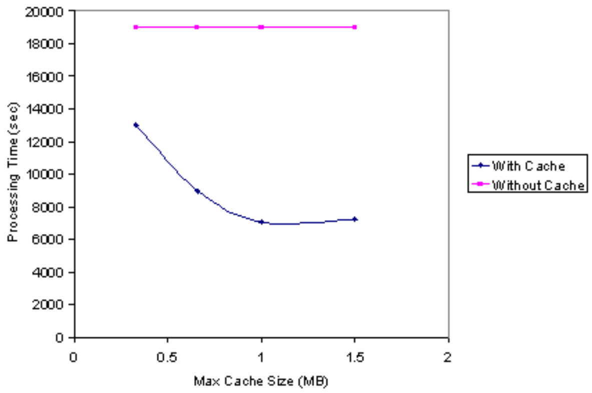

Figure 5: Query Processing Time

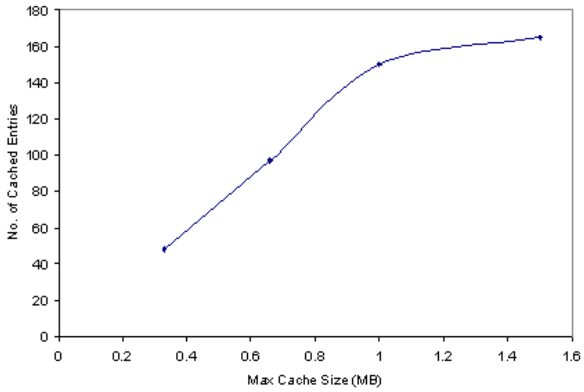

Figure 6: Number of Cached Entries