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

Riham Abdel Kader  
University of Twente  
Enschede, The Netherlands  
r.abdelkader@utwente.nl

Stefan Manegold  
CWI  
Amsterdam, The Netherlands  
Stefan.Manegold@cwi.nl

Peter Boncz  
CWI  
Amsterdam, The Netherlands  
P.Boncz@cwi.nl

Maurice van Keulen  
University of Twente  
Enschede, The Netherlands  
m.vankeulen@utwente.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 benefiting from information about intermediate data accurately estimated at run-time. The problem of join ordering has been well studied and extensively researched in the relational context which has led to the development of several optimization techniques like, for example, deterministic algorithms, randomized algorithms and genetic algorithms. These algorithms optimize the order of joins by exploring at compile time the search space of plans using different search strategies, and a cost model that associates an execution cost to each candidate plan. As the decisions made by a traditional optimizer highly depend on estimated values about, among others, document characteristics, intermediate results cardinality, and system load; the quality of the decisions are sometimes poor resulting in bad plans being picked for execution.

To overcome this problem, we propose the ROX algorithm which defers query optimization to run-time when such estimations can be made with a very high accuracy level.

ROX intertwines optimization and execution steps. In every optimization step, ROX uses sampling techniques to estimate the cardinality of unexecuted steps and joins, and makes a decision which sequence of operators to process next. Consequently, each execution step will provide updated knowledge and more accurate information 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 plan with 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 to find a good plan is restricted to a small fraction of the query’s execution time.

1. MOTIVATION FOR ROX

In order to choose a good execution plan from the search space of plans, a traditional optimizer estimates at multiple occasions the value of several factors that influence the cost corresponding to the different considered plans. The accuracy of the estimations can often be questionable.

In the relational context, there has been a significant number of attempts to develop good cost models to estimate the selectivity of operators. The accuracy of the estimations, although satisfying for a single operator, exponentially degrades through the plan. In the case of big queries, the propagated error can have a devastating effect on the choices made by the optimizer [5]. Additionally, to simplify the cardinality estimation problem, relational optimizers adopt the attribute value independence heuristic. In real-life data this assumption does not hold, rendering optimizers helpless in the face of correlation and resulting in wide errors in estimations [2]. Further, especially in the case of query pre-compilation, available resources might vary between the query optimization phase and execution time, and as a result the picked execution plan might perform as the worst choice. Finally, in the case of parameterized queries issued through pre-defined forms where every user enters a different set of values for the parameters, a pre-compiled query plan might need to be adapted to the given values.

In the more specific case of XQuery, query optimization is faced with some additional challenges. XQuery accesses data through the find doc(url), thus potentially with a “table” name that is computed at runtime; a feature typically encountered in document retrieval queries where a first IR phase identifies interesting documents. In such cases, static information to guide a query optimizer cannot be available. Most importantly, however, XML 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.

The crux of the ROX approach is the insight that using run-time feedback to gain more knowledge about the actual data distribution and system load is a promising way forward. ROX overcomes all of the aforementioned problems by deferring optimization to run-time, interleaving optimization decisions and execution steps, gaining at each iteration better knowledge about intermediate results and the query execution strategy to follow.
2. THE RUN-TIME XQUERY OPTIMIZER

2.1 Join Graph

ROX [6] 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 consists of XQuery parsing, normalization, compilation, peep-hole driven optimization and finally join graph extraction. The first four steps, described in [3], result in a DAG-shaped plan of relational operators. The join graph isolation process [4] aims at separating joins from blocking operators by pushing the latter above the joins creating a plan tail whose execution process [4] aims at separating joins from blocking operators by order of execution. Therefore, the application of ROX is preceded and XPath steps in the XQuery without any implication on their order of execution. It iteratively switches between optimization and execution steps. It guarantees XQuery semantics (duplicate free and required order). This results in a cluster of joins, selections, and projections forming the to be optimized join graph. This operation is performed by a finely-tuned set of rewriting rules which identify small patterns of operators and rewrite them based on properties pre-derived and associated to each operator. As an example, Figure 1 shows the join graph of the following XQuery which asks for all authors that have published in 4 different conferences and/or journals:

```xml
for $a1 in doc("DOC1.xml")//author,
    $a2 in doc("DOC2.xml")//author,
    $a3 in doc("DOC3.xml")//author,
    $a4 in doc("DOC4.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 node sets, 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 \( s_1 \xrightarrow{ax} s_2 \) where the label \( ax \) defines the axis of the step. The circle \( "o" \) 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 two relations \( s_1 \) and \( s_2 \) is depicted as \( s_1 \leftarrow s_2 \). The dotted edges in Figure 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.

2.2 Operators and Index Structures

As an XML database backend, we use the open-source system MonetDB/XQuery [1]. 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 [1]. 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 \( q \):

\[
D_{elt}(q) = \{pre | pre \in D_{elt} \land \text{qname}(pre) = q\}
\]

The value index supports a hash-based index for string equality lookups on text and attribute nodes. The basic idea of the value index is an ordered store of \((\text{val}, q_{elt}, q_{attr}, pre)\) tuples. Such a structure can in principle be used to find text-, element- and attribute-nodes using equi- or range-lookup on value. Given a value \( v \), the \text{value text index} returns the list of all candidate text nodes in the document \( D \) having a value \( v \):

\[
D_{text}(v) = \{t | t \in D_{text}; \text{fn:data}(t) = v\}
\]

Given a value \( v \), the \text{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 \( v \) in the document \( D \):

\[
D_{attr}(v, q_{elt}, q_{attr}) = \{e | e \in D_{elt}; e[@q_{attr}=v]\text{and}\text{qname}(e)=q_{elt}\}
\]

All indices are stored as in a materialized and physically clustered (index organized) tables, with node identifiers in index order. 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.

2.3 Sampling Operators

The ROX algorithm intertwines optimization and execution steps where optimization consists of estimating the cost of operators using sampling techniques. Sampling an operator consists of the optimizer’s input and then feeding the chosen subset into the operator. As start samples, ROX uses either a synthetic single-tuple relation orator’s execution using a subset of tuples randomly selected from its input. In other words, ROX samples a join or step operator by first randomly picking a small number 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 root node of the document, 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 \( \text{cutoff} \). 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 \( |R| = \frac{|S|}{f} \). In our join graph representation, edges correspond to join and XPath step operators. We therefore define the sampling function
Sample \( (e, S, T, \text{cutoff}) \) 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 no 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.

### 2.4 The ROX Algorithm

The ROX algorithm is explained in detail in [6]. We give here only a concise description of its steps which will also be illustrated during the demonstration. We first define the following notation.

- \( T(v) \) represents a table with all XML nodes that satisfy the annotated name and range-predicates of \( v \).
- \( \text{SampleSet}(v, \tau) \) represents a table containing a sample of XML nodes of size \( \tau \) randomly chosen from \( T(v) \).

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 have been 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 size of each edge. The weight of an edge is an estimate of the result cardinality of the step or join operator associated to it. It is computed by first sampling the operator associated to the edge, as described in Section 2.3, and then linearly extrapolating the result size of the sampling. For \( e = (v_1, v_2) \), we define:

\[
\text{EstimateWeight}(e) = \frac{\vert T(v_1) \vert \times \text{Sample}(e, \text{SampleSet}(v_1, \tau), T(v'), \tau)}{\text{SampleSet}(v_1, \tau)}
\]

where \( (v_1, v_2) = \begin{cases} (v_1, v_2) & \text{if } \vert T(v_1) \vert < \vert T(v_2) \vert \\ (v_2, v_1) & \text{otherwise} \end{cases} \)

The second phase of the algorithm alternates between exploring the join graph and the execution of operators until all operations of all edges have been executed.

The second phase of the algorithm alternates between exploring the join graph and the execution of operators. The exploration is performed by efficiently sampling path segments (sequences of steps and joins) using the ChainSample function described in Section 2.4.1. 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)sample 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.

#### 2.4.1 Chain Sampling

Chain sampling is the process of sampling a sequence of operators using the sampling result of one operator as input to the sampling of the subsequent one. Chain sampling is not only essential to detect correlations that exist between a chain of operators, but also to avoid a local minimum that might be encountered while exploring the search space.

The superiority criterion used by ROX while picking the next edge to execute is based on a heuristic that opts for the edge that has the smallest intermediate result cardinality, in other words the edge 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. This strategy consists of exploring and sampling ahead in the paths that branch from the edge with the smallest weight. Obviously, chain sampling is only performed if one of the edge’s vertices is branching. Otherwise, the edge is simply executed since it has no neighboring un-executed edges to explore. In the former case, the branching vertex with the smallest cardinality is chosen as 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. After each sampling round, path segments are compared by examining their associated properties. If a superior path segment is recognized, then chain sampling is halted and all operators along the path are executed.

The 2 following properties are associated to each path segment \( p \):

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

The optimizer compares all pairwise combination of path segments using the following stopping condition:

\[
\begin{align*}
\text{cost}(p_i) + @sf(p_i) \cdot \text{cost}(p_j) & \leq \text{cost}(p_j) \\
& \quad (1) \quad \text{(1): cost of executing } p_i \\
& \quad (2) \quad \text{(2): cost of executing } p_j \text{ using the new data returned from the execution of } p_i \\
& \quad (3) \quad \text{(3): cost of executing } p_j
\end{align*}
\]

The idea behind the equation is that, given two paths \( p_i \) and \( p_j \), if the execution of \( p_i \) followed by the execution of \( p_j \) is cheaper than executing \( p_j \) alone, we can safely execute \( p_i \). For example, if \( \text{cost}(p_j) \) was estimated to be equal to 1000 and the execution of \( p_i \) will reduce the intermediate result by half (i.e. \( \text{sf}(p_i) = 0.5 \)), 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 \( p_i \) will be cheaper to execute than \( p_i \) followed by any extension of \( p_i \) of the path segment \( p_j \).

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:

\[
\text{cost}(p_i) + @sf(p_i) \cdot \text{cost}(p_j) \leq \text{cost}(p_j) + @sf(p_j) \cdot \text{cost}(p_i)
\]

Applying the cutoff limit during each sampling round, as we have described in Section 2.3, 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 \( \tau \) after each round.
2.1 The first round of chain sampling

\[ Paths = \{p_1, p_2, p_3\} \]
\[ \{\text{cost, sf}\}(p_1) = [1500, 1.5] \]
\[ \{\text{cost, sf}\}(p_2) = [1000, 1] \]
\[ \{\text{cost, sf}\}(p_3) = [1200, 1.2] \]

2.2 The second round of chain sampling

\[ Paths = \{p_1, p_2, p_3, p_4\} \]
\[ \{\text{cost, sf}\}(p_1) = [1500, 1.5] \]
\[ \{\text{cost, sf}\}(p_2) = [2000, 1] \]
\[ \{\text{cost, sf}\}(p_3) = [1300, 0.1] \]
\[ \{\text{cost, sf}\}(p_4) = [3200, 2] \]

Figure 2: Illustration of Chain Sampling

Example. We illustrate the chain sampling process with the join graph shown in Figure 2.1. The edge with the smallest weight is \((v_2, v_3)\). Suppose that the cardinality of \(v_2\) is smaller than \(v_3\), so we choose \(v_2\) as 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, chain sampling was able to detect an existing selective correlation between the elements \(v_2, v_5, v_6, v_8\), and as a result the edges in path \(p_3\) will be executed instead of executing \((v_2, v_3)\) which was found earlier to be the best.

3. DEMONSTRATION OUTLINE

To test and demonstrate our approach, a prototype of ROX has been implemented in Java. We use the open-source system MonetDB/XQuery [1] as a database backend. Pathfinder, the XQuery processor implemented on top of MonetDB, generates the isolated join graph for a given XQuery and provides it as input to ROX. The demonstration aims at showing the following items:

- 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.
- For the DBLP scenario (Figure 3), the used XQuery and join graph are the ones presented in Section 2.1. The queried documents are extracted from the DBLP dataset which was split into multiple XML documents, one for each conference and journal. By replacing the 4 documents in the XQuery with 4 conferences and/or journals 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 can choose the research area from which each of the 4 documents will be picked, and can watch a step-by-step execution of the ROX algorithm.
- In addition to displaying the series of steps taken by ROX to optimize a query in the DBLP scenario, the quality of the selected joins’ order 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 corresponds to 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 an order of joins that reflects a simple “smallest-input-first” heuristic where the two smallest inputs are joined first, which is then joined with the second largest input, and so on. The execution time of ROX’s plan will be compared to that of the 3 considered plans.
- Deferring optimization to run-time comes with the risk of adding the overhead of exploration to the plan’s execution time. In ROX, the exploration overhead is caused by the use of sampling. We show that ROX is able to keep the overhead restricted to a small fraction of the query’s execution time.

4. REFERENCES


\(^1\)http://monetdb.cwi.nl/xml/
\(^2\)http://dblp.uni-trier.de/xml/