Efficient XML Query Processing in RDBMS Using GUI-Driven Prefetching in a Single-User Environment

Sandeep Prakash[1], Sourav S. Bhowmick[1,2], Klarinda G. Widjanarko[1,2], and C. Forbes Dewey Jr.[3]

[1] School of Computer Engineering, Nanyang Technological University, Singapore
[2] Singapore-MIT Alliance, Nanyang Technological University, Singapore
[3] Division of Biological Engineering, Massachusetts Institute of Technology, USA
{assourav,klarinda}@ntu.edu.sg, cfdewey@mit.edu

Abstract. In this paper, we address the problem of efficient processing of XQueries in single-user 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*. The latency offered by GUI-based query formulation is utilized to prefetch portions of the query results. To realize this, we present an algorithm for prefetching based on data synopses statistics and GUI actions during visual query formulation. Experimental evaluation indicates that prefetching is viable as the combined time taken by all the prefetching operations is not significantly more than normal query execution time. Our experiments in the context of biological data show that prefetching improves the query response time by 7-96% with a greater improvement for larger data sets. Also, we show the impact of errors committed by users during query formulation on the query performance.

1 Introduction

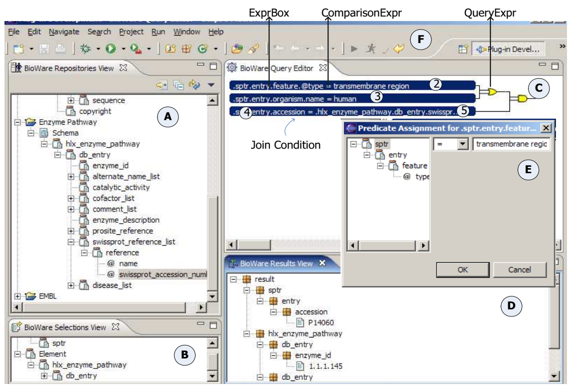
Querying XML data involves two key steps: *query formulation* and *efficient processing* of the formulated query. However, due to the nature of XML data, formulating an XML query using an XML query language such as XQuery requires considerable effort. A user must be completely familiar with the syntax of the query language, and must be able to express his/her needs accurately in a syntactically correct form. In many real life applications (such as life sciences) 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. The work presented here is part of our ongoing research of building a system called *Da Vinci's Notebook* that would empower biologists to explore huge volumes of experimental biology data. 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. 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. Second, each GUI action can possibly lead to more than one prefetching operation. Therefore, an algorithm needs to be designed to select the "best" operation. In this paper, we address these issues in detail. A short overview of this approach appeared as a poster paper in [3].

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 built *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. As we shall see in Section 5, our experiments with biological data indicate a performance improvement of 7% to 96% with an increasing improvement as the size of the data grows. Moreover, we also show that errors committed by users while formulating queries *do not* significantly affect the query performance.

2 Visual Query Interface

In this section, we present the visual interface which we shall use in the rest of the paper for formulating XML queries. Ideally, a full implementation of the GUI-driven prefetching system would require a fully-functional XQuery support. However, it is also true that a visual interface is useful when it serves the needs of the majority of the users


ExprBox
ComparisonExpr
QueryExpr
F
BioWare Repositories View
BioWare Query Editor
.sptr.entry.feature.@type = transmembrane region
.sptr.entry.organism.name = human
.sptr.entry.accession = .hlx_enzyme_pathway.db_entry.swisspr
2
3
4
5
C
A
Join Condition
Predicate Assignment for .sptr.entry.featur...
E
OK
Cancel
BioWare Results View
D
BioWare Selections View
B

(a) Query formulation.

```
for $b in /sptr/entry, $c in /enzyme/entry
where
  ($b/feature/@type='transmembrane region' or
   $b/organism/name='human') and
   $c/swissprot_reference/@swissprot_accession =
   $b/accession
return $b/sequence, $c/enzyme_id
```

(b) XQuery representation.

Fig. 1. Visual query interface and XQuery representation

in expressing majority of their queries, which are typically simple [1]. A complete but too complex graphical interface would fail both in replacing the textual language and in addressing all the users' needs [1]. Furthermore, the focus of this paper is to study the effect of GUI-driven prefetching on XML query processing and not design of a complete visual interface for formulating XML queries. Hence, we implemented an interface that supports simpler types of XQuery. These queries are sufficient to justify the positive contributions made by the GUI-based prefetching technique. Specifically, the syntax of the basic XQuery query that can be formulated using our GUI is as follows. Note that we assume that the DTDs/XML schemas of data sources are available to the user during query formulation.

$$\begin{array}{rl} \texttt{FOR} & x_1 \; in \; p_1, \ldots, x_n \; in \; p_n \\ \texttt{WHERE} & W \\ \texttt{RETURN} & r_1, r_2, \ldots, r_k \end{array}$$

where p_i is a simple linear path expression, W is a set of predicates that are connected by AND/OR operator(s). A predicate $w \in W$ can be one of the two forms: $s_i \; op \; c$ or $s_i \; op \; s_j$ where s_i and s_j are path expressions that may contain a selection predicates and c is a constant. The variable r_i is a simple path expression.

Our system allows the user to formulate visual queries in an intuitive manner without having to learn any query language. The user interface (Figure 1(a)) is presented as an adjustable multi-panel window comprising the following items. The *Repositories View* (labelled A) occupies the left pane. It serves as a data source browser in which the user can view the list of available data sources and their respective structures in terms of a tree display of their DTD/XML Schema. Showing multiple data sources allows the formulation of queries spanning more than one source. The data sources shown in Figure 1(a) are SWISSPROT and ENZYME.

The *Query Editors* are stacked in the middle pane (labeled C), with tabs for navigating between queries. It enables the user to specify the WHERE clause. The user drags the node to be queried from the *Repositories View* and drops it in a *Query Editor*. A *Condition Dialog* (labeled E), appears and the user is expected to fill in the condition that should be satisfied by the selected node. In Figure 1(a), the selected node is /sptr/entry/feature/@type and the condition is "=transmembrane region" thus forming the predicate .sptr.entry.feature.@type= " transmembrane region" (labeled 2). This expression is called Comparison Expr and the visual representation of a ComparisonExpr type is referred to as ExprBox.

The user can combine two or more visual components that represent the ComparisonExpr by dragging a region around them and assigning an AND or OR condition. In Figure 1(a), the first two ComparisonExpr (labeled 2 and 3) are combined using the OR operator thus forming the QueryExpr (.sptr.entry.feature.@type="transmembrane region" OR .sptr.entry.organism.name="human"). In order to specify a join condition two nodes, each representing one side of the join condition are selected and dragged on to the *Query Editor*. This is shown by the labels 4 and 5 in Figure 1(a). The visual representation of a QueryExpr type is also referred to as ExprBox.

The *Selections View* (labeled B) is a drop target for nodes dragged from the *Repositories View* and displays the nodes that will be visible in the result of the query. This enables the visual formulation and representation of the XQuery RETURN clause. The user can execute the query by clicking on the "Run" icon in the *Query Toolbar*. The *Results View* (labeled D) displays the query results.

To formulate a query, the user first selects the nodes that should be present in the RETURN clause. For instance, in Figure 1(a), the nodes selected are sequence and enzyme_id indicating that the user only wants to view these elements in the result. Next, the predicates in the WHERE clause are formulated in the *Query Editor*. The visual constructs in the *Query Editor* and *Selections View* need to be translated to formulate a complete XQuery. Each ComparisonExpr or QueryExpr can be combined to obtain a Query type. The translation to XQuery can be easily done by following the syntax presented earlier. Figure 1(b) shows the XQuery corresponding to Figure 1(a).

3 Computing Query Formulation Time

Our query processing approach utilizes the user's query formulation time to prefetch results of the intermediate queries. 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.

We have used the Keystroke-Level Model (KLM)[4] 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 [4]. 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 [4,6].

Figure 2(a) lists average task times for a subset of *physical operators* (K (key-stroking), P (pointing), H (homing), and D (drawing)) as defined by KLM [4]. Figure 2(b) depicts the estimated times for a set of *atomic actions* for visual query formulation. Note that the times are computed using the physical operators in Figure 2(a). Figure 2(c) shows the list of tasks the user needs to perform in order to formulate a query. Each task consists of a set of atomic actions (Figure 2(b)). For example, adding a join predicate (Task $T2$) involves selecting the two join nodes (Action $A1$ twice) and dragging them on to the *Query Editor* (Action $A2$). 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. That, is our QFT in the following discussion consists of a sequence of physical operators only. This assumption enables us to investigate the impact of prefetching for *minimum* QFT for a particular

Notation	Physical Operator	Average Time (s)
K	Keystroke	0.28
T(n)	Type a sequence of n characters on a keyboard	n x K
P	Point with mouse to a target on the display	1.1
B	Press or release mouse button	0.1
BB	Click mouse button	0.2
H	Moving the hand between keyboard and mouse	0.4

(a) Keystroke-Level Model

Task ID	Set of Task for QF	Sequence of Actions	Average Time (s)
T1	Add non-join predicate	<A1, A2, A3, A4, A5>	A1+A2+A3+A4+A5 = 9.9
T2	Add join predicate	<A1, A1, A2>	2A1+A2 = 3.6
T3	Combine predicate with AND/OR	<A9,A10,A5>	A9+A10+A5 = 3.8
T4	Add a RETURN clause element	<A1,A2>	A1+A2 = 2.4

(c) Average execution times for query formulation tasks

Undo ID	Task	Sequence of Actions	Average Time (s)
U1	Modify the LHS of a non-join predicate	<A1, A2, A5>	A1+A2+A5 = 3.7
U2	Modify the RHS of a non-join predicate	<A4, A5>	A4+A5 = 6
U3	Modify the comparison operator of a non-join predicate	<A3, A5>	A3+A5 = 2.8
U4	Modify the LHS or RHS of a join predicate	<A1, A2, A5>	A1+A2+A5 = 3.7
U5	Change a AND to a OR (or vice versa)	<A10>	A10 = 1.3
U6	Deleting a predicate/RETURN clause	<A5> (Click delete button in UNDO box)	A5=1.3

(d) Average execution times for UNDO tasks

Action ID	Atomic Actions	Sequence of physical operator	Average Time (s)
A1	Select predicate node	(a) Move the mouse on the node (P) (b) Press mouse button (B)	P+B=1.2
A2	Drag and drop predicate node	(a) Move the mouse to Query Editor (P) (b) Release mouse button (B)	P+B=1.2
A3	Selection of comparison condition in condition dialog box	(a) Move the mouse to V button (P) (b) Click mouse button (BB) (c) Click mouse button on selected condition (BB)	P+2BB=1.5
A4	Type comparison value (avg 10 characters)	(a) Move the mouse to text box (P) (b) Moving the hand between keyboard and mouse (H) (c) Type characters (T(10)) (d) Moving the hand between keyboard and mouse (H) [for subsequent action]	P+2H+T(10) = 1.1 + 0.8 + 2.8 = 4.7
A5	Click on a button in the combo box	(a) Move the mouse on the button (P) (b) Click mouse button (BB)	P+BB=1.3
A6	Select action to UNDO	(a) Move the mouse to UNDO icon (P) (b) Click mouse button (BB)	P+BB=1.3
A7	Click on UNDO	(a) Move the mouse to the action (P) (b) Click mouse button (BB)	P+BB=1.3
A8	Click on RUN	(a) Move the mouse to RUN icon (P) (b) Click mouse button (BB)	P+BB=1.3
A9	Drag predicate in Query Editor (for AND/OR clause)	(a) Drag mouse to other predicate (P) (b) Release mouse (B)	P+B=1.2
A10	Select AND/OR operator	(a) Move mouse on the AND or OR icon (P) (b) Click mouse button (BB)	P+BB=1.3

(b) Average execution times for atomic actions

Fig. 2. Query formulation times using Keystroke-Level model

query. Addition of *mental operators* while formulating a query will only increase the QFT and consequently increase the performance gain achieved due to prefetching. In other words, in this paper we investigate the benefits of prefetching for "worst case" QFT (without mental operators).

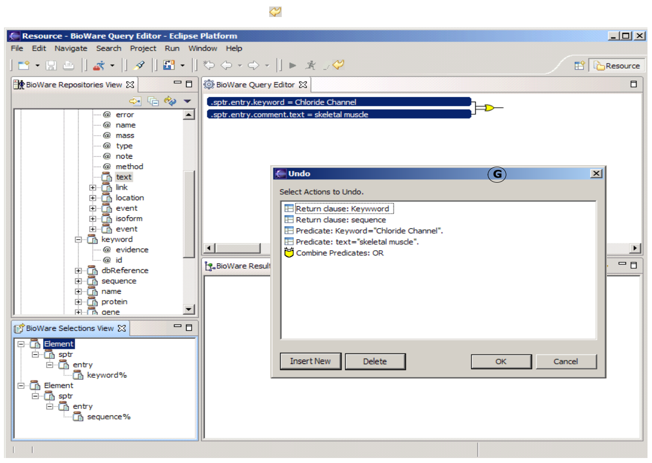
We first compute QFT in the absence of any query formulation error committed by the user. We call such QFT as *error-oblivious query formulation time* (EO_QFT). Note that our model for calculating the QFT can as well be used for other types of visual XML query formulation systems (such as XQBE [1]). This is because similar actions would be required to formulate a query.

3.1 Error-Oblivious QFT (EO_QFT)

Based on the timings (Figures 2(b) and 2(c)) discussed above the EO_QFT (denoted as T_f) for a query can be calculated as follows:

$$T_f = 9.9(x_{nj} - 1) + 3.6x_j + 3.8b + 1.3 \tag{1}$$

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 (Action $A8$ in Figure 2(b)). Observe that $(x_{nj} - 1)$ is used as prefetching can start only when the first query formulation step is complete in the *Query Editor*. That is, QFT does not include the time taken to add the `RETURN` clause. This is because if prefetching were to start as soon as the `RETURN` clause were added, it is possible to retrieve very large results many of which may not be relevant eventually as `WHERE` clause predicates are yet to be added in the *Query Editor*. Fortunately, as we shall in Section 5, we achieve significant performance improvement even though we postpone the prefetching till addition of a `WHERE` clause predicate in the *Query Editor*.



Select Actions to Undo.

Return clause: Keywword
Return clause: sequence
Predicate: Keyword="Chloride Channel".
Predicate: text="skeletal muscle".
Combine Predicates: OR

Fig. 3. Undo operation

3.2 Error-Conscious QFT (EC_QFT)

The above approach used to calculate error-oblivious QFT does not take into account errors committed by the user. These errors 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. We first discuss how the GUI enables the user to correct queries by undoing certain actions. Then, we compute the *error-conscious* QFT (EC_QFT) that incorporates QFE.

Figure 3 shows the interface presented to the user. When the user discovers a mistake he/she clicks on the UNDO icon (labeled F in Figure 1(a)). The user is then presented with the list of actions he/she has performed (labeled G in Figure 3). For example, in Figure 3 the list shows that the user has added two predicates and combined them using a conjunction. The user then selects the action(s) to be corrected. Suppose the user wanted the second predicate to be `.sptr.entry.comment.text="cardiac muscle"` instead of `.sptr.entry.comment.text="skeletal muscle"` in Figure 3. Consequently, the user has to modify the predicate by replacing `"skeletal muscle"` with `"cardiac muscle"`. In general, a user will execute the following steps to rectify a mistake.

Step 1 (*Click on the* UNDO *icon*): This takes $1.3s$ ($A7$ in Figure 2(b)).

Step 2 (*Select the action(s) to modify*): The user may select an action to update or delete by clicking on it or he/she may click the “Insert” button to insert new predicate(s) in the `WHERE` and `RETURN` clauses. Each action selection for update or delete will take at most $1.3s$ ($A6$ in Figure 2(b)). As there can be k number of actions to be modified, the total time will be $1.3k$ seconds. The time taken to click “Insert” button is $1.3s$ ($A5$ in Figure 2(b)). If there are i such clicks then the total time is $1.3i$. The time taken to insert new non-join/join predicate(s) is $(9.9i_{nj} + 3.6i_j)$ (Equation 1). Note that addition of `AND`/`OR` operators will be included by Step 3. The time taken to insert r `RETURN` clause elements is also $2.4r$ ($T4$ in Figure 2(c)). If "Insert" button is pressed then Step 3 is ignored by the user.

Step 3: In this step, some of the actions in Figure 2(d) need to be taken if the user selects action(s) for update or delete.

Step 4 (*Click on "OK" to accept the changes*): This will take $1.3s$ ($A5$ in Figure 2(b)) and will have to be done for each modification. As a result, the total time taken for this operation is $1.3 \times \Re$ where $\Re = (i_{nj} + i_j + r + p_\ell + p_r + p_c + p_j + p_d + p_b)$ and $p_\ell, p_r, p_c, p_j, p_d, p_b$ are numbers of times corrections $U1$, $U2$, $U3$, $U4$, $U5$, and $U6$ in Figure 2(d) are made respectively.

Step 5 (*Click on "OK" button in Figure 3*): This takes $1.3s$ ($A5$ in Figure 2(b)).

Therefore, *each* time the UNDO icon is clicked and a set of mistakes is corrected, the *additional* time taken for formulating a query will be $(2.6 + 1.3k + 1.3i + T_u)$ where $0 < k \leq \Re$, $i \geq 0$ and

$$\begin{aligned} T_u &= 9.9i_{nj} + 3.6i_j + 2.4r + 3.7p_\ell + 6p_r + 2.8p_c + 3.7p_j + 1.3p_d + 1.3p_b + 1.3\Re \\ &= 11.2i_{nj} + 4.9i_j + 3.7r + 5p_\ell + 7.3p_r + 4.1p_c + 5p_j + 2.6p_d + 2.6p_b \qquad (2) \end{aligned}$$

The query formulation time T_f can now be extended to incorporate QFEs. 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.

$$T_{fe} = 9.9(m_{nj} - 1) + 3.6m_j + 3.8m_b + \sum_{s=1}^{n} (2.6 + 1.3i_s + 1.3k_s + T_{u_s}) + 1.3 \quad (3)$$

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, 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.

4 GUI-Based Prefetching

We now describe our approach to improving query performance by utilizing the latency offered by GUI-based query formulation. Given an XML document and a path expression P the *Path Count* (denoted as $C(P)$) is defined as the number of leaf nodes that satisfy P. The $C(P)$ value for a non-root-to-leaf path P is $\sum_{j=1}^{k} C(P_j)$ where $P_1, P_2, \ldots, P_k$ are the root-to-leaf paths that satisfy P. Note that, as $C(P)$ increases so does the I/O cost of a query that contains P as one of its path expressions. The *Total Path Count* for an XML document is defined as $T = \sum_{j=1}^{N} C(P_j)$ where N is the number of distinct root-to-leaf paths in the XML document. Next, we define the notion of *value selectivity*. Given an XML document and a root-to-leaf path P, *value selectivity* $V(P)$ is defined as the number of nodes in the XML document with path P that have unique text values.

```
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
```

(a) Prefetching algorithm.

```
Input:  Expressions $K = \{\kappa_1, \kappa_2, ..., \kappa_n\}$ in descending order of cost($\kappa_i$).
        Materialization limit $L_M$.
Output: Coefficient of each $\kappa_i$ in the final materialization.
1:  start = 0, end = $2^n$ - 1, middle
2:  while start < end do
3:    middle = (start + end)/2
4:    /*GetSelection(order, n) generates
5:      the coefficients for order^th combination out of $2^n$-1.*/
6:    S = GetSelection( middle, n )
7:    $l_S = \sum_{j=0}^{n} s_j \times \text{cost}(\kappa_j)$
8:    if $l_m > L_M$ then
9:      end = middle - 1
10:   else if then
11:     start = middle + 1
12:   end if
13: end while
14: return GetSelection(middle, n)
```

(b) Algorithm selectMaterialization.

Fig. 4. Algorithms for prefetching

Based on the above definitions, the cost of evaluating a `QueryExpr` κ, denoted as $cost(\kappa)$, can be calculated as follows. (1) If κ `::= PathExpr (ValueComp) Literal` then the usual procedure to estimate the I/O cost is followed. When `ValueComp` is `"="` or `">"` $cost(\kappa) = C(P)/V(P)$ or $cost(\kappa) = C(P)/3$ respectively [5], where P is the parameter of type `PathExpr`. This can be extended to other types of `ValueComp`. (2) If κ `::= PathExpr (ValueComp) PathExpr` and the two `PathExpr` types are denoted as P_1 and P_2 then $cost(\kappa) = \frac{C(P_1)}{V(P_1)} \times \frac{C(P_2)}{V(P_2)}$. (3) If κ `::= ComparisonExpr (`$\wedge$`) ComparisonExpr` and the two `ComaprisonExpr` types are denoted as κ_1 and κ_2 then the probability that κ_i $(i = 1, 2)$ is satisfied is $\frac{cost(\kappa_i)}{T}$. Therefore, $cost(\kappa) = \frac{cost(\kappa_1) \times cost(\kappa_2)}{T}$. (4) If κ `::= ComparisonExpr (`$\vee$`) ComparisonExpr` and the two `ComaprisonExpr` types are denoted as κ_1 and κ_2 then $cost(\kappa) = cost(\kappa_1) + cost(\kappa_2)$. Note that the last two formulae can be extended for any number of conjunctions and disjunctions.

4.1 Prefetching Algorithm

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 realize this, the prefetching algorithm needs to perform prefetching operations at *certain* steps. In order to perform these operations, prefetching friendly GUI actions need to be identified first. Recall from Section 2, when a user formulates a query, constructs of types `QueryExpr` and `ComparisonExpr` are created. These types are parts of the final query and, therefore, are candidates for temporary materializations. Therefore, GUI actions that result in the addition of these types are also indicators for prefetching. These actions are: (1) the addition of an `ExprBox` and (2) combining two or more `ExprBox` types to create another `ExprBox` type that corresponds to a `QueryExpr` type.

Next, given a GUI state, *the optimal prefetching operations need to be determined*. 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 4(a) 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

```
Input: GUI state, last GUI operation op and
       current set of materializations $M$.
Output: Updated $M$.

1:  if op is add then
2:     selectMaterialization() /* refer to Figure 4(b) */
3:  end if
4:  if op is combine then
5:     $e_1$ and $e_2$ are the combined ExprBox types
6:     $m_1$ and $m_2$ are the corresponding materializations.
7:     if op is OR then
8:        if $m_1$ and $m_2$ then
9:           $m = m_1 \cup m_2$
10:          $M = M - (m_1) - m_2$
11:          $M = M \cup m$
12:       end if
13:       if $!m_1$ or $!m_2$ then
14:          $M$ = selectMaterialization()
15:       end if
16:    end if
17:    if op is AND then
18:       if $m_1$ and $m_2$ then /* $m_1$ and $m_2$ have already been materialized */
19:          $m$ = GetCommonNodes($m_1$,$m_2$)
20:          $M = M - (m_1) - m_2$
21:          $M = M \cup m$
22:       end if
23:       if $!m_1$ or $!m_2$ then /* $m_1$ or $m_2$ or both are not materialized yet */
24:          $M$ = selectMaterialization()
25:          if $(m_1 \wedge m_2) \in M$ then
                /* Use $m_1$ or $m_2$ to generate the new SQL query */
26:             SQL s = SQL query using only $m_1$ and $m_2$.
27:             $M = M - (m_1$ and $m_2)$
28:             materialize s.
29:             $M = M \cup$ s
30:          end if
31:       end if
32:    end if
33: end if
34: if op is UNDO then
35:    Cancel ongoing materialization.
36:    $M_D$ = completed materializations dependant on step being corrected.
37:    for all $m_d \in M_D$ do
38:       Delete $m_d$.
39:    end for
40: end if
    /*materialize the new ComparisonExpr or QueryExpr types in $M$*/
41: materializeNew()
```

Fig. 5. Algorithm replaceMaterialization

the user interface (lines 5 to 8) before selecting new materializations (line 3). Once new materializations are selected, existing ones are replaced (line 4).

Materialization Selection: At any given step during query formulation there can be more than one materialization option. Therefore, an algorithm that selects the "best" materialization is required. We begin by presenting two heuristics that are used in our algorithm.

*Heuristic 1:*We consider only disjunctions of `ComparisonExpr` and `QueryExpr` as candidates for temporary materializations. We elaborate on the rational behind this heuristic now. While formulating queries the GUI contains n `ComparisonExpr` and `QueryExpr` types (denoted as κ_i where $i = 1 \ldots n$. Then, the possible materializations are $(\kappa_1 \vee \kappa_2 \vee \kappa_3 \vee \ldots \vee \kappa_n)$, $(\kappa_1 \wedge \kappa_2 \vee \kappa_3 \vee \ldots \vee \kappa_n)$, $(\kappa_1 \wedge \kappa_2 \wedge \kappa_3 \vee \ldots \vee \kappa_n)$ and so on. The number of possible combinations is 2^{n-1}. Obviously, evaluating all possible materializations, though guaranteed to generate a useful materialization, is not feasible. Therefore, only disjunctions are generated. This is because given $\kappa_1, \ldots, \kappa_n$, $(\kappa_1 \wedge \kappa_2 \wedge \ldots \wedge \kappa_n)$ can be evaluated from the materialization of $(\kappa_1 \vee \kappa_2 \vee \ldots \vee \kappa_n)$.

Heuristic 2: 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.

Based on the above heuristics we define the notions of *materialization selection* and the *optimality* of a materialization selection. Given κ_1, $\kappa_2 \ldots \kappa_n$, a *materialization selection* is defined as $S = \{\mu_1, \mu_2, \ldots, \mu_n\}$ where $\mu_i \in \{0, 1\}$ and the cost associated with the selection (which is the same as the result size) is calculated as $l_S = \sum_{i=1}^{n} cost(\kappa_i) \times \mu_i$. Essentially, an expression κ_i is included in the materialization if $\mu_i = 1$. The cost l_S is a summation as only disjunctions are considered based on Heuristic 1.

The *optimality* of a materialization selection, denoted as $\Theta(S)$, is defined as follows. Given two materialization selections $S_a = \langle \mu_{a_1}, \mu_{a_2}, \ldots, \mu_{a_n} \rangle$ and $S_b = \langle \mu_{b_1}, \mu_{b_2}, \ldots, \mu_{b_n} \rangle$, $\Theta(S_a) > \Theta(S_b)$ if and only if $(\sum_{i=1}^{n} \mu_{a_i} > \sum_{i=1}^{n} \mu_{b_i}) \vee (\sum_{i=1}^{n} \mu_{a_i} = \sum_{i=1}^{n} \mu_{b_i} \wedge l_{S_a} > l_{S_b})$. This optimality condition satisfies Heuristic 2. We elaborate on the usefulness of this with an example. Consider a GUI state with three expressions κ_1, κ_2 and κ_3 such that $cost(\kappa_1) > cost(\kappa_2) > cost(\kappa_3)$. The most desirable materialization selection would be $S = \{1, 1, 1\}$ as it will include all the expressions. However, if $l_S > L_M$ then selections with only two expressions will have to be considered. Then, the optimal materialization would be $S = \{1, 1, 0\}$ as it includes the expressions that will yield the largest result. This can be extended to generate the sequence $\Theta(\{1,1,1\}) > \Theta(\{1,1,0\}) > \Theta(\{1,0,1\}) > \ldots > \Theta(\{0,0,1\}) > \Theta(\{0,0,0\})$. Note that this sequence can be generated for any number of expressions n.

The algorithm is shown in Figure 4(b). The input to the algorithm is the list of `ComparisonExpr` and `QueryExpr` types, κ_i, currently present in the GUI. They are listed in decreasing order of $cost(\kappa_i)$ as discussed above. Essentially, the algorithm performs a binary search over this sequence to determine the best materialization given the limit L_M. Notice that the sequence need not be pre-generated. The $GetSelection$ method returns a selection S given its order in the sequence and the number of expressions n. For example, in the case where $n = 3$, $GetSelection(3, 3)$ would return $\{1, 0, 1\}$ - the third selection for three rules. Similarly, $GetSelection(1, 3)$ would return $\{1, 1, 1\}$. It can be shown that the overall time complexity of the algorithm is $O(n^3)$. Once the list of expressions is selected by the algorithm a separate materialization, denoted as $\mathcal{M}_{\kappa_i}$, is maintained for each κ_i. Note that a disjunction of the selected κ_is could be maintained instead. However, the cost for both is approximately the same and is equal to $\sum cost(\kappa_i)$.

Materialization Replacement: Once the optimal materialization to replace the current state is selected it needs to be generated preferably using the results from the previous materializations. The materialization replacement algorithm is presented in Figure 5. The worst case complexity of the replacement algorithm without executing the new materialization is $O(n^3)$ - when `selectMaterialization()` is called. The overall time taken depends on the execution time the SQL query(s) corresponding to the new materialization.

5 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. As mentioned in Section 1, our approach can be built on any XML-to-relational storage mechanism. In this paper, we have adopted our schema-oblivious XML storage system called SUCXENT++ [8].

The experiments were carried out with three data sets of size 300MB, 600MB and 1200MB respectively generated by combining the data sets shown in Figure 6(a). The 300MB data sets was generated using 150MB each of the SWISS-PROT and EMBL data sets. The 600MB data set was generated using 300MB each and the 1200MB data set

Data	URL	Size (MB)	Node Count	Leaf Count	Depth
Swiss-Prot	http://us.expasy.org	600	26,035,096	17,385,288	6
EMBL	http://ebi.ac.uk	600	15,265,784	13,460,524	6
Enzyme	http://ebi.ac.uk	3	86,413	74,892	7
Total		1203	41,387,293	30,920,704	

(a) Data Set

#	Query	Characteristic	T_f (s)	Result Size

Q1	for $b in /sptr/entry
	where $b/protein/name = 'Sesquiterpene Cyclase'
	return $b/accession
		- Database: Swiss-Prot
		- single non-join predicate
		- small result size
			1.3	3

Q2	for $b in /sptr/entry
	where $b/feature[@type = 'transmembrane region'] and $b/organism/name = 'human'
	return $b/accession
		- Database: Swiss-Prot
		- two non-join predicates
		- AND operator
		- large result size
			11.7	2838

Q3	for $b in /sptr/entry
	where ($b/keyword = 'Chloride Channel' or $b/comment/text = 'skeletal muscle')
	return $b/accession,$b/sequence
		- Database: Swiss-Prot
		- two non-join predicates
		- OR operator
		- small result size
			14.8	145

Q4	for $b in /sptr/entry
	where ($b/keyword = 'Chloride Channel' or $b/comment/text = 'skeletal muscle')
	and $b/organism/name='human'
	return $b/accession,$b/sequence
		- Database: Swiss-Prot
		- three non-join predicates
		- AND/OR operator
		- small result size
			24.9	43

Q5	for $b in /embl/entry
	where $b/keyword = "%gene%"
	return $b/accession
		- Database: EMBL
		- single non-join predicate
		- large result size
			1.3	3349

Q6	for $b in /embl/entry
	where $b/source/organism='Homo Sapiens'
	and $b/keyword = '%gene%'
	return $b/accession
		- Database: EMBL
		- two non-join predicates
		- AND operator
		- large result size
			11.98	3278

Q7	for $b in /embl/entry
	where $b/descr = '%gene%' or $b/keyword = '%gene%'
	return $b/accession
		- Database: EMBL
		- two non-join predicates
		- OR operator
		- large result size
			11.98	3883

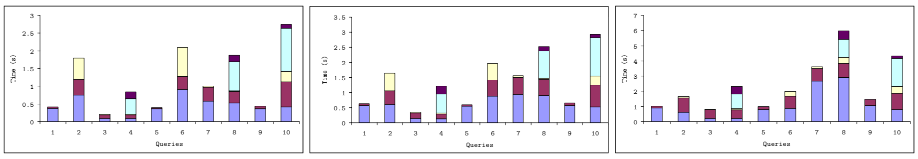
Q8	for $b in /embl/entry
	where ($b/descr = '%gene%' or $b/keyword = '%gene%') and $b/source/organism = "Homo Sapiens"
	return $b/accession
		- Database: EMBL
		- three non-join predicates
		- AND/OR operator
		- large result size
			24.34	3596

Q9	for $b in /sptr/entry, $c in /embl/entry
	where $b/protein/name = 'Sesquiterpene Cyclase' and $b/dbReference[@id=$c/accession]
	return $b/accession, $c/accession
		- Database: Swiss-Prot, EMBL
		- single join predicate
		- AND operator
			8.7	3

Q10	for $b in /sptr/entry, $c in /enzyme_pathway/entry, $d in /embl/entry
	where $d/keyword = '%gene%' and $b/accession=$c/swissprot_reference/reference and $b/dbreference[@type="EMBL"]
	and $b/dbReference[@id=$d/accession]
	return $b/accession, $c/accession
		- Database: Swiss-Prot, Enzyme and EMBL
		- two join predicates
		- three Boolean operators
			26.22	68

(b) Queries

Fig. 6. Data set and queries



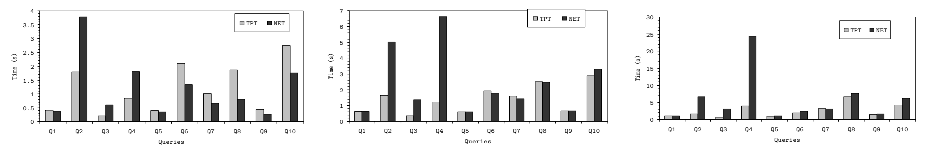
(a) 300 MB (b) 600 MB (c) 1200 MB

Fig. 7. Materialization replacement cost

was generated using the complete data sets. The 3MB ENZYME data set was used in all experiments. It is not reflected in the respective sizes due to its much smaller size. Ten queries were used to test the system. The list of queries together with their EO_QFT values and query results size for 1200MB data is shown in Figure 6.

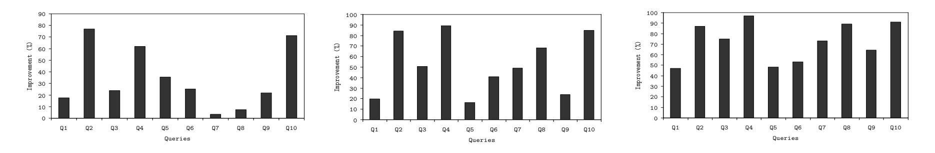
We now define few terms that are used in the subsequent discussion. The response time as perceived by the user when prefetching is not employed is called the *normal execution time* (*NET*) (denoted as T_n). The *perceived response time* (*PRT*) is the query response time when prefetching is employed. 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*). The total time taken for all prefetching operations is called *total prefetching time* (*TPT*). Next we define the notion of *error realization distance*. Consider a query with n formulation steps where the user clicks on "Run" at nth step. 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*, denoted as ϵ, is defined as $\epsilon = q - p$.

Materialization Replacement Cost: Figure 7 shows the results of materialization replacement cost. Here the running times of individual materialization operations are presented. Each section of the stacked columns represents the running time associated with



(a) 300 MB (b) 600 MB (c) 1200 MB

Fig. 8. NET vs. TPT

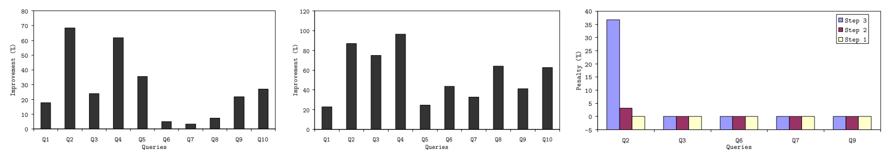


(a) 300 MB (b) 600 MB (c) 1200 MB

Fig. 9. NET vs EO_PRT

the corresponding materialization. For example, Q1 has two formulation steps and, therefore, two sections in the corresponding stacked column. There are two main observations. First, the increase in the running times as the data set size increases is less than linear. Therefore, the cost associated with materialization replacement is scalable. Second, the replacement cost for disjunctions is less than that for conjunctions. This is reflected in the results for queries involving disjunction (Q3, Q4, Q7 and Q8) as opposed to queries involving conjunction (Q2, Q6, Q9 and Q10). This is expected as the materialization selection algorithm selects materializations with disjunctions (Heuristic 1). As a result, evaluating conjunctions would involve an additional step.

NET vs TPT: This experiment is required to test the viability of prefetching. Figure 8 shows the results for this experiment. There are three main observations. The first is that the difference is not significant indicating that prefetching is a viable option. The second observation is that the conjunctive queries show a smaller difference than disjunctive ones. This is because conjunctive queries are evaluated from the corresponding disjunction based on materialization selection/replacement algorithms. This means that conjunctive queries will have a more significant prefetching overhead. This observation can be extended to queries that proceed from less selective partial queries to more selective final queries during formulation. The final observation is that for some of the queries (e.g., Q2, Q4, Q10), interestingly, the sum of the prefetching operations is less than the actual query execution time. This difference increases with data set size. This can be explained as follows. The search phase during query optimization typically treats the estimated cost model parameter values as though they were completely precise and accurate, rather than the coarse estimates that they actually are. Consequently, the relational query optimizer may fail to produce query plans that are more robust to estimation errors especially for complex queries. For Q2, Q4, and Q10, individual prefetching



(a) 300 MB (b) 1200 MB (c) 300MB - 3 form. steps

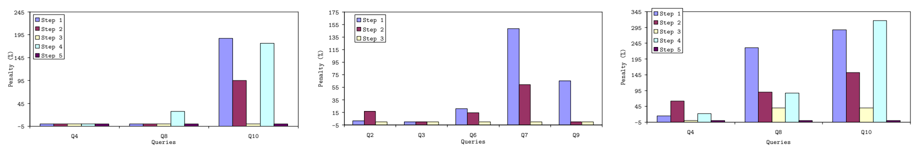
Fig. 10. EC_PRT vs NET and EC_PRT vs EO_PRT(1)

queries are relatively simpler compared to a single normal query. Hence, we observe such response time.

NET vs EO_PRT: The next experiment compares the *NET* with the *error-oblivious* perceived response time. This comparison is done as a percentage of improvement over normal execution. It is measured as $improvement$ = (1 - $\frac{EO\_PRT}{NET}$)$\times 100$. Figure 9 show the results for the three 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 16-89% for the 600MB data set and 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. Q2 seems to go against this observation. As mentioned earlier, this is due to the wide gap in the optimality of the query plans generated in the two approaches.

NET vs EC_PRT: In this experiment we evaluate the effect of QFE on perceived response time over normal execution time. 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[2]. The results are presented in Figures 10(a) and 10(b). We only take the smallest and the largest data sets (300MB and 1200MB) for this experiment. 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. Hence, QFEs do not significantly affect the performance improvement achieved by GUI-driven prefetching.

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 invoked just before clicking "Run") and vary error realization distance. Figures 10(c) and 11 show the results for the 300MB and 1200MB data sets. Figure 10(c) shows the results for queries that have three formulation steps (two predicates and a conjunction/disjunction) other than clicking on "Run" and Figure 11(a)



(a) 300MB - 5 form. steps (b) 1200MB - 3 form. steps (c) 1200MB - 5 form. steps

Fig. 11. EC_PRT vs EO_PRT (2)

shows the results for queries with five formulation steps. The three values shown for each query in Figure 10(c) measure the penalty when the error was committed at the first step, the second step and the third step respectively (variation of ϵ). The $penalty$ axis starts at -5 to allow the display of cases where $penalty = 0$.

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 Q2 for the 1200MB data set shows an increase as the evaluation of the second predicate is more expensive than the first. Similar phenomenon is observed for query Q4. Second, the impact of QFE increases with data set size. The 1200MB data set shows a maximum increase of 316%. The 300MB data set shows a maximum increase of 187%. The impact of QFE is felt on only four queries for the 300MB data set whereas all queries are effected for the 1200MB data set. This can be attributed to the higher cost of reevaluating materializations for the larger data set.

6 Related Work

GUI-latency driven optimization: Closest to our work is the effort by Polyzotis et al. [7] in *speculative* query processing. The method described is for relational data and incorporates speculation where the final query (or sub-queries that will be present in the final query) is predicted based on the user's usage profile. Machine learning techniques are applied on past user actions and a user-behavior model is formulated. In comparison, our approach employs deterministic prefetching without speculating on the final form of the query. This could result in a less than maximum gain in certain cases but there are no penalties. Speculation can lead to execution time penalties when the prediction is incorrect. In our case, this problem does not arise. Furthermore, we do not need to keep track of user's usage profile, but still can achieve comparable query performance improvement.

Prefetching and Caching: To the best of our knowledge, we have not found any published work related to prefetching techniques for XML data. Closest to the prefetching approach is caching, which although investigated extensively in relational database systems, is a relatively new area of research for XML data. However, XML caching techniques mentioned in [9] operate on the final query and do not take into account the individual steps in query formulation. In our approach, partial queries are materialized at each formulation step by utilizing the latency offered by GUI-driven query formulation. This presents a significant advantage over caching as every query benefits from prefetching unlike caching - where only those queries whose results have been cached improve in performance.

7 Conclusions and Future Work

The main contribution of this paper is to show that the latency offered by visual query formulation can be utilized to prefetch partial results so that the final query can be answered in a shorter time. We show that prefetching is viable as the combined time taken by all the prefetching operations is not significantly more than normal query execution time. In fact, for some queries the total time taken by all prefetching operations is less than the normal execution time due to a better query plan generated by the relational query optimizer. Our experiments also show that prefetching improves the perceived query response time by 7-96% with a greater improvement for larger data sets. In addition, query formulation errors have no significant influence on the perceived response time compared to the normal execution time. GUI-driven prefetching is potentially of value in XML query processing context where one would like to use a user-friendly GUI to formulate queries. Future directions of research include extension of our prefetching technique to more advanced XQueries, more sophisticated I/O cost estimation technique, and explore benefits of prefetching in a multiuser environment.

References

1. E. Augurusa, D. Braga, A. Campi, S. Ceri. Design and Implementation of a Graphical Interface to XQuery. In *ACM SAC*, 2003.
2. S. S. Bhowmick and S. Prakash. Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching in a Single-User Enviroment. *Technical Report*, CAIS-03-2005, School of Computer Engg, NTU, 2005 (Available at `http://www.ntu.edu.sg/home/assourav/papers/cais-03-2005-TR.pdf`).
3. S. S. Bhowmick and S. Prakash. Every Click You Make, I Will be Fetching It: Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching. In *ICDE*, 2006 (Poster paper).
4. 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.
5. G. Graefe (ed.). Special Issue on Query Processing in Commercial Database Management Systems. *IEEE Data Engineering*, 16:4, 1993.
6. L. Luo and B. E. John. Predicting Task Execution Time on Handheld Devices Using the Keystroke-Level Model. *In ACM CHI*, 2005.
7. N. Polyzotis and Y. Ioannidis. Speculative Query Processing. In *CIDR*, 2003.
8. S. Prakash, S. S. Bhowmick, S. K. Madria. Efficient Recursive XML Query Processing Using Relational Databases. *In DKE)*, 58(3), 2006.
9. L.-H. Yang, M.-Li. Lee, and W. Hsu. Efficient Mining of XML Query Patterns for Caching. In *VLDB*, 2003.