Every Click You Make, I Will Be Fetching It: Efficient XML Query Processing in RDMS Using GUI-driven Prefetching

Sourav S Bhowmick Sandeep Prakash
Nanyang Technological University, School of Computer Engineering, Singapore
assourav@ntu.edu.sg

1 Introduction

Querying XML data involves two key steps: *query formulation* and *efficient processing* of the formulated query. However, due to the nature of XML data, formulating an XML query using an XML query language such as XQuery requires considerable effort. A user must be completely familiar with the syntax of the query language, and must be able to express his/her needs accurately in a syntactically correct form. In many real life applications it is not realistic to assume that users are proficient in expressing such textual queries. Hence, there is a need for a user-friendly visual querying schemes to replace data retrieval aspects of XQuery. In this paper, we address the problem of efficient processing of XQueries in the relational environment where the queries are formulated using a user-friendly GUI. We take a novel and non-traditional approach to improving query performance by *prefetching data during the formulation of a query in a single-user environment*. The latency offered by the GUI-based query formulation is utilized to prefetch portions of the query results. The basic idea we employ for prefetching is that we prefetch constituent path expressions, store the intermediary results, reuse them when connective is added or "Run" is pressed.

To the best of our knowledge, this is the first work that makes a strong connection between prefetching-based XML query processing and GUI-based query formulation. The key advantages of our approach are as follows. First, our optimization technique is build *outside* the relational optimizer and is orthogonal to any other existing optimization techniques. Hence, our approach provides us with the flexibility to "plug" it on top of any existing optimization technique for processing XML data in relational environment. Second, our approach is not restricted by the underlying schema of the database. As a result, it can easily be integrated with any relational storage approaches. Third, the prefetching-based query processing is transparent from the user. Consequently, there does not exist any additional cognitive overhead to the users while they formulate their queries using the GUI. Finally, our non-traditional approach noticeably improve the performance of XML query execution by 7% to 96%. Moreover, we also show that errors committed by users while formulating queries *do not* significantly affect the query performance.

2 Computing Query Formulation Time

In order to determine the time available for prefetching and to measure the improvement provided by prefetching, the time required to formulate a query visually needs to be measured. This is referred to as the *query formulation time (QFT)*. It is the duration between the time the first predicate is added and the execution of the "Run" command as prefetching can start only when the first predicate is known. In the remaining part of the paper, we use the GUI described in [1] as framework for modeling query formulation time.

We have used the Keystroke-Level Model (KLM)[2] to calculate QFT. The KLM is a simple but accurate means to produce quantitative, *a priori* predictions of task execution time. These times are has been estimated from experimental data [2]. The basic idea of KLM is to list the sequence of keystroke-level actions that the user must perform to accomplish a task, and sum the time required by each action. The KLM has been applied to many different tasks such as text editing, spreadsheets, graphics applications, handheld devices, and highly interactive tasks.

We use the list of average task times for a subset of *physical operators* (K (key-stroking), P (pointing), H (homing), and D (drawing)) as defined by KLM [2]. Using this list of physical operators, we compute the estimated times for a set of *atomic actions* for visual query formulation. The list of tasks the user needs to perform in order to formulate a query using our GUI is basically consists of a set of these atomic actions. The estimated time taken to perform each task is simply the sum of average times of the atomic actions. *Note that QFT does not include higher level mental tasks for formulating a query such as planning a query formulation strategy.* These tasks depend on what cognitive processes are involved, and is highly variable from situation to situation or person to person. We assume that the user has already planned the set of actions he/she is going to take to formulate his/her query and any other mental tasks. This assumption enables us to investigate the impact of prefetching for *minimum* QFT for a particular query. Addition of *mental operators* while formulating a query will only increase the QFT and consequently increase the performance gain achieved due to prefetching. Note that our model for calculating the QFT can as well be used for other types of visual XML query formulation systems.

Error-oblivious QFT (EO_QFT): We first compute QFT in the absence of any query formulation error committed by the user. We call such QFT as *error-oblivious query formulation time* (EO_QFT). The EO_QFT (denoted as T_f) for a query can be calculated as follows: $T_f = 9.9(x_{nj} - 1) + 3.6x_j + 3.8b + 1.3$ where x_{nj} is the number of non-join predicates, x_j is the number of join predicates, b is the number of boolean operators in the query, and 1.3s is the time taken to click on the "Run" icon. Observe that $(x_{nj} - 1)$ is used as prefetching can start only when the first query formulation step is complete in the *Query Editor*.

Error-conscious QFT (EC_QFT): The errors committed by the user while formulating a query are referred to as *query formulation errors* (QFE). Note that QFEs may impact our prefetching approach. Hence, it is necessary to quantify the effect of QFEs by extending EO_QFT with the time lost due to QFEs (*error-conscious* QFT (EC_QFT)). If the user clicks on UNDO n times and corrects a set of mistakes each time then *error-conscious query formulation time* (denoted as T_{fe}) is given by the following equation: $T_{fe} = 9.9(m_{nj} - 1) + 3.6m_j + 3.8m_b + \sum_{s=1}^{n}(2.6 + 1.3i_s + 1.3k_s + T_{u_s}) + 1.3$ where k_s,i_s and T_{u_s} are the number of actions to be modified, the number of times "Insert" button is selected for inserting new predicates, and the total time taken to correct the mistakes respectively, for the s^{th} instance of the UNDO operation. The variables m_{nj}, m_j, and m_b are the number of non-join predicates, number of join predicates, and the number of boolean operators *correctly* added during query formulation respectively. Note that m_{nj}, m_j, and m_b do not include those predicates and boolean operators that contain mistakes or inserted/deleted during UNDO operation. The detailed computation for the above equation is given in [1].

3 GUI-Based Prefetching

In order to expedite XML query processing using such GUI-based prefetching, two key tasks must be addressed. First, given a user-friendly visual query interface, GUI actions that can be used as indicators to perform prefetching need to be identified. These actions are: (1) the addition of a path expression predicate and (2) combining two or more predicates using AND/OR operator to create another complex expression. Second, each GUI action can possibly lead to more than one prefetching operation. Therefore, *a cost-based algorithm that selects the "best" materialization is required.* We use two heuristics for this algorithm. First, we consider only disjunctions of predicates as candidates for temporary materializations. This is because evaluating all possible materializations, though guaranteed to generate a useful materialization, is not feasible. Second, given a materialization space limit L_M, we include the *maximum possible* number of expressions κ_i in the materialization. This is because the greater the number of expressions included in the current materialization the greater the usefulness of the intermediate result towards evaluating the final result. Finally, since each prefetching operation is useful for the next, *existing materializations need to be replaced with new materializations preferably using the previous materializations*. Figure 1 shows the overall prefetching algorithm. The process continues till the user clicks on "Run" to execute the query (line 2). The process waits for changes in the user interface (lines 5 to 8) before selecting new materializations (line 3). Once new materializations are selected, existing ones are replaced (line 4). Details of the algorithm is given in [1].

```
Input: Actions from the query interface.
Output: Intermediate materializations.
1:  State S = getGUIState()
    /*prefetch till user executes query*/
2:  while S != "Execute Query" do
       /*Call materialization selection algorithm*/
3:     selectMaterialization()
       /*Call materialization replacement algorithm*/
4:     replaceMaterialization()
5:     S_1 = getGUIState()
6:     while S_1 == S do /*wait till GUI state changes*/
7:       S_1 = getGUIState()
8:     end while
9:  end while
```

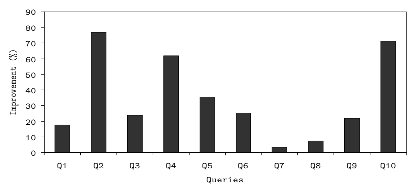
Figure 1. Prefetching algorithm.

4 Performance Study

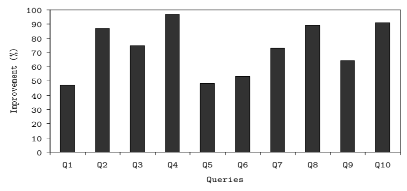
The prototype system of GUI-driven prefetching technique was implemented using JDK1.5. The visual interface was built as a plug-in for the Eclipse platform (www.eclipse.org). The RDBMS used was SQL Server 2000 running on a P4 1.4GHz machine with 256MB RAM. In this paper, we have adopted our schema-oblivious XML storage system called SUCXENT++ (**S**chema **U**n**C**oncious **X**ML **EN**abled Sys**T**em) [3]. The experiments were carried out with tdata sets of size 300MB and 1200MB [1]. Ten queries were used to test the system [1]. These queries vary in the number of predicates, cojunctions/disjunctions and result size.

We define the followings for the experiments. The response time as perceived by the user when prefetching is not employed is called the *normal execution time* (*NET*). The *perceived response time* (*PRT*) is the query response time when prefetching is employed. In the absence of QFEs, we refer to the PRT as *error-oblivious* perceived response time (*EO_PRT*). If QFEs are present then we refer to the PRT as *error-conscious* perceived response time (*EC_PRT*).

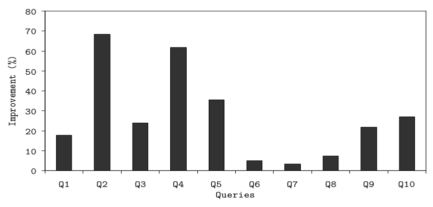
NET vs EO_PRT: This comparison is done as a percentage of improvement over normal execution. It is measured



(a) 300 MB

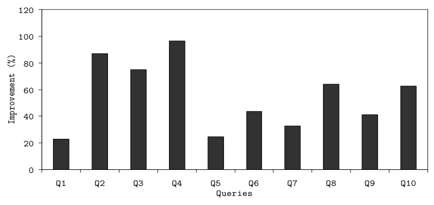


(b) 1200 MB

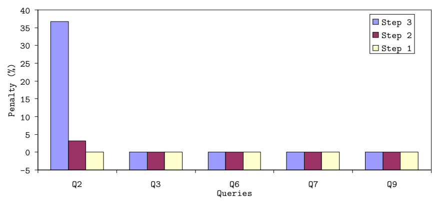


(c) 300 MB

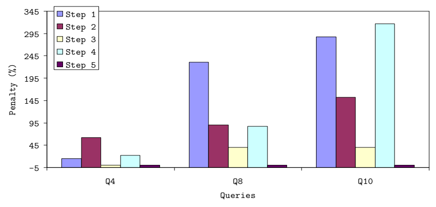
Figure 2. NET vs EO_PRT and EC_PRT vs NET(1).



(a) 1200 MB



(b) 300MB - Three formulation steps



(c) 1200MB - Five formulation steps

Figure 3. EC_PRT vs NET(2) and EC_PRT vs EO_PRT.

as $improvement$ = (1 - $\frac{EO\_PRT}{NET}$)$\times 100$. Figures 2(a) and (b) show the results for the two data sets. There are two main observations. First, the improvement in performance is more for larger data sets. For the 300MB data set the improvement range is 7-76%. This range increases to 47-96% for the 1200MB data set. The second observation is that simple queries (Q1, Q5 and Q9) with one predicate and small result sets benefit the least. Queries with multiple predicates and large result sets benefit the most. This is indeed encouraging as query response time is more critical for large data set. Also queries with disjunctions benefit more than the queries with conjunctions. This is expected as the materialization selection algorithm selects disjunctions as the intermediate results.

NET vs EC_PRT: This comparison is done as a percentage of improvement over normal execution. It is measured as $improvement$ = (1 - $\frac{EC\_PRT}{NET}$)$\times 100$. In this experiment we present the worst-case value for EC_PRT as discussed in [1]. This will give us an estimate of the upper-bound of the effect QFE can have on prefetching. The results are presented in Figures 2(c) and 3(a). The main observation is that EC_PRT is still significantly better than NET for most queries. Also observe that similar to EO_PRT, there is larger improvement for larger data size.

EC_PRT vs EO_PRT: This comparison is done to measure the penalty on PRT due to QFE. It is measured as *penalty* = $\frac{EC\_PRT - EO\_PRT}{EO\_PRT} \times 100$. Again, the worst case value of EC_PRT is used for comparison. Particularly, we measure EC_PRT for $q = n - 1$ (UNDO operation is invoked just before clicking "Run") where n is the number of query formulation steps and the user clicks on "Run" at nth step. We vary the *error realization distance* which is defined as follows. Suppose that the error is committed at pth step and the UNDO operation is invoked at qth step where $0 < p < q \leq n - 1$. Then, the *error realization distance* is defined as $q - p$. Figures 3(b) and 3(c) show the results for the 300MB and 1200MB data sets. Figure 3(b) shows the results for queries that have three formulation steps (two predicates and a conjunction/disjunction) other than clicking on "Run". The three values shown for each query measure the penalty when the error was committed at the first step, the second step and the third step respectively. The $penalty$ axis starts at -5 to allow the display of cases where $penalty = 0$. Figure 3(c) shows the results for queries with five formulation steps.

The results shown highlight two main points. First, QFE generally has a greater effect with the increase in error realization distance. This is expected as an early mistake will lead to more materializations being recalculated. However, there are some exceptions. The query Q4 for the 1200MB data set shows an increase as the evaluation of the second predicate is more expensive than the first. Second, the impact of QFE increases with data set size. The 1200MB data set shows a maximum increase of 316%. This can be attributed to the higher cost of reevaluating materializations for the larger data set.

References

[1] S. S. BHOWMICK AND S. PRAKASH. Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching in a Single-User Environment. *Tech. Report*, School of Comp. Engg, NTU, 2005 (Available at `www.ntu.edu.sg /home/assourav/papers/cais-03-2005-TR.pdf`).

[2] S. K. CARD, T. P. MORAN, AND A. NEWELL. The Keystroke-level Model for User Performance Time with Interactive Systems. *Commun. ACM*, 23(7):396–410, 1980.

[3] S. PRAKASH AND S. S. BHOWMICK. Efficient Recursive XML Query Processing in Relational Databases. *In ER*, 2004.