Statistical Learning Techniques for Costing XML Queries

Ning Zhang[1] Peter J. Haas[2] Vanja Josifovski[2] Guy M. Lohman[2] Chun Zhang[2]

[1] University of Waterloo
200 University Ave. W., Waterloo, ON, Canada
nzhang@uwaterloo.ca

[2] IBM Almaden Research Center
650 Harry Road, San Jose, CA, USA
phaas,vanja,lohman,chunz@us.ibm.com

Abstract

Developing cost models for query optimization is significantly harder for XML queries than for traditional relational queries. The reason is that XML query operators are much more complex than relational operators such as table scans and joins. In this paper, we propose a new approach, called COMET, to modeling the cost of XML operators; to our knowledge, COMET is the first method ever proposed for addressing the XML query costing problem. As in relational cost estimation, COMET exploits a set of system catalog statistics that summarizes the XML data; the set of "simple path" statistics that we propose is new, and is well suited to the XML setting. Unlike the traditional approach, COMET uses a new statistical learning technique called "transform regression" instead of detailed analytical models to predict the overall cost. Besides rendering the cost estimation problem tractable for XML queries, COMET has the further advantage of enabling the query optimizer to be self-tuning, automatically adapting to changes over time in the query workload and in the system environment. We demonstrate COMET's feasibility by developing a cost model for the recently proposed XNAV navigational operator. Empirical studies with synthetic, benchmark, and real-world data sets show that COMET can quickly obtain accurate cost estimates for a variety of XML queries and data sets.

1 Introduction

Management of XML data, especially the processing of XPath queries [5], has been the focus of considerable research and development activity over the past few years. A wide variety of join-based, navigational, and hybrid XPath processing techniques are now available; see, for example, [3, 4, 11, 25]. Each of these techniques can exploit structural and/or value-based indexes. An XML query optimizer can therefore choose among a large number of alternative plans for processing a specified XPath expression. As in the traditional relational database setting, the optimizer needs accurate cost estimates for the XML operators in order to choose a good plan.

Unfortunately, developing cost models of XML query processing is much harder than developing cost models of relational query processing. Relational query plans can be decomposed into a sequence of relatively simple atomic operations such as table scans, nested-loop joins, and so forth. The data access patterns for these relational operators can often be predicted and modeled in a fairly straightforward way. Complex XML query operators such as TURBOXPATH [14] and holistic twig join [7], on the other hand, do not lend themselves to such a decomposition. The data access patterns tend to be markedly non-sequential and therefore quite difficult to model. For these reasons, the traditional approach [21] of developing detailed analytic cost models based on a painstaking analysis of the source code often proves extremely difficult.

In this paper, we propose a statistical learning approach called COMET (COst Modeling Evolution by Training) for cost modeling of complex XML operators. Previous research on cost-based XML query optimization has centered primarily on cardinality estimation; see, e.g., [1, 9, 18, 23]. To our knowledge, COMET is the first method ever proposed for addressing the costing problem.

Our current work is oriented toward XML repositories consisting of a large corpus of relatively small XML documents, e.g., as in a large collection of relatively small customer purchase orders. We believe that such repositories will be common in integrated business-data environments. In this setting, the problems encountered when modeling I/O costs are relatively similar to those encountered in the relational setting: assessing the effects of caching, comparing random versus sequential disk accesses, and so forth. On the other hand, accurate modeling of CPU costs for XML operators is an especially challenging problem relative

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

to the traditional relational setting, due to the complexity of XML navigation. Moreover, experiments with DB2/XML have indicated that CPU costs can be a significant fraction (30% and higher) of the total processing cost. Therefore our initial focus is on CPU cost models. To demonstrate the feasibility of our approach, we develop a CPU cost model for the XNav operator, an adaptation of TurboXPath. Our ideas, insights, and experiences are useful for other complex operators and queries, both XML and relational.

The Comet methodology is inspired by previous work in which statistical learning methods are used to develop cost models of complex user-defined functions (UDFs)—see [13, 15]—and of remote autonomous database systems in the multidatabase setting [19, 26]. The basic idea is to identify a set of query and data "features" that determine the operator cost. Using training data, Comet then automatically learns the functional relationship between the feature values and the cost—the resulting cost function is then applied at optimization time to estimate the cost of XNav for incoming production queries.

In the setting of UDFs, the features are often fairly obvious, e.g., the values of the arguments to the UDF, or perhaps some simple transformations of these values. In the multidatabase setting, determining the features becomes more complicated: for example, Zhu and Larson [26] identify numerically-valued features that determine the cost of executing relational query plans. These authors also group queries by "type", in effect defining an additional categorically-valued feature. In the XML setting, feature identification becomes even more complex. The features that have the greatest impact on the cost tend to be "posterior" features—such as the number of data objects returned and the number of candidate results inserted in the in-memory buffer—that depend on the data and cannot be observed until after the operator has finished executing. This situation is analogous to what happens in relational costing and, as in the relational setting, Comet estimates the values of posterior features using a set of catalog statistics that summarize the data characteristics. We propose a novel set of such "simple path" (SP) statistics that are well suited to cost modeling for complex navigational XML operators, along with corresponding feature-estimation procedures for XNav.

The Comet approach is therefore a hybrid of traditional relational cost modeling and a statistical learning approach: some analytical modeling is still required, but each analytical modeling task is relatively straightforward, because the most complicated aspects of operator behavior are modeled statistically. In this manner we can take advantage of the relative simplicity and adaptability of statistical learning methods while still exploiting the detailed information available in the system catalog. We note that the query features

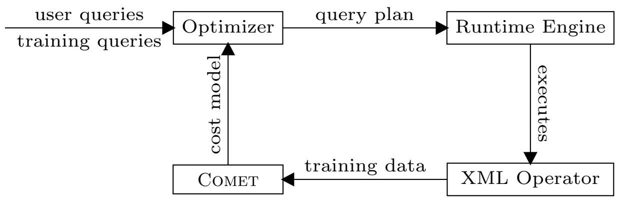


Figure 1: Use of Comet in self-tuning systems

can be defined in a relatively rough manner, as long as "enough" features are used so that no important cost-determining factors are ignored; as discussed in Section 3.3, Comet's statistical learning methodology automatically handles redundancy in the features.

Any statistical learning method that is used in Comet must satisfy several key properties. It must be fully automated and not require human statistical expertise, it must be highly efficient, it must seamlessly handle both numerical and categorical features, and it must be able to deal with the discontinuities and nonlinearities inherent in cost functions. One contribution of this paper is our proposal to use the new *transform regression* (TR) method recently introduced by Pednault [17]. This method is one of the very few that satisfy all of the above criteria.

A key advantage of Comet's statistical learning methodology is that an XML query optimizer, through a process of query feedback, can exploit Comet in order to be self-tuning. That is, the system can automatically adapt to changes over time in the query workload and in the system environment. The idea is illustrated in Figure 1: user queries are fed to the optimizer, each of which generates a query plan. During plan execution, the runtime engine executes the operator of interest and a runtime monitor records the feature values and subsequent execution costs. The Comet learner then uses the feedback data to update the cost model. Our approach can leverage existing self-tuning technologies such as those in [2, 10, 15, 19, 22]. Observe that the model can initially be built using the feedback loop described above, but with training queries instead of user queries. The training phase ends once a satisfactory initial cost model is generated, where standard techniques such as n-fold cross-validation (see, e.g., [12, Sec. 7.10]) can be used to assess model quality.

The rest of the paper is organized as follows. In Section 2, we provide some background information on XML query optimization and the XNav operator. In Section 3, we describe the application of Comet to cost modeling of XNav. In Section 4, we present an empirical assessment of Comet's accuracy and execution cost. In Section 5 we summarize our findings and give directions for future work.

2 Background

We first motivate the XML query optimization problem and then give an overview of the XNav operator.

2.1 XML Processing and Query Optimization

We use a running example both to motivate the query optimization problem and to make our COMET description concrete. The example is excerpted from the XQuery use cases document [8] with minor modifications.

Example 1 *Consider the following FLWOR expression, which finds the titles of all books having at least one author named "Stevens" and published after 1991.*

```
<bib>
  {
    for $b in doc("bib.xml")/bib/book
    where $b/authors//last = "Stevens" and
          $b/@year > 1991
    return
        <book>{ $b/title }</book>
  }
</bib>
```

The three path expressions in the for- and where-clauses constitute the matching part, and the return-clause corresponds to the construction part. In order to answer the matching part, an XML query processing engine may generate at least three query plans:

1. Navigate the `bib.xml` document down to find all `book` elements under the root element `bib` and, for each such `book` element, evaluate the two predicates by navigating down to the attribute `year` and element `last` under `authors`.
2. Find the elements with the values "`Stevens`" or "`1991`" through value-based indexes, then navigate up to find the parent/ancestor element `book`, verify other structural relationships, and finally check the remaining predicate.
3. Find, using a twig index, all tree structures in which `last` is a descendant of `authors`, `book` is a child of `bib`, and `@year` is an attribute of `book`. Then for each `book`, check the two value predicates.

Any one of these plans can be the best plan, depending on the circumstances. To compute the cost of a plan, the optimizer estimates the cost of each operator in the plan (e.g., index access operator, navigation operator, join) and then combines their costs using an appropriate formula. For example, let p_1, p_2, and p_3 denote the path expressions `doc("bib.xml")/bib/book`, `authors//last[.="Stevens"]`, and `@year[.>1991]`, respectively. The cost of the first plan above may be modeled by the following formula:

$$cost_{nv}(p_1) + |p_1| \times cost_{nv}(p_2) + |p_1[p_2]| \times cost_{nv}(p_3),$$

where $cost_{nv}(p)$ denotes the estimated cost of evaluating the path expression p by the navigational approach, and $|p|$ denotes the cardinality of path expression p. Therefore the costing of path-expression evaluation is crucial to the costing of alternative query plans, and thus to choosing the best plan.

Algorithm 1 XNAV Pattern Matching

```
XNAV(P : ParseTree, X : XMLDocument)
 1  match_buf ← {root of P};
 2  while not end-of-document
 3    do x ← next event from traversal of X;
 4       if x is a startElement event for XML node y
 5           if y matches some r ∈ match_buf
 6               set r's status to TRUE;
 7               if r is a non-leaf
 8                   set r children's status to FALSE;
 9                   add r's children to match_buf;
10               if r is an output node
11                   add r to out_buf;
12               if r is a predicate tree node
13                   add r to pred_buf;
14           elseif no r ∈ match_buf is connected by //-axis
15               skip through X to y's following sibling;
16       elseif x is endElement event for XML node y
17           if y matches r ∈ match_buf
18               remove r from match_buf;
19           if y is in pred_buf
20               set y's status to the result of evaluating
                    the predicate;
21           if the status of y or one of its children is FALSE
22               remove y from out_buf;
```

2.2 The XNAV Operator

XNAV is a slight adaptation of the stream-based TURBOXPATH algorithm described in [14] to pre-parsed XML stored as paged trees. As with TURBOXPATH, the XNAV algorithm processes the path query using a single-pass, pre-order traversal of the document tree. Unlike TURBOXPATH, which copies the content of the stream, XNAV manipulates XML tree references, and returns references to all tree nodes that satisfy a specified input XPath expression. Another difference between TURBOXPATH and XNAV is that, when traversing the XML document tree, XNAV skips those portions of the document that are not relevant to the query evaluation. This behavior makes the cost modeling of XNAV highly challenging. A detailed description of the XNAV algorithm is beyond the scope of this paper; we give a highly simplified sketch that suffices to illustrate our costing approach.

XNAV behaves approximately as pictured in Algorithm 1. Given a parse tree representation of a path expression and an XML document, XNAV matches the incoming XML elements with the parse tree while traversing the XML data in document order. An XML element *matches* a parse-tree node if (1) the element name matches the node label, (2) the element value satisfies the value constraints if the node is also a predicate tree node, and (3) the element satisfies structural relationships with other previously matched XML elements as specified by the parse tree.

An example of the parse tree is shown in Figure 2. It represents the path expression `/bib/book[authors//last="Stevens"][@year>1991]/title`. In this parse tree, each unshaded node corresponds to a "NodeTest" in the path expression, except that the node labeled with "`r`" is a special node representing the starting

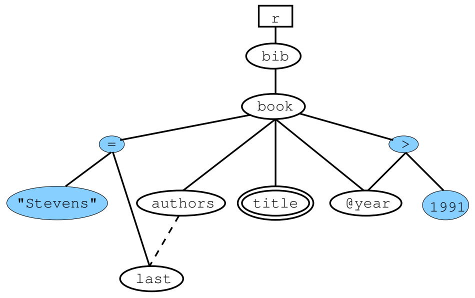


Figure 2: A parse tree

node for the evaluation (which can be the document root or any other internal node of the document tree). The doubly-circled node is the "output node". Each NodeTest with a value constraint (i.e., a predicate) has an associated predicate tree. These are shaded in Figure 2. Edges between parse tree nodes represent structural relationships (i.e., axes). Solid and dashed lines represent child ("/") and descendant ("//") axes, respectively. Each predicate is attached to an "anchor node" (`book` in the example) that represents the XPath step at which the predicate appears.

For brevity and simplicity, we consider only path expressions that contain / and //-axes, wildcards, branching, and value-predicates. COMET can be extended to handle position-based predicates and variable references (by incorporating more features into the learning model).

3 The COMET Methodology

COMET comprises the following the basic approach: (1) Identify algorithm, query, and data features that are important determinants of the cost—these features are often unknown *a priori*; (2) Estimate feature values using statistics and simple analytical formulas; (3) Learn the functional relationship between feature values and costs using a statistical or machine learning algorithm; (4) Apply the learned cost model for optimization, and adapt it via self-tuning procedures.

The COMET approach is general enough to apply to any operator. In this section, we apply it to a specific task, that of modeling the CPU cost of the XNAV operator. We first describe the features that determine the cost of executing XNAV, and provide a means of estimating the feature values using a set of "SP statistics." We then describe the transform regression algorithm used to learn the functional relationship between the feature values and the cost. Finally, we briefly discuss some approaches to dynamic maintenance of the learning model as the environment changes.

3.1 Feature Identification

We determined the pertinent features of XNAV both by analyzing the algorithm and by experience and experimentation. We believe that it is possible to identify the features automatically, and this is part of our future work.

As can be seen from Algorithm 1, XNAV employs three kinds of buffers: output buffers, predicate buffers, and matching buffers. The more elements inserted into the buffers, the more work performed by the algorithm, and thus the higher the cost. We therefore chose, as three of our query features, the total number of elements inserted into the output, predicate, and matching buffers, respectively, during query execution. We denote the corresponding feature variables as **#out_bufs**, **#preds_bufs**, and **#match_bufs**.

In addition to the number of buffer insertions, XNAV's CPU cost is also influenced by the total number of nodes in the XML document that the algorithm "visits" (i.e., does not skip as in line 15 of Algorithm 1). We therefore included this number as a feature, denoted as **#visits**. Another important feature that we identified is **#results**, the number of XML elements returned by XNAV. This feature affects the CPU cost in a number of ways. For example, a cost is incurred whenever an entry in the output buffer is removed due to invalid predicates (line 22); the number of removed entries is roughly equal to **#out_bufs** – **#results**.

Whenever XNAV generates a page request, a CPU cost is incurred as the page cache is searched. (An I/O cost may also be incurred if the page is not in the cache.) Thus we included the number of page requests as a feature, denoted as **#p_requests**. Note that **#p_requests** cannot be subsumed by **#visits**, because different data layouts may result in different page-access patterns even when the number of visited nodes is held constant.

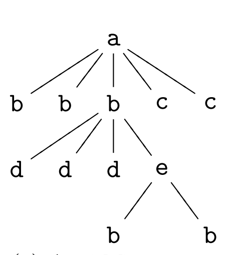
A final key component of the CPU cost is the "post-processing" cost incurred in lines 17 to 22. This cost can be captured by the feature **#post_process**, defined as the total number of endElement events that trigger execution of one or more of lines 18, 20, and 22.

3.2 Statistics and Feature Estimation

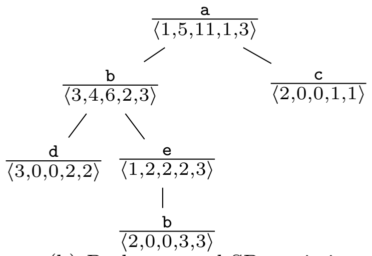
Observe that each of the features that we have identified is a *posterior* feature in that the feature value can only be determined after the operator is executed. COMET needs, however, to estimate these features at optimization time, prior to operator execution. As in the relational setting, COMET computes estimates of the posterior feature values using a set of catalog statistics that summarize important data characteristics. We describe the novel SP statistics that COMET uses and the procedures for estimating the feature values.

3.2.1 Simple-Path Statistics

Before describing our new SP statistics, we introduce some terminology. An XML document can be rep-



(a) An XML tree T



(b) Path tree and SP statistics

Figure 3: An XML tree, its path tree and SP statistics

resented as a tree T, where the nodes correspond to elements and the arcs correspond to 1-step child relationships. Given any path expression p and an XML tree T, the *cardinality of p under T*, denoted as $|p(T)|$ (or simply $|p|$ when T is clear from the context), is the number of result nodes that are returned when p is evaluated on the XML document represented by T. A *simple path expression* is a linear chain of (non-wildcard) NodeTests that are connected by child-axes. For example, /bib/book/@year is a simple path expression, whereas //book/title and /*/book[@year]/publisher are not. A *simple path p in T* is a simple path expression such that $|p(T)| > 0$. Denote by $\mathcal{P}(T)$ the set of all simple paths in T.

For each simple path $p \in \mathcal{P}(T)$, COMET maintains the following statistics:

1. **cardinality**: the cardinality of p under T, that is, $|p|$.
2. **children**: number of p's children under T, that is, $|p/*|$.
3. **descendants**: number of p's descendants under T, that is, $|p//*|$.
4. **page cardinality**: number of pages requested in order to answer the path query p, denoted as $\|p\|$.
5. **page descendants**: number of pages requested in order to answer the path query $p//*$, denoted as $\|p//*\|$.

Denote by $s_p = \langle s_p(1), \ldots, s_p(5)\rangle$ the forgoing statistics, enumerated in the order given above. The SP statistics for an XML document represented by a tree T are then defined as $\mathcal{S}(T) = \{\, (p, s_p) : p \in \mathcal{P}(T)\,\}$.

SP statistics can be stored in a *path tree* [1], which captures all possible simple paths in the XML tree. For example, Figure 3 shows an XML tree and the corresponding path tree with SP statistics. Note that there is a one-to-one relationship between the nodes in the path tree T_p and the simple paths in the XML tree T. Alternatively, we can store the SP statistics in a more sophisticated data structure such as TreeSketch [18] or simply in a table. Detailed comparisons of storage space and retrieval/update efficiency are beyond our current scope.

3.2.2 Feature Estimation

Algorithm 2 Estimation Functions

```
VISITS(proot : ParseTreeNode)
1  $v \leftarrow 0$;
2  for each non-leaf node $n$ in depth-first order
3    do $p \leftarrow$ the path from $proot$ to $n$;
4       if $p$ is a simple path (i.e., no //-axis)
5          if one of $n$'s children is connected by //-axis
6             $v \leftarrow v + |p//*|$;
7             skip $n$'s descendants in the traversal;
8          else $v \leftarrow v + |p/*|$;
9  return $v$;

RESULTS(proot : ParseTreeNode)
1  $t \leftarrow$ the trunk in $proot$
2  return $|t|$;

PAGES(proot : ParseTreeNode)
1  $p \leftarrow 0$; $R \leftarrow \emptyset$;
2  $L \leftarrow$ list of all root-to-leaf paths in depth-first order;
3  for every pair of consecutive paths $l_i, l_{i+1} \in L$
4    do add common subpath between $l_i$ and $l_{i+1}$ to $R$;
5  for each $l \in L$
6    do $p \leftarrow p + \|l\|$;
7  for each $r \in R$
8    do $p \leftarrow p - \|r\|$;
9  return $p$;

BUF-INSERTS(p : LinearPath)
1  if $p$ is not recursive
2       return $|p|$;
3  else $m \leftarrow 0$;
4       for each recursive node $u$ such that $p = l//u$
5         do $m \leftarrow m + \sum_{i=1}^{d} |l\{//u\}^{*i}|$;
6       return $m$;

MATCH-BUFFERS(proot : ParseTreeNode)
1  $m \leftarrow 0$;
2  for each non-leaf node $n$
3    do $p \leftarrow$ the path from $proot$ to $n$;
4       $m \leftarrow m +$ BUF-INSERTS($p$) $\times$ FANOUT($n$);
5  return $m$;

PRED-BUFFERS(proot : ParseTreeNode)
1  $r \leftarrow 0$;
2  for each predicate-tree node $n$
3    do $p \leftarrow$ the path from $proot$ to $n$;
4       $r \leftarrow r +$ BUF-INSERTS($p$);
5  return $r$;

OUT-BUFFERS(proot : ParseTreeNode)
1  $t \leftarrow$ the trunk in $proot$
2  return BUF-INSERTS($t$);

POST-PROCESS(proot : ParseTreeNode)
1  $L \leftarrow$ all possible paths in the parse tree rooted at $proot$;
2  $n \leftarrow 0$;
3  for each $l \in L$
4    do $n \leftarrow n +$ BUF-INSERTS($l$);
5  return $n$;
```

Algorithm 2 lists the functions that estimate the feature values from SP statistics. These estimation functions allow path expressions to include arbitrary number of //-axes, wildcards ("*"), branches, and value-predicates. The parameter *proot* of the functions is the special root node in the parse tree (labeled as "r" in Figure 2). In the following, we outline the rationale behind each function and illustrate using the example shown in Figure 2.

Visits: The function VISITS in Algorithm 2 is straightforward. At each step of a path expression, if the current NodeTest `u` is followed by /, then a traversal of the children of `u` ensues. If `u` is followed by //, then a traversal of the subtree rooted at `u` ensues. E.g., for the parse tree in Figure 2,

$$\#\textbf{visits} = 1 + |\texttt{/*}| + |\texttt{/bib/*}| + |\texttt{/bib/book/*}| + |\texttt{/bib/book/authors//*}|,$$

where the first term in the sum corresponds to the document root, matched with the node `r` in the parse tree.

Results: We estimate **#results** as the cardinality of the "trunk," i.e., the simple path obtained from the original path expression by removing all branches. This estimate is cruder than the more expensive methods proposed in the literature, e.g., [1, 18]. Our experiments indicate, however, that a rough (over)estimate suffices for our purposes, mainly due to COMET's bias compensation (Section 3.3.1; also see Section 4.3 for empirical verification). For the parse tree in Figure 2, the estimate is simply

$$\#\textbf{results} \approx |\texttt{/bib/book/title}|.$$

Page Requests: The function PAGES computes the number of pages requested when evaluating a particular path expression. We make the following buffering assumption: when navigating the XML tree in a depth-first traversal, a page read when visiting node x is kept in the buffer pool until all x's descendants are visited.

Under this assumption, observe that, e.g.,:

1. $\|\texttt{/a[b][c]}\| = \|\texttt{/a/b}\| = \|\texttt{/a/c}\| = \|\texttt{/a/*}\|$.
2. $\|\texttt{/a[b/c][d/e]}\| \approx \|\texttt{/a/b/c}\| + \|\texttt{/a/d/e}\| - \|\texttt{/a/*}\|$.

The above observation is generalized to path expressions with more than two branches in function PAGES of Algorithm 2. For the parse tree in Figure 2, the feature estimate is:

$$\begin{aligned}\#\textbf{p\_requests} &\approx \|\texttt{/bib/book/authors//*}\| \\ &\quad + \|\texttt{/bib/book/title}\| + \|\texttt{/bib/book/@year}\| \\ &\quad - \|\texttt{/bib/book/*}\| - \|\texttt{/bib/book/*}\| \\ &= \|\texttt{/bib/book/authors//*}\|\end{aligned}$$

Buffer insertions for recursive queries: Before we explain how the values of **#out_bufs**, **#preds_bufs**, and **#match_bufs** are estimated, we first describe COMET's method for calculating the number of buffer insertions for a recursive query. Buffer insertions occur whenever an incoming XML event matches one or more nodes in the matching buffer (line 5 in Algorithm 1). An XML event can create two or more matching-buffer entries for a single parse tree node when two parse-tree nodes connected by one or more //-axes have the same name.

In this case, the number of buffer insertions induced by a recursive parse tree node `u` can be estimated as follows: first, all nodes returned by l//`u` are inserted into the buffer, where l is the prefix of the path from root to `u`. Next, all nodes returned by l//`u`//`u` are inserted, then all nodes returned by l//`u`//`u`//`u`, and so forth, until a path expression returns no results. The total number of nodes inserted can therefore be computed as $\sum_{i=1}^{d} |l\{\texttt{//u}\}^{*i}|$, where d is the depth of the XML tree and $\{//u\}^{*i}$ denotes the i-fold concatenation of the string "//`u`" with itself.

The function BUF-INSERTS in Algorithm 2 calculates the number of buffer insertions for a specified linear path expression that may or may not contain recursive nodes. If the path has no recursive nodes, the function simply returns the cardinality of the path. Otherwise, the function returns the sum of number of insertions for each recursive node. BUF-INSERTS is called by each of the last four functions in Algorithm 2.

Matching buffers: The feature **#match_bufs** is the total number of entries inserted into the matching buffer, which stores those candidate parse tree nodes that are expected to match with the incoming XML nodes. In Algorithm 1, whenever an incoming XML event matches with a parse tree node u, a matching-buffer entry is created for every child of u in the parse tree. Therefore, we estimate **#match_bufs** by summing FANOUT(u) over every non-leaf parse-tree node u, where FANOUT(u) denotes the number of u's children. For the parse tree in Figure 2, there are no recursive nodes, so that **#match_bufs** is estimated as:

$$\begin{aligned}\#\textbf{match\_bufs} &\approx |\texttt{/bib}| + 3 \times |\texttt{/bib/book}| \\ &\quad + |\texttt{/bib/book/authors}| + |\texttt{/bib/book/authors//last}|,\end{aligned}$$

where the factor 3 is the fanout of node `book` in the parse tree.

Predicate buffer and *output buffer:* The derivation of the function OUT-BUFFERS is similar to that of RESULTS, and the derivation of PRED-BUFFERS is straightforward.

Post-processing: According to Algorithm 1, post-processing is potentially triggered by each endElement event (line 16). If the closing XML node was not matched with any parse tree node, no actual processing is needed; otherwise, the buffers need to be maintained (lines 17 to 22). Thus the feature **#post_process** can be estimated by the total number of XML tree nodes that are matched with parse tree nodes. For the parse tree in Figure 2, **#post_process** is estimated as

$$\begin{aligned}\#\textbf{post\_process} &\approx 1 + |\texttt{/bib}| + |\texttt{/bib/book}| \\ &\quad + |\texttt{/bib/book/authors}| + |\texttt{/bib/book/authors//last}| \\ &\quad + |\texttt{/bib/book/title}| + |\texttt{/bib/book/@year}|,\end{aligned}$$

where the first term results from the matching of the root node.

3.3 Statistical Learning

We now discuss COMET's statistical learning component.

3.3.1 The General Learning Problem

Given a set of d features, the goal of the statistical learner is to determine a function f such that, to a good approximation,

$$\text{cost}(q) = f(v_1, v_2, \ldots, v_d) \tag{1}$$

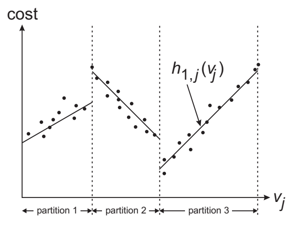
for each query q—here $v_1, v_2, \ldots, v_d$ are the d feature values associated with q. COMET uses a supervised learning approach: the training data consists of $n \geq 0$ points $x_1, \ldots, x_n$ with $x_i = (v_{1,i}, v_{2,i}, \ldots, v_{d,i}, c_i)$ for $1 \leq i \leq n$. Here $v_{j,i}$ is the value of the jth feature for the ith training query q_i and c_i is the observed cost for q_i. As discussed in the introduction, the learner is initialized using a starting set of training queries, which can be obtained from historical workloads or synthetically generated. Over time, the learner is periodically retrained using queries from the actual workload.

For each "posterior" feature, COMET actually uses estimates of the feature value—computed from catalog statistics as described in Section 3.2—when building the cost model. That is, the ith training point is of the form $\hat{x}_i = (\hat{v}_{1,i}, \hat{v}_{2,i}, \ldots, \hat{v}_{d,i}, c_i)$, where $\hat{v}_{j,i}$ is an estimate of $v_{j,i}$. An alternative approach uses the actual feature values for training the model. The advantage of our method is that it automatically compensates for systematic biases in the feature estimates, allowing COMET to use relatively simple feature-estimation formulas. This desirable feature is experimentally verified in Section 4.3.

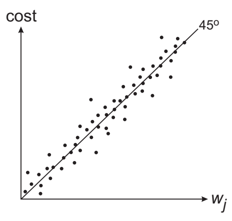
3.3.2 Transform Regression

For reasons discussed previously, we use the recently-proposed transform regression (TR) method [17] to fit the function f in (1). Because a published description of TR is not readily available, we expend some effort on outlining the basic ideas that underlie the algorithm; details of the statistical theory and implementation are beyond the current scope. TR incorporates a number of modeling techniques in order to combine the strengths of decision tree models—namely computational efficiency, nonparametric flexibility, and full automation—with the low estimation errors of a neural-network approach as in [6]. In our discussion, we suppress the fact that the feature values may actually be estimates, as discussed in Section 3.3.1.

The fundamental building block of the TR method is the Linear Regression Tree (LRT) [16]. TR uses LRTs having a single level, with one LRT for each feature. For the jth feature, the corresponding LRT splits the training set into mutually disjoint partitions based on the feature value. The points in a partition are projected to form reduced training points of the form $(v_{j,i}, c_i)$; these reduced training points are then used to fit a univariate linear regression model of cost as a function of v_j. Combining the functions from each partition leads to an overall piecewise-linear function $h_{1,j}(v_j)$ that predicts the cost as a function of the jth feature value. A typical function $h_{1,j}$ is displayed in Figure 4(a), along with the reduced training points. Standard classification-tree methodology is used to automatically determine the number of partitions and the splitting points.



(a) Cost vs v_j



(b) Cost vs w_j

Figure 4: Feature linearization

Observe that, for each feature j, the cost is approximately a linear function of the transformed feature $w_j = h_{1,j}(v_j)$; see, e.g., Figure 4(b). In this figure, which corresponds to the hypothetical scenario of Figure 4(a), we have plotted the pairs $(w_{j,i}, c_i)$, with $w_{j,i} = h_{1,j}(v_{j,i})$ being the value of the transformed jth feature for query q_i. In statistical-learning terminology, the transformation of v_j to w_j "linearizes" the jth feature with respect to cost. A key advantage of our methodology is the completely automated determination of this transformation. Because the cost is now linear with respect to each transformed feature, we can obtain an overall first-order cost model using multiple linear regression on the transformed training points $\{ (w_{1,i}, \ldots, w_{d,i}, c_i) : 1 \leq i \leq n \}$. The current implementation of the TR algorithm uses a greedy forward stepwise-regression algorithm. The resulting model is of the "generalized additive" form

$$g^{(1)}(v_1, v_2, \ldots, v_d) = a_0 + \sum_{j=1}^{d} a_j w_j = a_0 + \sum_{j=1}^{d} a_j h_{1,j}(v_j).$$

So our initial attempt at learning the true cost function f that appears in (1) yields the first-order model $f^{(1)} = g^{(1)}$. Note that, at this step and elsewhere, the stepwise-regression algorithm automatically deals with redundancy in the features (i.e., multicolinearity): features are added to the regression model one at a time, and if two features are highly correlated, then only one of the features is included in the model.

The main deficiency of the first-order model is that each feature is treated in isolation. If the true cost function involves interactions such as $v_1 v_2$, $v_1^2 v_2$, or $v_1^{v_2}$, then the first-order model will not properly account for these interactions and systematic prediction errors will result. One approach to this problem is to explicitly add interaction terms to the regression model, but it is extremely hard to automate the determination of precisely which terms to add. The TR algorithm uses an alternative approach based on "gradient boosting." After determining the first-order model, the TR algorithm computes the residual error for each test query: $r_i^{(1)} = c_i - f^{(1)}(v_{1,i}, v_{2,i}, \ldots, v_{d,i})$ for $1 \leq i \leq n$. TR then uses the methodology described above to develop a generalized additive model $g^{(2)}$ for predicting the residual error $r^{(1)}(q) = \mathrm{cost}(q) - f^{(1)}(v_1, v_2, \ldots, v_d)$. Then our second-order model is $f^{(2)} = g^{(1)} + g^{(2)}$. This process can be iterated m times to obtain a final mth-order model of the form $f^{(m)} = g^{(1)} + g^{(2)} + \cdots + g^{(m)}$. The TR algorithm uses standard cross-validation techniques to determine the number of iterations in a manner that avoids model overfitting.

The TR algorithm uses two additional techniques to improve the speed of convergence and capture non-linear feature interactions more accurately. The first trick is to use the output of previous iterations as regressor variables in the LRT nodes. That is, instead of performing simple linear regression analysis during the kth boosting iteration on pairs of the form $(v_{j,i}, r_i^{(k)})$ to predict the residual error as a function of the jth feature, TR performs a multiple linear regression on tuples of the form $(v_{j,i}, r_i^{(0)}, r_i^{(1)}, \ldots, r_i^{(k-1)}, r_i^{(k)})$; here $r_i^{(0)} = f^{(1)}(v_{1,i}, v_{2,i}, \ldots, v_{d,i})$ is the first-order approximation to the cost. This technique can be viewed as a form of successive orthogonalization that accelerates convergence; see [12, Sec. 3.3]. The second trick is to treat the outputs of previous boosting iterations as additional features in the current iteration. Thus the generalized additive model at the kth iteration is of the form

$$\begin{aligned} &g^{(k)}(v_1, \ldots, v_d, r^{(0)}, \ldots, r^{(k-1)}) \\ &\quad = a_0 + \sum_{j=1}^{d} a_j h_{k,j}(v_j) + \sum_{j=0}^{k-1} a_{d+j+1} h_{k,d+j+1}(r^{(j)}), \end{aligned}$$

where each function $h_{k,s}$ is obtained from the LRT for the jth feature using multivariate regression.[1]

We emphasize that the features $v_1, v_2, \ldots, v_d$ need not be numerically valued. For a categorical feature, the partitioning of the feature-value domain by the corresponding LRT has a general form and does not correspond to a sequential splitting as in Figure 4(a); standard classification-tree techniques are used to effect the partitioning. Also, a categorical feature is never used as a regressor at the LRT node—this means that the multivariate regression model at a node is sometimes degenerate, that is, equal to a fixed constant a_0. When all nodes are degenerate, the LRT reduces to a classical "regression tree" in the sense of [12, Sec. 9.2.2]).

[1]Strictly speaking, we should write $h_{k,j}(v_j, r^{(0)}, \ldots, r^{(k-1)})$ instead of $h_{k,j}(v_j)$ and similarly for $h_{k,d+j+1}(r^{(j)})$.

3.3.3 Updating the Model

COMET can potentially exploit a number of existing techniques for maintaining statistical models. The key issues for model maintenance include (1) when to update the model, (2) how to select appropriate training data, and (3) how to efficiently incorporate new training data into an existing model. We discuss these issues briefly below.

One very aggressive policy updates the model whenever the system executes a query. As discussed in [19], such an approach is likely to incur an unacceptable processing-time overhead. A more reasonable approach updates the model either at periodic intervals or when cost-estimation errors exceed a specified threshold (in analogy, for example, to [10]). Aboulnaga, et al. [2] describe an industrial-strength system architecture for scheduling statistics maintenance; many of these ideas can be adapted to the current setting.

There are many ways to choose the training set for updating a model. One possibility is to use all of the queries seen so far, but this approach can lead to extremely large storage requirements and sometimes a large CPU overhead. Rahal, et al. [19] suggest some alternatives, including using a "backing sample" of the queries seen so far. An approach that is more responsive to changes in the system environment [19] uses all of the queries that have arrived during a recent time window (or perhaps a sample of such queries). It is also possible to maintain a sample of queries that contains some older queries, but is biased towards more recent queries.

Updating a statistical model involves either recomputing the model from scratch, using the current set of training data, or using an incremental updating method. Examples of the latter approach can be found in [15, 19], where the statistical model is a classical multiple linear regression model and incremental formulas are available for updating the regression coefficients. There is currently no method for incrementally updating a TR model, although research on this topic is underway. Fortunately, our experiments indicate that even recomputing a TR model from scratch is an extremely rapid and efficient operation; our experiments indicate that a TR model can be constructed from several thousand training points in a fraction of a second.

4 Performance Study

In this section, we demonstrate the COMET's accuracy using a variety of XML datasets and queries. We also study COMET's sensitivity to errors in the SP statistics. Finally, we examine COMET's efficiency and the size of the training set that it requires.

data sets	total size	# of nodes	avg. depth	avg. fan-out	# simple paths
`rf.xml`	3 MB	108,832	4.94	11.94	20
`rd.xml`	865 KB	12,949	15.16	2.43	2,354
`nf.xml`	6.7 MB	200,293	5.25	24.11	109
`nd.xml`	188 KB	4,096	7.5	2.0	4,096
`TPC-H`	34 MB	1,106,689	3.86	14.79	27
XMark	11 MB	167,865	5.56	3.66	514
NASA	25 MB	474,427	5	2.81	95
XBench (DC/MD)	43 MB	933,480	3.17	6.0	26
XBench (TC/MD)	121 MB	621,934	5.16	3.73	32

Table 1: Characteristics of the experimental data sets

4.1 Experimental Procedure

We performed experiments on three different platforms running Windows 2000 and XP, configured with different CPU speeds (1GHz, 500MHz, 2GHz) and memory sizes (512MB, 384MB, 1GB). Our results are consistent across different hardware configurations.

We used synthetically generated data sets as well as data sets from both well-known benchmarks and a real-world application. Although our motivating scenario is XML processing on large corpus of relatively small documents, we also experimented on some data sets containing large XML documents to see how COMET performs. The results are promising.

For each data set, we generated three types of queries: simple paths (SP), branching paths (BP), and complex paths (CP). The latter type of query contains at least one instance of //, *, or a value predicate. We generated all possible SP queries along with 1000 random BP and 1000 random CP queries. These randomly generated queries are non-trivial. A typical CP query looks like `/a[*][*[*[b4]]]/b1[//d2[./text()<70.449]]/c3`. For each data set, we computed SP statistics. Then, for each query on the data set, we computed the feature-value estimates and measured the actual CPU cost. The estimated feature values together with actual CPU costs constituted the training data set. To measure the CPU time for a given query accurately, we ran the query several times to warm up the cache, and then used the elapsed time for the final run as our CPU measurement.

We applied 5-fold cross-validation to the training data in order to gauge COMET's accuracy. The cross-validation procedure was as follows: we first randomly divided the data set into five equally sized subsets. Each subset served as a testing set and the union of the remaining four subsets served as a training set. This yielded five training-testing pairs. For each such pair, COMET learned the model from the training set and applied it to the testing set. We then combined the (predicted cost, actual cost) data points from all five training-testing pairs to assess COMET's accuracy.

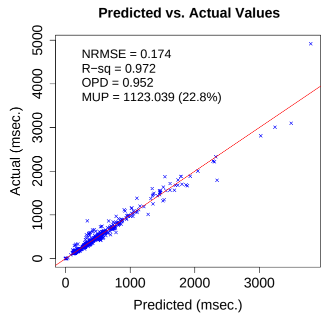
The above procedure was carried out in the same way for synthetic and benchmark workloads, except that for each benchmark data set, we not only used synthetic queries, but also used the path expressions in the benchmark queries for testing. More specifically, we added (predicted cost, actual cost) data points obtained from those path expressions into each of the testing sets during the 5-fold cross-validation.

4.2 Accuracy of COMET

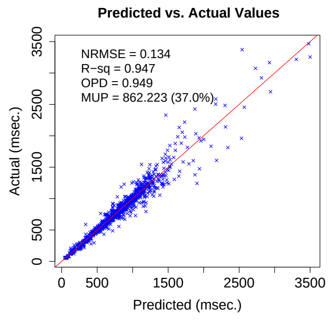
We use several metrics to measure COMET's accuracy. Each metric is defined for a set of test queries $Q = \{ q_1, q_2, \ldots, q_n \}$, and for query q_k, we denote by c_k and $\hat{c}_k$ the actual and predicted XNAV CPU costs.

- **Normalized Root-Mean-Squared Error (NRMSE)**: This metric is a normalized measure of the average prediction error, and is defined as
$$\text{NRMSE} = \frac{1}{\bar{c}}\left(\frac{1}{n}\sum_{i=1}^{n}(c_i - \hat{c}_i)^2\right)^{1/2},$$
where $\bar{c}$ is the average of $c_1, c_2, \ldots, c_n$.
- **Coefficient of Determination (R-sq)**: This metric, which measures the proportion of variability in the cost predicted by COMET, is given by
$$\text{R-sq} = \frac{\left(\sum_{i=1}^{n}(c_i - \bar{c})(\hat{c}_i - \bar{\hat{c}})\right)^2}{\left(\sum_{i=1}^{n}(c_i - \bar{c})^2\right)\left(\sum_{i=1}^{n}(\hat{c}_i - \bar{\hat{c}})^2\right)},$$
where $\bar{\hat{c}}$ is the average of $\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_n$.
- **Order-Preserving Degree (OPD)**: This metric is tailored to query optimization and measures how well COMET preserves the ordering of query costs. A pair of queries (q_i, q_j) is *order preserving* provided that $c_i\ (<,=,>)\ c_j$ if and only if $\hat{c}_i\ (<,=,>)\ \hat{c}_j$. Given a set of queries $Q = \{q_1, q_2, \ldots, q_n\}$, we then set $\text{OPD}(Q) = |\text{OPP}|/n^2$, where OPP is the set of all order preserving pairs.
- **Maximum Under-Prediction Error (MUP)**: This metric, defined as $\text{MUP} = \max_{1 \le i \le n}(c_i - \hat{c}_i)$, measures the worst-case underprediction error. This metric is frequently used by commercial optimizers that strive for good average behavior by avoiding costly query plans. Over-costing good plans is less of a concern in practice.

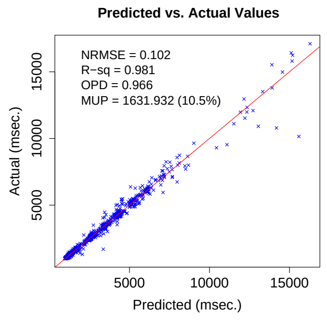
In the figures that follow, we plot the predicted versus actual values of the XNAV CPU cost. Each point in the plot corresponds to a query. The solid 45° line corresponds to 100% accuracy. We also display in each plot the accuracy measures defined above. For ease of comparison, we display in parentheses the MUP error as a percentage of the actual CPU cost.



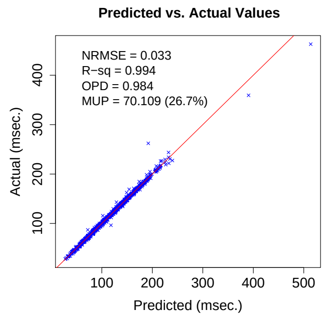
(a) `rf.xml` (CP queries)



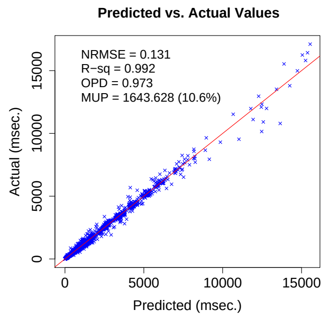
(b) `rd.xml` (CP queries)



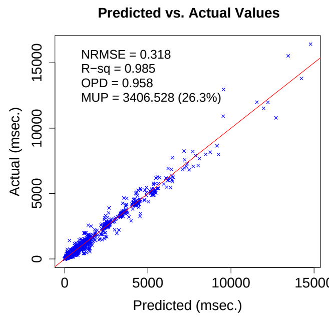
(c) `nf.xml` (CP queries)



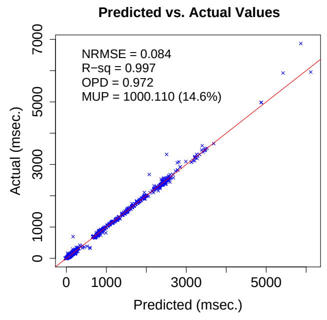
(d) `nd.xml` (CP queries)



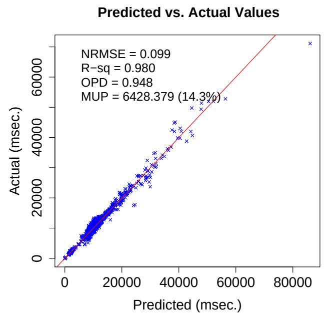
(e) Mixed data sets (CP queries)



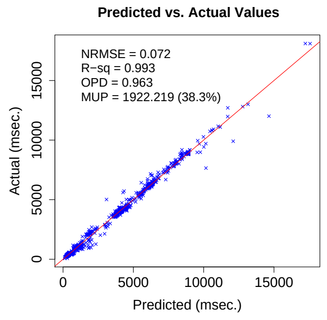
(f) Mixed data sets (Mixed queries)



(g) XMark mixed queries



(h) TPC-H mixed queries



(i) XBench TC/MD mixed queries

Figure 5: Accuracy of COMET for synthetic, benchmark, and real-world workloads

4.2.1 Synthetic Data

Figures 5(a)–5(f) illustrate the accuracy of COMET using the synthetic workloads, which systematically "stress test" COMET. We show the results only for CP queries; results using other queries are similar.

The synthetic datasets are generated according to the recursiveness and the depth of the XML tree. The four combinations produce four XML data sets: `rf.xml`, `rd.xml`, `nf.xml`, and `nd.xml`, where "`r`", "`n`", "`f`", and "`d`" stand for "recursive", "non-recursive", "flat", and "deep". These combinations represent a wide range, and usually extreme cases, of different properties in the documents. Table 1 displays various characteristics of the synthetic data sets.

Figures 5(a)–5(d) show results for CP queries on relatively homogeneous data sets. COMET's accuracy is very respectable, with errors ranging between 3% and 17%. Figure 5(e) shows results for CP queries with mixed data sets, and Figure 5(f) shows the results of mixed SP, BP, and CP queries with mixed data sets. We note that the presence of heterogeneous XML data does not appear to degrade accuracy when the queries are of the same type. That is, it suffices to use a single mixed set of data to train COMET. A comparison of Figures 5(e) and 5(f) indicates that the presence of different query types can adversely impact COMET's accuracy. This result is borne out by other experiments (not reported here), and suggests that query type might fruitfully be included as an additional (qualitative) feature.

4.2.2 Benchmarks and Real-World Data

The TPC-H benchmark is relational data wrapped in XML tags. The schema is very regular: it has no recursion and the tree is quite flat. The XMark [20] data set is generated with scale factor 0.1. It has a fair amount of recursion and the tree is fairly deep. The NASA data set[2] is real-world data having a small degree of recursion and medium depth. We also used two data sets from XBench [24]. The data-centric multi-document (DC/MD) data set models the Web-based e-commerce transactional data TPC-W. It consists of 25,920 small files of size 1 KB to 2 KB. The documents are non-recursive and quite flat. The text-centric multi-document (TC/MD) data set has statistical properties similar to the Reuters news corpus and the Springer digital library. This data set consists of $2,422$ files with various sizes from 1 KB to 100 KB. The documents contain recursion and some of them are quite deep. Other characteristics of the data sets are given in Table 1.

Figures 5(g), 5(h), and 5(i) show COMET's accuracy on the XMark, TPC-H, and XBench TC/MD data, respectively, using mixed SP, BP, and CP queries. We omit the figures for the NASA and XBench DC/MD data as they are very similar. COMET performs consistently well on all of these data sets. As in the synthetic case, COMET's accuracy is fairly insensitive to the type of data, making it suitable in an environment with heterogeneous data or changing schemas.

4.3 Effect of Errors in SP Statistics

To test COMET's sensitivity to errors in the SP statistics, we multiplied each statistic by a random nonnegative "error ratio" prior to training and testing. We observed the values of NRMSE, R-sq, and OPD as we varied the expected value of the error ratios. Figure 6 displays results for using mixed queries on the NASA data; results for other scenarios are similar.

As can be seen, COMET remains accurate despite the perturbation of the SP statistics. A key reason, as mentioned previously, is that COMET is both trained and tested using estimated feature values. So long as the feature-value estimates err in a consistent way (which they tend to do in practice), COMET can automatically compensate for the bias and produce accurate cost estimates. This feature allows us to use fairly simple statistics, as well as efficient algorithms

[2]Available at `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

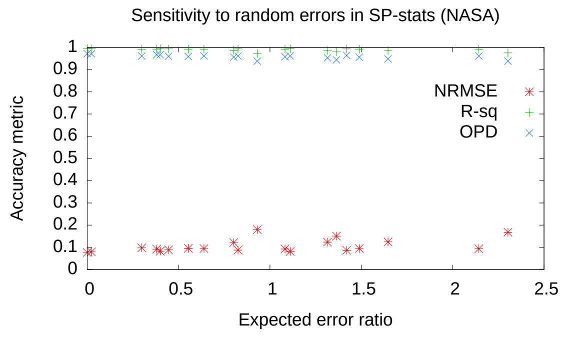


Figure 6: Sensitivity of accuracy to SP stats errors (NASA, mixed queries)

to estimate feature values, without compromising the accuracy of cost prediction.

4.4 Efficiency of COMET

There are two types of cost incurred by the COMET system over and above the usual costs incurred by a query optimizer: the cost of collecting the training data and the cost of building the prediction model from the training data. In the self-tuning scenario discussed in Section 1, the test queries are generated as part of the production workload, so that the first type of cost reduces to the overhead of recording and maintaining the query feedback results. Experience with query feedback systems [22] suggests that this additional monitoring cost is small in practice (less than 5% overhead). To assess the magnitude of the second type of cost, we tested 190 training sets of sizes $20, 40, \ldots, 3800$. In every case, the time to build the TR model is less than 1 second, ranging from 0.36 to 0.83 second. Such fast performance greatly simplifies the issue of how to update the cost model—the optimizer can simply build a new model from scratch whenever necessary.

4.5 Size of the Training Set

We investigated COMET's learning rate using CP queries over a heterogeneous synthetic data set comprising 3982 training points. We selected 50 random queries as test queries. Then, from the rest of the data set, we randomly chose 20 points as the first training set, built the TR model, and then computed the NRMSE for the 50 test queries. We then added 20 more queries to the training set, rebuilt the TR model, and recomputed the NRMSE for the 50 test queries. Continuing in this manner, we generated the learning curve displayed in Figure 7.

As can be seen from the figure, accuracies of around 10% can be achieved with a training-set size of about 1000 training queries; as discussed above, for this number of queries a TR model can be built in less than one second.

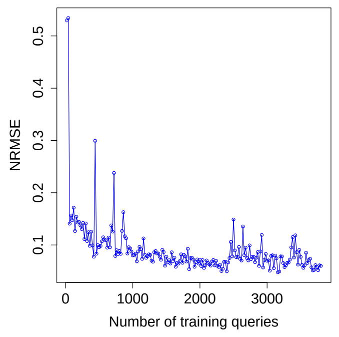


Figure 7: Effect of training-set size on accuracy

5 Conclusion

As query operators become more and more elaborate and data becomes more complex, the traditional approach of using detailed analytical models to estimate query costs is becoming increasingly unworkable. In this paper, we have outlined the COMET statistical learning approach to cost estimation, and have demonstrated its feasibility by applying it to the XNAV operator. To our knowledge, COMET represents the first proposed solution to the problem of XML cost modeling. COMET avoids the need for detailed cost models: the problem is reduced to the simpler task of identifying a sufficient (usually small) set of cost-determining features and developing a relatively simple method of estimating each feature. A key advantage of our approach is that it permits adaptation and flexibility in the face of changing workloads and a changing computing environment. Such flexibility is becoming increasingly important in light of current trends toward highly distributed systems composed of extremely heterogeneous, possibly remote and/or unreliable data sources.

We plan to apply the COMET to the problem of estimating I/O costs and also to devise mechanisms for automatic feature identification. We also plan to refine our methods for dynamically maintaining the learning model in order to minimize overheads while dealing effectively with multiuser environments.

Acknowledgment

We wish to thank Edwin Pednault for his help and advice with respect to the TR algorithm.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. VLDB 2001.

[2] A. Aboulnaga, P. J. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated Statistics Collection in DB2 UDB. VLDB 2004.

[3] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002.

[4] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. ICDE 2003.

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. Available at `http://www.w3.org/TR/xpath20/`.

[6] J. Boulos, Y. Viemont, and K. Ono. A Neural Networks Approach for Query Cost Evaluation. *IPSJ Journal*, 2001.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. SIGMOD 2002.

[8] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. Available at `http://www.w3.org/TR/xmlquery-use-cases`.

[9] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. SIGMOD 2002.

[10] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. VLDB 1997.

[11] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed Mode XML Query Processing. VLDB 2003.

[12] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Springer, 2001.

[13] Z. He, B. S. Lee, and R. R. Snapp. Self-tuning UDF Cost Modeling Using the Memory-Limited Quadtree. EDBT 2004.

[14] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 14(2), 2005.

[15] B. S. Lee, L. Chen, J. Buzas, and V. Kannoth. Regression-Based Self-Tuning Modeling of Smooth User-Defined Function Costs for an Object-Relational Database Management System Query Optimizer. *The Computer Journal*, 2004.

[16] R. Natarajan and E. P. D. Pednault. Segmented Regression Estimators for Massive Data Sets. SDM 2002.

[17] E. Pednault. Transform Regression and the Kolmogorov Superposition Theorem. Technical Report RC23227 (W0406-014), IBM Thomas J. Watson Research Center, 2004.

[18] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. SIGMOD 2004.

[19] A. Rahal, Q. Zhu, and P.-Å. Larson. Evolutionary Techniques for Updating Query Cost Models in a Dynamic Multidatabase Environment. *The VLDB Journal*, 13(2), 2004.

[20] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. SIGMOD 1979.

[22] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO — DB2's LEarning Optimizer. VLDB 2001.

[23] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. VLDB 2004.

[24] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. ICDE 2004.

[25] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. ICDE 2004.

[26] Q. Zhu and P.-Å. Larson. Building Regression Cost Models for Multidatabase Systems. PDIS 1996.