ROX: The Robustness of a Run-time XQuery Optimizer Against Correlated Data

Riham Abdel Kader [#1], Peter Boncz [*2], Stefan Manegold [*3], Maurice van Keulen [#4]

[#] *University of Twente*
Enschede, The Netherlands
[1] r.abdelkader@utwente.nl
[4] m.vankeulen@utwente.nl

[*] *CWI*
Amsterdam, The Netherlands
[2] P.Boncz@cwi.nl
[3] Stefan.Manegold@cwi.nl

***Abstract*— We demonstrate ROX, a run-time optimizer of XQueries, that focuses on finding the best execution order of XPath steps and relational joins in an XQuery. The problem of join ordering has been extensively researched, but the proposed techniques are still unsatisfying. These either rely on a cost model which might result in inaccurate estimations, or explore only a restrictive number of plans from the search space. ROX is developed to tackle these problems. ROX does not need any cost model, and defers query optimization to run-time intertwining optimization and execution steps. In every optimization step, sampling techniques are used to estimate the cardinality of unexecuted steps and joins to make a decision which sequence of operators to process next. Consequently, each execution step will provide updated and accurate knowledge about intermediate results, which will be used during the next optimization round. This demonstration will focus on: *(i)* illustrating the steps that ROX follows and the decisions it makes to choose a good join order, *(ii)* showing ROX's robustness in the face of data with different degree of correlation, *(iii)* comparing the performance of the plan chosen by ROX to different plans picked from the search space, *(iv)* proving that the run-time overhead needed by ROX is restricted to a small fraction of the execution time.**

I. Motivation for ROX

In their search for a good execution plan, relational compile-time optimizers rely on pre-collected statistics and an accurate cost model to estimate the selectivity of operators. The accuracy of these estimations, although satisfying for a single operator, exponentially degrades through the plan resulting, in the case of big queries, in the execution of bad plans. Additionally, to simplify the cardinality estimation problem, these optimizers assume the attribute value independence heuristic, which does not hold in real-life data, resulting in wide errors in estimations and rendering optimizers helpless in the face of correlation. Some databases solve this problem by creating indeces on multiple columns or collecting group column statistics, however; the challenge remains in knowing beforehand which columns are correlated and in storing and maitaining the statistics. In the XML context, data and query operators are more complex than relational ones because of the structural nature and expressiveness of the language, therefore cost models developed for XML are simply not available or, if present, by far less accurate than their relational counterparts, turning the field of cardinality estimation in the context of XQuery into deadly optimizer quicksand. Moreover XQuery accesses data through the `fn:doc`(*url*), thus potentially with a "table" name that is computed at runtime. In such cases, static information to guide a query optimizer cannot be available.

To overcome the shortcomings of compile-time optimizers, adaptive query processing techniques have been developed. Some of these apporaches, like [1], first generate a potential good plan and re-optimizes it when the observed cost exceeds static estimations. The quality of choices made by these techniques still highly depend on the quality of collected statistics and cost model, and therefore can not spot, early enough, opportunities where the existence of correlations can speed up query evaluation. To overcome this problem, some approaches monitor the performance of query execution and feedback the observations to the optimizer to adjust the cost model and statistics [2], however such optimizers are very complex modules. Routing-based techniques [3] optimize a query by routing each tuple to the most efficient sequence of operators based on observed statistics. The drawback of such techniques is that they require the presence of symmetric operators, can only cover a restrictive number of alternative plans, and suffer from the overhead of maitaining query execution states.

The ROX approach does not depend on any collected statistics or cost model. It defers optimization to run-time, interleaving sampling-based optimization decisions and execution steps, gaining at each iteration better knowledge about intermediate results characteristics and consequently which execution strategy to follow.

II. The Run-time XQuery Optimizer

A. Join Graph

ROX [4] fully integrates plan optimization with processing by iteratively switching between optimization and execution steps. It takes as input a join graph which is a representation of the joins and XPath steps in the XQuery without any implication on their order of execution. Therefore, the application of ROX is preceded by a compilation phase which

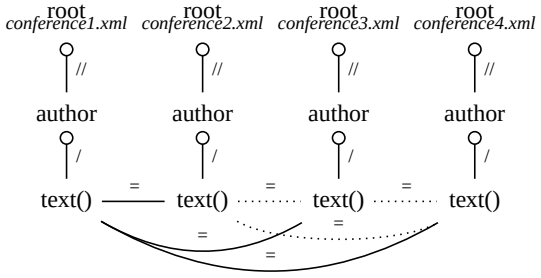


Fig. 1. Join Graph of the 4-way join XQuery

consists of XQuery parsing, normalization, compilation, peephole driven optimization and finally join graph extraction. The first four steps, described in [5], result in a DAG-shaped plan of relational operators. The join graph isolation process [6] aims at separating joins from blocking operators by pushing the latter above the joins creating a plan tail whose execution ensures XQuery semantics (duplicate free and required order). This results in a cluster of joins, selections, and projections forming the to be optimized join graph. Fig. 1 shows the join graph of the following XQuery:

```
for $a1 in doc("conference1.xml")//author,
    $a2 in doc("conference2.xml")//author,
    $a3 in doc("conference3.xml")//author,
    $a4 in doc("conference4.xml")//author
where $a1/text() = $a2/text() and
      $a1/text() = $a3/text() and
      $a1/text() = $a4/text()
return $a1
```

The vertices in a join graph represent index-selectable sets of element, text, and attribute nodes. The edges specify all XPath step and join relationships between the nodes. A step join between two relations s_1 and s_2 is depicted by an edge $s_1 \circ\!\!-\!\!\!\stackrel{ax}{}\!\!\!-\!\!\circ\, s_2$ where the label ax defines the axis of the step. The circle "$\circ$" denotes the direction of the step, *i.e.*, which of the two relations represents the context node sequence of the step. Note that the direction is only a representational issue; the algorithm may very well decide to execute it in the reverse direction. A relational join between s_1 and s_2 is depicted as $s_1 \stackrel{=}{---} s_2$. The dotted edges in Fig. 1 represent join equivalences, and are added by ROX to broaden the search space of plans allowing for more flexibility to find a good plan.

B. Operators and Index Structures

As an XML database backend, we use the open-source system MonetDB/XQuery [7]. In MonetDB, XML documents are shredded into relational tables using a range-based pre/post encoding representation. That is, every XML node is stored in a separate relational tuple, and is referred to using the node identifier *pre*. An advantage of the adopted range-encoding is that XPath axes can be evaluated with only standard relational operators. It has been proved, however, that performance gain is possible if a *tree aware* operator is used [7]. As a consequence, the XQuery module of MonetDB has extended its relational algebra with the staircase join operator, a structural join capable of exploiting the tree properties of the pre/post plane. The staircase join can evaluate with linear complexity all XPath axes by making at most a single sequential pass over the document, returning duplicate-free, in document order results. Note that the ideas in ROX can be used with other operators and do not require the presence of a staircase join.

In addition, MonetDB/XQuery implements an element index and a value index that covers all text and attribute values. All index lookup operations provide a list of node identifiers, duplicate-free and in document order. Given a qualified name q, the *element index* returns the list of all elements in the document D satisfying q:

$$\overset{D_{elt}}{\triangledown}(q) = \{pre | pre \in D_{elt} \wedge \texttt{qname}(pre) = q\}$$

The value index supports a hash-based index for string equality lookups on text and attribute nodes. Given a value v, the *text value index* returns the list of all candidate text nodes in the document D having a value v:

$$\overset{D_{text}}{\triangledown}(v) = \{t | t \in D_{text}\text{: } \texttt{fn:data}(t) = v\}$$

Given a value v, the *attribute value index* returns the list of the parent elements with qualified name q_{elt} of all candidate attributes with qualified name q_{attr} having a value v:

$$\overset{D_{attr}}{\triangledown}(v, q_{elt}, q_{attr}) = \{e | e \in D_{elt}\text{: } e@q_{attr} = v \wedge \texttt{qname}(e) = q_{elt}\}$$

All indices are stored in a materialized and physically clustered (index organized) tables. The complexity of an index lookup, and consequently the cost of finding the count of nodes that satisfy an index lookup, is *independent* of the index result size, and is logarithmic to the index size.

C. Sampling Operators

The ROX algorithm intertwines optimization and execution steps where optimization consists of estimating the cost of operators using sampling techniques. ROX samples a join or step operator by first randomly picking a small set of tuples from one of the operator's input and then feeding the chosen subset into the operator. As start samples, ROX uses either a synthetic single-tuple relation containing the document's root node, or a set of tuples drawn from element and text indices.

Although sampling an operator joins one of its input tables with a small set of tuples, the result size might be large in case of high join hit ratios – the Cartesian product in the worst case. To eliminate the risk of generating large sampling results, ROX sets a limit on the number of tuples produced by a sampling operations. This is done by stopping the sampled join process when the count of generated results has reached a certain *cutoff*. To still ensure an accurate estimate of the join's cardinality, the sampling process will keep track of the fraction f of the sampled tuples that have been processed and will extrapolate the size of the full result R as $|R| = \frac{cutoff}{f}$.

In our join graph representation, edges correspond to join and XPath step operators. We therefore define the sampling function *Sample*$(e, S, T, cutoff)$ as the partial execution of the join or step operator corresponding to the edge e using as input the sample S and the table T where execution stops as soon as the size of the generated result reaches the *cutoff* limit.

Another requirement for efficient sampling is the use of physical operators that have the "zero-investment" property with respect to the sampled input. This represents operators whose cost only depends on the cardinality of the input and do not require any investment, like sorting, prior to starting result generation. In our scenario, all operators used for sampling and executing the join graph, including staircase joins, obey the zero-investment condition.

D. The ROX Algorithm

We give here a concise explanation of the ROX algorithm and refer readers to [4] for a detailed description. We first define the following notation. Given a join graph $G = (V, E)$, and a vertex $v \in V$:

- $T(v)$ represents a table with all XML nodes that satisfy the annotated name and range-predicates of v.
- *SampleSet*(v, τ) represents a table containing a sample of XML nodes of size τ randomly chosen from $T(v)$. Unless specified otherwise, we used a default τ of 100.

The main algorithm of the run-time optimizer consists of two phases. The first phase initializes the Join Graph, and the second alternates search space exploration and execution of operators until all operations of all edges are executed.

The join graph initialization estimates the cardinality of nodes satisfying each vertex in the graph and materializes a sample of these nodes. This is efficiently provided by an index look-up. The materialized samples are then used to estimate the weight of each edge which represents an estimation of the result cardinality of the step or join operator associated to it. It is computed by first sampling the associated operator, as described in Section II-C, and then linearly extrapolating the result size of the sampling. For $e = (v_1, v_2)$, we define:

$$\begin{aligned} &\textit{EstimateWeight}(e) = \frac{|T(v)|}{\tau} \times \\ &\quad |\textit{Sample}(e, \textit{SampleSet}(v, \tau), T(v'), \tau)| \\ &\text{where } (v, v') = \begin{cases} (v_1, v_2) & \text{if } |T(v_1)| < |T(v_2)| \\ (v_2, v_1) & \text{otherwise} \end{cases} \end{aligned}$$

The sample is chosen from the vertex of e that has the smallest size, since picking a sample from a smaller table provides a more representative set of tuples.

The second phase of the algorithm alternates between join graph exploration and operator execution. The exploration searches for a superior path segement (sequence of steps and joins) by applying *chain sampling*, a process that efficiently samples a sequence of operators using the sampling result of one operator as input to the sampling of the subsequent one. As soon as a path segment is found to be superior to others, the sampling stops, the associated steps and joins are executed, their results are materialized. The newly materialized intermediate result of each vertex v along the executed path is used to update the weight of all un-executed edges outgoing of v. This is accomplished by (re)sampling the edges, using as input a sample of the new result generated from the joins' execution. Then the exploration process searching for the superior path segment restarts, benefiting from the newly obtained data and more accurate statistical knowledge. These steps iterate until all operations in the join graph are executed. Note that re-sampling the edges after each execution allows ROX to identify arbitrary correlations between joins.

The superiority criterion used by ROX for picking the next edge to execute is based on a heuristic that opts for the operator that generates the smallest intermediate result, in other words the edge e with the smallest weight. Since this choice might be a *local minimum*, ROX adopts a chain sampling strategy to climb the hill and investigate the presence of a better execution path which produces a result with a smaller size. Therefore the paths that branch from the edge e are explored and sampled ahead. Obviously, chain sampling is only performed if one of the vertices of e is branching. Otherwise, e is simply executed since it has no neighboring un-executed edges to explore. In the former case, the branching vertex with the smallest cardinality will be the starting point of exploration.

Chain sampling explores the branches in a breadth first manner, defining path segments in the join graph. Each iteration samples the next edge in every branch, extending each path segment with an extra edge. In some iterations, when branching vertices are encountered, new path segments are created. For each path segment p, we note:

- *cost*(p) as the estimated cumulative cardinality of all intermediate results of p. Each time an edge e is sampled and added to p, the cost of p is incremented with the cardinality estimated from the sampling of e.
- *sf*(p) as the scale factor of p. It represents the join hit ratio ($\frac{output\_size}{input\_size}$) resulting from executing p.

After each sampling iteration, the optimizer compares all pairwise combinations of path segments to find a superior one using the following *stopping condition*:

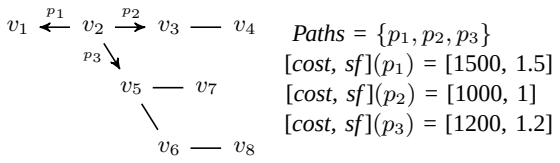
$$\underbrace{\textit{cost}(p_i)}_{①} + \underbrace{\textit{sf}(p_i) * \textit{cost}(p_j)}_{②} \leq \underbrace{\textit{cost}(p_j)}_{③}$$

① : cost of executing p_i
② : cost of executing p_j using the new data returned from the execution of p_i
③ : cost of executing p_j

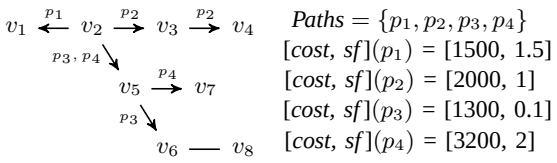
The idea behind the equation is, given two paths p_i and p_j, if p_i's execution followed by p_j's execution is cheaper than executing p_j alone, we can safely execute p_i. For example, if *cost*(p_j) was estimated to be equal to 1000 and the execution of p_i will reduce the intermediate result by half (*i.e.* *sf*$(p_i) = 0.5$), then the cost of executing p_j after p_i will be equal to 500. If p_i happens to cost less than 500 satisfying the above condition, it is guaranteed that $p_i p_j p_k$ is cheaper to execute than $p_j p_k p_i$ for any extension p_k of the path segment p_j. Once the superior path segment p_i is found, chain sampling is halted and all operators along p_i are executed.

If after each sampling round, the *stopping condition* is never satisfied, chain sampling will progress until all branches are fully explored. In this case, the path segment p_i that satisfies the following equation is chosen for execution:

$$\textit{cost}(p_i) + \textit{sf}(p_i) * \textit{cost}(p_j) \leq \textit{cost}(p_j) + \textit{sf}(p_j) * \textit{cost}(p_i)$$



2.1 The first round of chain sampling



2.2 The second round of chain sampling

Fig. 2. Illustration of Chain Sampling

Applying the *cutoff* limit during each sampling round, as we have described in Section II-C, might result in degrading the quality of the sample set, making it less representative of the original data. To avoid this problem, we increment the sampling cutoff with τ after each round.

Example. We illustrate chain sampling with the join graph in Fig. 2.1. Suppose that the edge with the smallest weight is (v_2, v_3), and that the cardinality of v_2 is smaller than v_3 (so v_2 is the starting point of chain sampling). Sampled edges are indicated by arrows and are labeled with the path segment they belong to. In Figure 2.2, the *stopping condition* holds for $i = 3$ and $j = [1, 2, 4]$, then chain sampling is stopped although the edge (v_6, v_8) can still be sampled. In this case, a selective correlation between the elements v_2, v_5 and v_6 is detected, and as a result the edges in p_3 will be executed instead of the edge (v_2, v_3) which was found earlier to be the best.

III. Demonstration Outline

To test and demonstrate our approach, a prototype of ROX has been implemented in Java. Pathfinder, the XQuery processor implemented on top of MonetDB [7], generates the isolated join graph for a given XQuery and provides it as input to ROX. The demonstration will show the following items:

(i) The demo GUI is a Java applet that depicts the join graph of a given XQuery. Documents from the XMark benchmark[1] and the DBLP XML dataset[2] will be shredded into MonetDB. Users can then enter their own queries and examine the optimization process as conducted by ROX (join graph initialization, chain sampling, execution), or opt for running one of our prepared scenarios. We have set up two different scenarios that clearly show the potentials of ROX in which queries are already defined for each of the datasets.

[1] `http://monetdb.cwi.nl/xml/`
[2] `http://dblp.uni-trier.de/xml/`

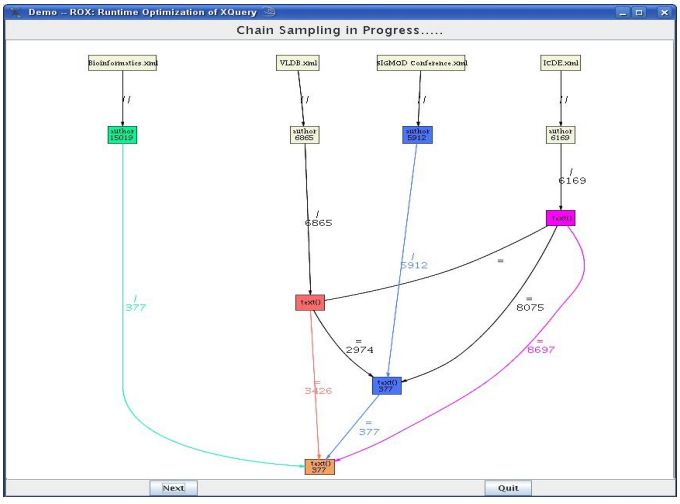


Fig. 3. A snapshot of the demonstration of the DBLP scenario. Blue edges denote executed operators while other colored edges represent chain sampling

(ii) For the DBLP scenario (Fig. 3), the used XQuery and join graph are the ones presented in Section II-A. The queried documents are extracted from the DBLP dataset which was split into multiple XML documents, one for each conference, after being scaled to ∼45 GB (*i.e.* 100 times bigger). By replacing the 4 documents in the XQuery with 4 conferences chosen from one or multiple research areas, the degree of correlation in the query is varied: it is in general more likely that authors publish in one research area, than that an author publishes in multiple different research areas. Users will choose the research areas from which each of the 4 documents will be picked, creating therefore different degrees of correlations. Then they can watch a step-by-step execution of ROX and assess its robustness against correlations.

(iii) In addition to displaying the steps taken by ROX to optimize a query in the DBLP scenario, the quality of the selected plan will be assessed. For comparison, we consider three plans each denoting a different joins' order. The first two plans represent the joins' order that generates respectively the *smallest* and *largest* cumulative intermediate result size. The third plan is the one that would be generated by a "*classical*" compile time optimizer, which is capable of accurately estimating the cardinality of operations carried on a single document but lacks the ability of estimating the correlations existing among several documents. This results in a joins' order that reflects a "smallest-input-first" heuristic where the two smallest inputs are joined first, which is then joined with the third largest input, and so on. The execution time of ROX's plan will be compared to that of the three considered plans.

(iv) In ROX, deferring optimization to run-time comes with the risk of adding the sampling overhead to the plan's execution time. The demo will show that ROX is able to restrict the overhead to a small fraction of the query's execution time.

References

[1] S. Babu, P. Bizarro, and D. DeWitt, "Proactive Re-Optimization," in *SIGMOD*, 2005.
[2] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO - DB2's LEarning Optimizer," in *VLDB*, 2001.
[3] R. Avnur and J. Hellerstein, "Eddies: continuously adaptive query processing," *SIGMOD*, 2000.
[4] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen, "ROX: Run-Time Optimization of XQueries," in *SIGMOD*, 2009.
[5] T. Grust, "Purely Relational FLWORs," in *<XIME-P/>*, 2005.
[6] T. Grust, M. Mayr, and J. Rittinger, "XQuery Join Graph Isolation," in *ICDE*, 2009.
[7] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine," in *SIGMOD*, 2006.