ROX: Run-time Optimization of XQueries

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

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

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

Maurice van Keulen
University of Twente
Enschede, The Netherlands
m.vankeulen@utwente.nl

ABSTRACT

Optimization of complex XQueries combining many XPath steps and joins is currently hindered by the absence of good cardinality estimation and cost models for XQuery. Additionally, the state-of-the-art of even relational query optimization still struggles to cope with cost model estimation errors that increase with plan size, as well as with the effect of correlated joins and selections.

In this research, we propose to radically depart from the traditional path of separating the query compilation and query execution phases, by having the optimizer execute, materialize partial results, and use sampling based estimation techniques to observe the characteristics of intermediates. The proposed technique takes as input a Join Graph where the edges are either equi-joins or XPath steps, and the execution environment provides value- and structural-join algorithms, as well as structural and value-based indices.

While run-time optimization with sampling removes many of the vulnerabilities of classical optimizers, it brings its own challenges with respect to keeping resource usage under control, both with respect to the materialization of intermediates, as well as the cost of plan exploration using sampling. Our approach deals with these issues by limiting the run-time search space to so-called "zero-investment" algorithms for which sampling can be guaranteed to be strictly linear in sample size. All operators and XML value indices used by ROX for sampling have the zero-investment property.

We perform extensive experimental evaluation on large XML datasets that shows that our run-time query optimizer finds good query plans in a robust fashion and has limited run-time overhead.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Performance

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

One of the tasks of a database optimizer is to enumerate at compile time several equivalent candidate execution plans and to identify the one to be executed based on its cost value. Several factors such as size of input, selectivity and available resources affect the cost of a given plan. The value of these factors is *estimated* by the optimizer, hence the accuracy of the estimations is not guaranteed.

In the well-studied case of relational query optimization, there exists three general dangers that undermine the accuracy of such estimates, even if we assume the presence a reliable cost model. In case of large query plans, the variance in estimation errors, while originally small for an individual operator, propagates through the plan in a multiplicative way, causing ever wider error variance [23]. Thus, the estimation precision of cost models exponentially deteriorates with query complexity. Further, it is still a common approach for optimizers to assume value independence of attributes in a document, while in real-life data often strong *correlations* exist. In such cases, this wrong assumption creates strong estimation errors. Finally, the available resources are crucial for plan efficiency; however the system load may change between query compilation and query evaluation time, especially in case of query pre-compilation.

In the case of XQuery processing, two additional factors undermine the accuracy of static query optimization. First, XML cost prediction, where rather than (foreign-key) equi-joins, structural join predicates come into play, is still an open research area. So, the earlier assumption of a working cost model does not hold in the XML case. A second problem is that in XQuery, parametric queries are more prevalent than in the relational case. A prominent example of that is that in SQL the tables accessed by a query are always known at compile time, while in XQuery data is accessed through the `fn:doc`(*url*) and `fn:collection`(*url*) functions, which may receive a run-time parameter. As the values of the parameters can strongly affect the cost of the query, and in case of document access can even prevent all access to data statistics, this is a serious roadblock towards reliable compile-time optimization of XQueries.

1.1 Run-time Query Optimization

In this paper we suggest a new approach to run-time query optimization. In contrast to dynamic query evaluation [16, 10], which determines at compile time a few alternative execution plans and decides at run-time based on a parametric comparison the one to execute, our ROX approach limits static compilation to normalization, simplification and the identification of so-called Join Graphs. These Join Graphs cluster XML navigation, relational joins, and structure and value predicates of an XQuery [18], and are subjected to the ROX run-time optimization (e.g. ordering of steps and joins).

Although such a proposal of merging the optimization and evaluation phases of a query falls into the category and can be compared to Adaptive Query Processing [12] techniques suggested in the relational case, it is the first in the context of XML and XQuery.

ROX executes the operations in the Join Graph one by one, fully materializing partial results. By doing so, the properties of already available intermediate data can be analyzed and used for determining the operators to execute next. This decision involves taking a small sample from the already available partial results and pre-executing different operators with that sample, taking note of both execution time and the properties of the result. In order to climb the hill and avoid a local optimum, we efficiently use deep "chain" sampling to examine multiple routes of consecutive operators until one route is designated as the most optimal for the next execution.

It is crucial to recognize, that ROX is not merely running a query optimizer at run-time, but in fact intertwines and integrates query optimization into the query evaluation procedure. This introduces the risk of materializing too large results as well as spending too much resources on the sampling-based optimization. Our extensive tests show that ROX not only consistently finds good plans from the XQuery search space even in the presence of correlated data, but also keeps sampling and materialization cost low.

Contributions. We view our contributions as *(i)* ROX is one of the very few techniques in the relational context and the first in XML that goes beyond simply moving query optimization to run-time to intertwining it with query evaluation. *(ii)* the chain-sampling technique that we propose provide the first generic and robust method to deal with any type of correlated data; *(iii)* the resulting ROX optimizer clearly improves the state-of-the art in XQuery optimizers both in plan quality as well as running time.

Outline. In Section 2 we introduce the basic building blocks for ROX: the Join Graphs, the used physical indexing and query evaluation algorithms, and our operator sampling method. Section 3 then describes in detail the ROX query optimization and evaluation algorithm. In Section 4 we extensively test ROX, showing that it can robustly find for queries with different degree of correlation a near-optimal plan, clearly beating a classical static query optimization strategy. Finally, in Section 5 we cover related work, then discuss future work in Section 6 before concluding in Section 7.

2. PRELIMINARIES

We now describe the foundations on which ROX builds: Join Graph Isolation, the physical data structures and algorithms used in the Join Graph evaluation, and the used sampling techniques.

2.1 Join Graphs

The main goal of ROX is to optimize at run-time the order in which the joins and XPath steps are executed. Therefore, an order-independent representation of all selection, join and step relationships needs to be conveyed to the run-time environment as part of the execution plan. We have chosen for an adapted form of a *Join Graph* as our order-independent representation, since Join Graphs have already been used in relational databases for similar purposes.

DEFINITION 1. *A Join Graph $G = (V, E)$ is defined as an edge labeled graph where:*

- a vertex $v \in V$ represents a relation of XML nodes which are input or output to join and path step operators of the query. A vertex v can be annotated with:

- *an element qualified name representing those XML element nodes with a certain qualified name,*
- *a text node with possibly a range-selection predicate denoting those text nodes with a certain value,*
- *an attribute node with possibly a range-selection predicate, denoting those attribute nodes with a certain value,*

Note that in principle, as Join Graphs can be surrounded by other parts of the query plan, some vertices in the graph could be pre-materialized input tables. For simplicity of presentation, we leave these out of our examples.

- an edge $e \in E$ represents a path step or join operator in the query. We distinguish between:

- *a step join, such as the staircase join described in Section 2.2, and*
- *a relational join, which according to the XQuery semantics computes a join using a value-based comparison of both inputs. Typically, the input vertices of such joins are text- or attribute-nodes.*

A Join Graph, input to the ROX algorithm, is obtained as follows. First, an initial relational query plan is generated from an XQuery using the relational compilation and peep-hole driven optimization described in [17]. This initial plan is statically optimized in such a way that specific types of operators are grouped together forming a Join Graph representation. The rewrite rules move numbering, distinct and sort operations out of the way (either downwards or upwards), creating a cluster of selection, projection, join and step operators. The boundaries of those sections are then detected and the clusters are replaced by corresponding Join Graphs. Doing so seems as easy said as done, but in reality requires a finely tuned set of optimization rules [18]. The execution plan with embedded Join Graphs is conveyed to the run-time environment of ROX for optimization, effectively deferring to run-time any decisions on the execution order of the joins and steps in the Join Graph. Occasionally, some operator constructs separating two groups of joins and steps can not be pushed below or above the clusters, resulting in a plan containing two isolated Join Graphs connected by the blocking operators. ROX will then optimize the different Join Graph sub-plans, and this way allows us to support the entire XQuery language while focusing on the optimization of the crucial order of joins and steps, including non-tree patterns often excluded [22].

The semantics ("result") of the Join Graph is a fully joined relation containing attributes of base relations. Subsequent projections in the plan specify which part of the fully joined relation we are interested in. Furthermore, the defined Join Graph does not guarantee the order and distinctness properties of the output as implied by the XQuery semantics. *Sort* and *Distinct* operators are needed to accomplish this. The static optimization rules have the effect that *Sort* and *Distinct* operators form a tail connected to the Join Graph. It may seem suboptimal to strictly separate the joins from these operators in the tail, however; it is possible, after identifying the Join Graph and during its run-time optimization, to push these operators, most crucially *Distinct*, between the joins. This is considered as an extra optimization step and is left as future work.

Example Figure 1 shows an execution plan with the embedded Join Graph of the following XQuery Q:

```
let $r := doc("auction.xml")
for $a in $r//open_auction[./reserve]/bidder//personref,
    $b in $r//person[.//education]
where $a/@person = $b/@id
return $a
```

The rectangle frames the Join Graph of the XQuery. The edges specify all step and join relationships between index-selectable node sets and attribute nodes. Without loss of generality, we limit ourselves to equi-joins as the most important representatives of a relational join. A relational join between two relations v_1 and v_2 is depicted as $v_1 \stackrel{=}{—} v_2$. A step join between two relations v_1 and v_2 is

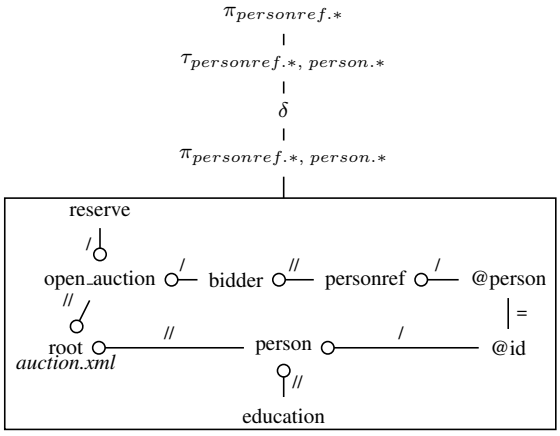


Figure 1: Corresponding Join Graph and tail of Q

depicted by an edge $v_1 \overset{ax}{\circ\!\!-\!\!-} v_2$ 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 is the context node sequence of the step join. Note that the direction is only a representational issue; the algorithm may very well decide to execute the step in the reverse direction. The semantics of the example query is a sequence of *personref* nodes where the order and duplicates are determined by both **for**-variables $a and $b bound respectively to *personref* and *person* elements. Therefore, the tail of the plan consists of a projection on the attributes corresponding to the *personref* and *person* elements, a distinct operator to remove duplicate (*personref*, *person*) node pairs, a numbering operation τ which determines the correct sort order of the resulting tuples (it sorts first by *personref* node identity, then by *person* node identity), and finally a projection to keep only the attributes corresponding to the *personref* nodes.

2.2 Algorithms and Index Structures

As ROX lives in a relational infrastructure, it applies to XQuery systems that use a relational database back-end. We use the open-source database system MonetDB/XQuery [4], that employs such a form of pure relational (tabular) storage of schema-free XML. That is, XML documents are shredded into tables, where information on every XML node is stored in a separate relational tuple. XML nodes are referred to using *node identifiers*, whose shape is in principle independent of the ROX approach (often-used incarnations are either from the family of variable-sized identifiers such as Dewey numbers [27] or range-based). MonetDB/XQuery uses the range-based pre/post relational node encoding, where pre, a virtual generated number reflecting the order of opening tags in the document, denotes the *node identifier* attribute in each tuple.

XML Extensions. The XQuery module of MonetDB provides a number of extensions of the relational algebra that are specific for XML. The most prolific of these are operators to shred XML into tabular form, to serialize tabular data in XML, and the *staircase* join [19] operator, a structural join which is defined as follows:

$$\overset{D_k/axis}{\leftthreetimes}(C,S) = \{[c,s] | c \in C, s \in S : \texttt{kind}(s.pre) = k \wedge s.pre \in axis(c.pre)\} = R,$$

with: $axis \in \{\texttt{anc}, \texttt{ancs}, \texttt{child}, \texttt{parent}, \texttt{desc}, \texttt{self}\ \texttt{descs}, \texttt{foll}, \texttt{folls}, \texttt{prec}, \texttt{precs}\}$,
$k \in \{\texttt{*}, \texttt{doc}, \texttt{elem}, \texttt{text}, \texttt{attr}, \texttt{comment}, \texttt{pi}\}$

The staircase join is a structural join which can process a single XPath step (*axis*::k) using as starting point the set of context nodes C. It takes as second input either the entire document $S = D_*$, or a kind restriction $S = D_k$ on return node kind k, or any subset $S \subset D_*$. It selects and returns all nodes in S that satisfy the relation (*axis*::k) with any node in C. The result R returned by the evaluation is a *set* of tuples (nodes), duplicate-free and in document order (i.e. sorted on *pre*). Though defined here as a set, the implementation of the staircase algorithms expect both inputs C and S to be tuple sequences sorted on pre.

Table 1 lists the relational operators used by ROX to process join graphs, and some of their properties such as cost. It shows that the staircase join is a highly efficient operator, in fact it can evaluate with linear complexity all types of XPath axes by making at most a single pass over the input S. Note that the ideas in ROX do not only apply to staircase joins – the collection of physical operators that ROX currently uses simply stems from the choice of MonetDB/XQuery as evaluation platform.

Auxiliary Operators. Additionally, the MonetDB/XQuery runtime module provides some relational procedures (similar to PL/SQL, these map into normal relational subqueries) that help probe the XML-specific indices. Various XML indexing structures have been proposed, such as element indices [6], data guides [15], and various kinds of structural synopses [14, 30].

Currently, in MonetDB/XQuery there is an element index and a value index that covers the values of all text and attribute nodes in the document. These indices can be seen as B-trees that store a series of *node identifiers* in index order, thus a range lookup comes down to determining the start and end boundaries of the selected range. As part of that action, the exact count of selected nodes already becomes available; therefore the cost of just counting the number of qualifying tuples is independent of the result size $|R|$ of

Relational Operators		**Cost**
$\overset{merge}{\bowtie}(C,S) \Rightarrow R$		$min(\|C\|, \|S\|) + \|R\|$
$\overset{hash}{\bowtie}(C,S) \Rightarrow R$		$\|C\| + \|S\| + \|R\|$
$\overset{nl}{\bowtie}(C,S) \Rightarrow R$		
$\times(C,S) \Rightarrow R$		(no sampling allowed)
$\overset{scan}{\sigma}(C,S) \Rightarrow R$		$\|C\|$
Structural Join	XPath predicate	**Cost**
$\overset{D_k/descs}{\leftthreetimes}(C,S) \Rightarrow R$	//k	$\|R\| + \|C\|$, iff $S = D$
$\overset{D_k/desc}{\leftthreetimes}(C,S) \Rightarrow R$	descendant::k	$\|R\| + log(max(\|C\|, \|S\|))*$
$\overset{D_k/child}{\leftthreetimes}(C,S) \Rightarrow R$	/k	$min(\|C\|, \|S\|)$, otherwise
$\overset{D_k/anc}{\leftthreetimes}(C,S) \Rightarrow R$	ancestor::k	$\|C\| * log(\|D\|)$
$\overset{D_k/ancs}{\leftthreetimes}(C,S) \Rightarrow R$	ancestor-or-self::k	
$\overset{D_k/foll}{\leftthreetimes}(C,S) \Rightarrow R$	following::k	$\|R\| + \|C\|$
$\overset{D_k/prec}{\leftthreetimes}(C,S) \Rightarrow R$	preceding::k	
$\overset{D_k/folls}{\leftthreetimes}(C,S) \Rightarrow R$	following-sibling::k	$\|C\|$
$\overset{D_k/precs}{\leftthreetimes}(C,S) \Rightarrow R$	preceding-sibling::k	
$\overset{D_k/par}{\leftthreetimes}(C,S) \Rightarrow R$	parent::k	
$\overset{D_k/self}{\leftthreetimes}(C) \Rightarrow R$	self::k	
Relational Sub-Queries for XML access		**Cost**
$\overset{D_{elt}}{\triangledown}(q_{elt}) \Rightarrow R$		$log(\|D_*\|) + \|R\|$
$\overset{D_{text}}{\triangledown}(v) \Rightarrow R$		
$\overset{D_{attr}}{\triangledown}(v, q_{elt}, q_{attr}) \Rightarrow R$		

Table 1: Physical Operators used by ROX

the index lookup. The result of an index lookup is a sequence of node identifiers (*pre*), duplicate free and in document order.

Given a qualified name q (and URI) of an element, the *element index* returns the list of all elements in document D satisfying q:

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

The basic idea of the *value index* in MonetDB/XQuery is an ordered store of $(val, q_{elt}, q_{attr}, pre)$ tuples. Such a structure can be used to find element-, text- and attribute-nodes using equi- or range-lookup on value. Depending on the node type, the lookup query can include a restriction condition on element name q_{elt} (for element and attribute nodes), and additionally a restriction on attribute name q_{attr} (for attribute nodes). In our experiments, we used the released version of MonetDB that 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 document D having a value v:

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

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

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

For each vertex in the Join Graph denoting a given element name, a text or attribute node with an equality predicate, an index lookup can be used to efficiently retrieve, as well as determine the count of, all qualifying matches. It is also possible, given a set of values, to probe the value index to evaluate any equi-join in the Join Graph.

2.3 Cut-Off Sampled Operators

The ROX algorithm intertwines operation execution with sampling-based cost estimation, where it uses a new *chain sampling* technique to explore multiple operations ahead in order to decide upon the next execution step. Operator sampling consists of executing the operator with a sample of its input. In case of joins, it corresponds to joining one of the operator's input to a sample drawn from the second input. The start samples used by ROX are either a synthetic single-tuple relation containing the root node of the document, or a set of tuples sampled from indices. Efficient and reliable sampling from indices, using techniques like *partial sum trees* is well-known [26], and is achieved in case of the element indices in MonetDB/XQuery. ROX then samples steps and joins by feeding the obtained start samples into these operators.

Join sampling has been studied extensively, however the goal in previous work usually was to obtain an unbiased sample of the join result [7, 21]. In contrast, we use sampling to learn about correlated relationships between tuple distributions in the joined relationships, by monitoring sample tuples as they flow through multiple sampled joins. Therefore, we are interested in an *input biased* sample, that has the property that the join hit ratio of each input tuple that makes it into the join result is accurately reflected in the join sample. While this may sound complicated, it in fact leads to a simple well-known sampling methodology, proposed in *index based join selectivity estimation* [29], which takes a sample of input tuples from the outer operand, and looks-up (efficiently, using an index) *all* matching tuples in the inner operand.

In order to ensure efficient sampling, ROX restricts join sampling to physical operators that have the "zero-investment" property with respect to the sampled input. The zero-investment property means that the complexity of the operator only depends on the cardinality of that input. This rules out any algorithm that, prior to producing results, makes an investment that is linear (or worse) with respect to any other inputs. This condition is a generalization of the "index-available" condition that is long known to simplify the issue of efficiently obtaining reliable join samples [29]. Equi-join algorithms that have this property are merge join (only applicable if inner input is ordered) and nested-loop index lookup. The latter operation typically applies in our Join Graph when there is an equality edge touching an attribute- or text-node (in which case the XML value indices are used). Similarly, a crucial feature of the staircase joins is that they also conform to the zero-investment property with regards to their input C, which makes them fully applicable in the ROX context. In fact, we observed that our Join Graphs can be fully executed in MonetDB/XQuery with zero-investment operators.

Join operations can in principle return the Cartesian product, i.e. their worst case complexity is quadratic. Though this certainly is not typically the case, our run-time query optimization should guarantee efficiency against high join hit ratios blowing up a result sample. Therefore, rather than producing the full result of a join sampling followed by reducing it, we prefer to do this in one step. In particular, we enforce the operation $OP(c, S)$ between sampled input c and relation S to have cost linear in the size τ of the sample. This is currently simulated by cutting off the result generation early. Consequently we need to observe the fraction f of c tuples that were processed at this point, in order to extrapolate the size of the full, unlimited, result r' as $|r'| = \frac{|r|}{f}$. A simple way of doing so is to make sure c carries a row-identifier, densely increasing from 1, which is propagated in the output and computing $f = \frac{max(r.rowid)}{max(c.rowid)}$.

For example, if we work with an average sample size $\tau = 1000$, and start with such a sample from relation A, and join that with the full relation B (with, say, a join hit ratio $h = 5$), rather than taking a uniform sample of 1000 from the 5000 hits, we prefer to conserve the full join result for around 200 A tuples, generating directly a result of 1000 tuples instead of 5000. Therefore to keep sample sizes within a usable range, our approach is thus to cut-off the generation of join results, always taking note of the used reduction factor f (e.g. *f=0.2* here) to properly scale our cost estimations.

The sampling operation is denoted by $\trianglerighteq_l$, and can take two types of input, a table T or an operator *OP*. The operation $\trianglerighteq_l(T)$ refers to picking from T a random sample of size l, while $\trianglerighteq_l(OP)$ represents a partial execution of the operator *OP* where the execution stops as soon as the number of generated tuples reaches the limit l. The output of $\trianglerighteq_l$ is a table containing the result of the sampling operation. In case of operator sampling, $\trianglerighteq_l(OP)$ will additionally return an estimation of the cardinality of the full execution of *OP* using the observed reduction factor f as described earlier. Note that the use of the limit l parameter in the sampling operation $\trianglerighteq_l$ introduces a statistical bias towards the tuples early in the sample. As an alternative, a non front-biased cut-off sampled join would observe the amount h_t of outer tuple hits on the current inner (sampled) tuple t, and after fully processing that outer tuple, would skip the h_t following outer tuples. This ensures that the join result stays in line with the input without front bias. In our current version of ROX, we accept the front bias risk and refrain from extending MonetDB/XQuery with such new physical cut-off (step) join operators, because our focus is on dynamic query optimization rather than on new sampling methods, and the experiments have proved that ROX performs well even in the presence of such a risk.

3. ALGORITHM

The join ordering problem boils down to analyzing the Join Graph in search of a (near-)optimal join order from the entire search space

of possible execution orders. ROX interleaves optimization and execution steps, exploring the search space by efficiently sampling path segments (sequences of step and join operators). As soon as a path segment is found to be superior to others, the sampling stops, the associated step and join operators are executed, their results are materialized, and the process of searching for the next superior path segment starts, benefiting from the newly obtained data and more accurate statistical knowledge.

We first define some needed notation. Given a Join Graph $G = (V, E)$, a vertex $v \in V$, and an edge $e \in E$

- $T(v)$ represents a table with all XML nodes satisfying v.
- $S(v)$ represents a table containing a random sample of XML nodes satisfying v.
- $card(v)$ is the estimated number of XML nodes satisfying v.
- $edges(v)$ represents all outgoing un-executed edges of v.
- $w(e)$ is the weight of e being an estimation of the cardinality of the result of the step or join operator associated to e.
- $exec(e, T_1, T_2)$ represents the result of the execution of the operator associated with edge e on input tables T_1 and T_2.

The main algorithm of the run-time optimizer is shown in Algorithm 1. It consists of two phases. The first phase initializes the Join Graph. The second phase alternates search space exploration and path segment execution until all edges have been executed.

Phase 1 (Algo1: lines 1-4). First for each vertex v, a sample of size τ of tuples satisfying the annotated name and range-predicates of v is materialized and the total number of satisfying tuples is estimated (lines 1-2). This is efficiently provided by an index lookup. To keep the cost of this operation low, it is restricted to vertices representing XML elements or text elements with an equality predicate condition. Unless specified otherwise, we use, throughout the algorithm, a default sample size of 100 ($\tau = 100$).

Second, for each edge $e = (v_1, v_2)$, a weight is computed by linearly extrapolating the result of sampling e (lines 3-4). We define

$$\textit{EstimateCard}(e) = \frac{card(v)}{\tau} \times est$$
$$\text{where } (R, est) = \trianglerighteq_\tau(exec(e, S(v), T(v')))$$
$$\text{and } (v, v') = \begin{cases} (v_1, v_2) & \text{if } card(v_1) < card(v_2) \\ (v_2, v_1) & \text{otherwise} \end{cases}$$

As described in Section 2.3, sampling e is done by picking a sample from the input table of one of e's vertices, followed by the cut-off execution of e's operator with the other vertex input table. We choose to use the smallest vertex as input for sampling, because a sample from a smaller table provides a more representative set of the data, leading to a more accurate estimation of the cardinality of the step or join result. An edge whose both vertices do not have a materialized sample $S(v)$, will stay unweighted for now. Since the table $T(v')$ is still undefined at this stage, the execution of e will instead be carried on the sample set $S(v')$.

Phase 2 (Algo1: lines 5-19). The second phase of the algorithm alternates between exploring the search space for a superior path segment and executing this path segment, until all operations in the Join Graph are executed. The Join Graph exploration is performed by the *ChainSample* function (line 6) described in detail in Section 3.1. It analyses candidate path segments in the graph to identify the one that is superior. All edges along the chosen path segment are then executed (line 7-13), and the knowledge in the Join Graph is updated (line 14-19). To execute an edge $e = (v_1, v_2)$, its input tables $T(v_1)$ and $T(v_2)$, if undefined yet, are initialized to the complete result of the corresponding index lookup (line 8-12). This operation is again restricted to vertices representing an

```
   INPUT    : Join Graph G = (V, E)
1  FOR each v ∈ V | v is an element with qname x or a text node with
   predicate "= x" DO
2  |  (S(v), card(v)) ← ▷=τ(▽(x));
3  FOR each e = (v1, v2) ∈ E | S(v1) ≠ null ∨ S(v2) ≠ null DO
4  |  w(e) = EstimateCard(e);
5  WHILE there are more edges to execute DO
6  |  Path p ← ChainSample();
7  |  FOR each edge e = (v1, v2) ∈ p DO
8  |  |  FOR v ∈ {v1, v2} DO
9  |  |  |  IF T(v) = null ∧ v is a root node THEN
10 |  |  |  |  T(v) ← pre
   |  |  |  |         1 ;
11 |  |  |  |  ELSE IF T(v) = null ∧ v is an element with qname
   |  |  |  |  x or a text node with predicate "= x" THEN
12 |  |  |  |  T(v) ←▽(x);
13 |  |  |  exec(e, T(v1), T(v2));
14 |  |  |  FOR v ∈ {v1, v2} DO
15 |  |  |  |  UpdateTable(v);
16 |  |  |  |  S(v) ← ▷=τ( T(v));
17 |  |  |  |  card(v) ← |T(v)|;
18 |  |  |  |  FOR each e ∈ edges(v) DO
19 |  |  |  |  |  w(e) = EstimateCard(e);
```

Algorithm 1: Run-time Optimizer

XML element or a text element with an equality predicate condition. After e is processed, $T(v_1)$ and $T(v_2)$ are updated to include only those tuples that satisfy the execution, and the sample set $S(v)$ and cardinality $card(v)$ of both v_1 and v_2 are updated accordingly (line 15-17). Consequently all remaining un-executed edges incident to v_1 and v_2 are sampled to compute their new weight (line 18-19). Note that this also happens for previously computed weights; these are thus re-sampled with a new sample taken from the updated tables. This is a crucial feature of ROX: simply adjusting the already computed weights by e.g. multiplying with the join hit ratio of the executed path would not be enough as this implies an independence assumption. By re-sampling, ROX is able to detect arbitrary correlations between edges in the Join Graph.

3.1 Chain Sampling

The heuristic used by ROX is to execute the edge whose intermediate result cardinality is smallest. Because the weight of an edge represents an estimate of its intermediate result cardinality, this seems a matter of simply choosing the edge with the smallest weight. This may be, however, only a *local minimum* in the search space due to correlations. The function *ChainSample* (see Algorithm 2) invests a small amount of time to climb the hill to be *sure* that another chain of operators (i.e., a path segment) would not produce an intermediary result with smaller cardinality.

The algorithm samples a chain by using the output of sampling one operator as input to the sampling of the next one. Since checking all possible paths in the graph is too expensive, the algorithm explores only those paths that branch from the edge with the smallest weight. By sampling ahead in the branches, the algorithm may discover that a branch due to correlations produces a result of much lower or higher cardinality than the estimations (weights) initially predicted, hence the branch proves superior or inferior to others.

Algo2: lines 2-5. Chain sampling first determines the edge e with the smallest weight. If both its vertices are not branching, no chain sampling is performed because all edges neighboring e are already executed. In this case *ChainSample* returns e for execution. Other-

```
   INPUT  : Join Graph G = (V, E)
   OUTPUT : Path p
1  e = (v1, v2) | e ∈ E ∧ w(e) = min_{e_i ∈ E} w(e_i);
2  IF |edges(v1)| > 1 ∨ |edges(v2)| > 1 THEN
3  |  source ← v | card(v) = min_{v_i ∈ {v1,v2}} card(v_i);
4  ELSE
5  |  RETURN {e};
6  Path p ← {};
7  StopVertex(p) ← source;
8  I(p) ← S(source);
9  paths.insert(p);
10 cutoff ← τ;
11 WHILE ∃ more edges to sample DO
12 |  cutoff ← cutoff + τ;
13 |  FOR each p ∈ paths DO
14 |  |  v ← StopVertex(p);
15 |  |  IF edges(v) > 0 THEN
16 |  |  |  paths.remove(p);
17 |  |  FOR each e = (v, v') ∈ edges(v) DO
18 |  |  |  Path p' ← p ∪ {e};
19 |  |  |  (I(p'), est) ← ▷_cutoff(exec(e, I(p), T(v')));
20 |  |  |  StopVertex(p') ← v';
21 |  |  |  cost(p') ← cost(p) + est * card(source) ÷ τ;
22 |  |  |  sf(p') ← est ÷ τ;
23 |  |  |  paths.insert(p');
24 |  FOR each p_i ∈ paths DO
25 |  |  FOR each p_j ∈ paths | i ≠ j DO
26 |  |  |  IF cost(p_i) + sf(p_i) * cost(p_j) ≤ cost(p_j) THEN
27 |  |  |  |  p = p_i;
28 |  |  |  ELSE
29 |  |  |  |  p = null;  break;
30 |  |  IF p ≠ null THEN
31 |  |  |  RETURN p;
32 FOR each p_i ∈ paths DO
33 |  FOR each p_j ∈ paths | i ≠ j DO
34 |  |  IF cost(p_i)+sf(p_i)*cost(p_j) ≤ cost(p_j)+sf(p_j)*cost(p_i)
   |  |  THEN
35 |  |  |  p = p_i;
36 |  |  ELSE
37 |  |  |  p = null;  break;
38 |  IF p ≠ null THEN
39 |  |  RETURN p;
```

Algorithm 2: Chain Sample

wise, it determines which of its vertices is the best starting point for exploring neighboring un-executed branches, by choosing the one with the smallest cardinality. We refer to this vertex as *source*.

Algo2: lines 6-37. The branches are explored in a breadth first manner starting from the *source* vertex. Each round consists of extending path segments (line 18) by sampling the next possible edge in every branch (line 19). Note that additional path segments may be created when other branching vertices are encountered.

Each path segment p is associated with a number of properties:

- *StopVertex*(p) is the vertex from which p's next sampling round will start (line 20). It is initialized to *source* (line 7).
- $I(p)$ is the sample set to use as input in the next sampling round of p. It is initialized with a sample of size τ from the *source* (line 8). In every subsequent round, it consists of the output table of the previous sampling operation (line 19).
- *cost*(p) is the estimated combined cardinality of all intermediate results of path segment p. Each time p is extended with an edge e, its cost is incremented with the estimated cardinality of e (line 21).
- *sf*(p) is the scale factor of p. It represents the join hit ratio ($\frac{output\_size}{input\_size}$) resulting from executing p (line 22).

After each sampling round, the optimizer updates the properties of the path segments (line 19-22), and compares them to check if one proves to be superior (lines 24-29) allowing for an interleaving with the execution of the superior path. The optimizer makes such a decision by comparing the *cost* and *sf* of all pairwise combinations of path segments using the following stopping condition (line 26):

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

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

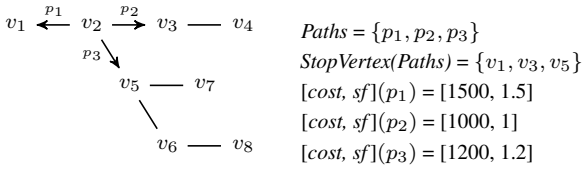
The idea behind the equation is that, given two paths p_i and p_j, if the execution of p_i followed by the execution of p_j is cheaper then executing p_j alone, we can safely execute p_i. For example, if *cost*(p_j) was estimated to be equal to 1000 and the execution of p_i will reduce the intermediate result by half (*i.e.* *sf*(p_i) $= 0.5$), then the cost of executing p_j after p_i is estimated to be equal to 500. If p_i happens to cost less than 500, thus satisfying the above condition, we can decide to stop chain sampling, because 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. Therefore, if the above condition holds, we stop chain sampling and interleave optimization with the execution of p_i. If the *stopping condition* is not satisfied at the end of a sampling round, a further round of sampling is initiated until either the stopping condition is satisfied or all branches are fully explored. In this case, the algorithm picks the best candidate path segment based on the equation in line 34.

We have previously noted that our simple cut-off technique used while sampling indices and step/join operators to limit the size of generated result, is biased towards the front input tuples. This can become a problem in chain sampling as the bias accumulates over subsequent executed operators. We now mitigate this problem, by incrementing the sampling limit (referred to as *cutoff*) with τ after each round (line 12). We recognize that this solution is not ideal, however, a better approach would require modifications inside the steps and joins of MonetDB/XQuery (see the end of Section 2.3).

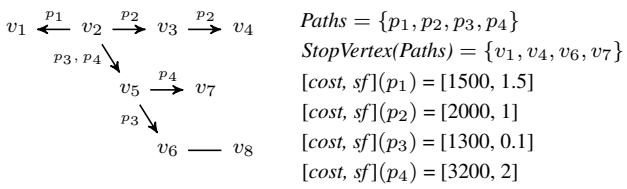
Example. We illustrate the chain sampling process with the example Join Graph shown in Figure 2.1. The edge with the smallest weight is (v_2, v_3). Suppose that the cardinality of v_2 is smaller, so we choose v_2 as the *source*. Sampled edges are indicated with an arrow and labeled with the path segment they belong to. In Figure 2.2, the stopping condition holds for $i = 3$ and $j = [1, 2, 4]$. Therefore, the algorithm stops chain sampling although there is one more edge (v_6, v_8) which can still be sampled. In this case, chain sampling was able to detect an existing selective correlation between the elements v_2, v_5 and v_6, and as a result the optimizer will execute the edges in path p_3 instead of executing edge (v_2, v_3) which was found earlier to be the best.

3.2 The Power of the Run-time Optimizer

In this section, we present an example that illustrates the behavior of our run-time query optimization algorithm on the following

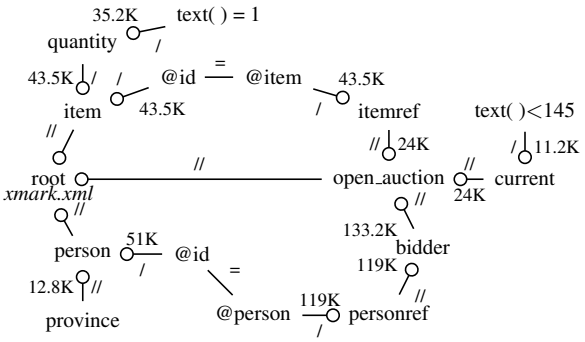


2.1 The first round of chain sampling using v_2 as *source*

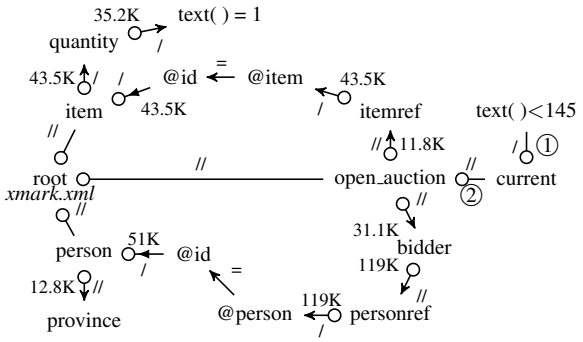


2.2 The second round of chain sampling

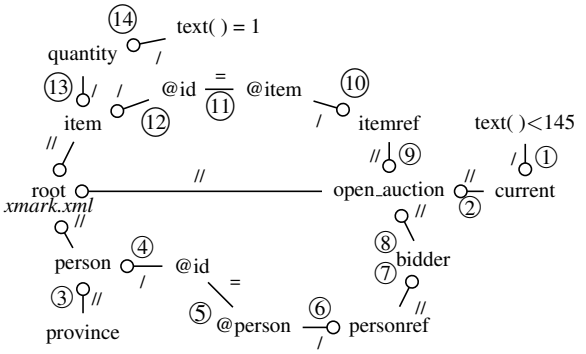
Figure 2: Illustration of Chain Sampling



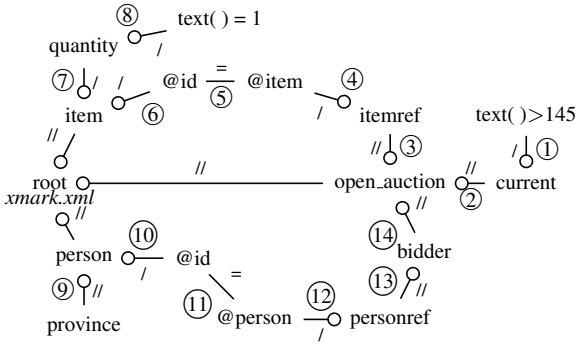
3.1 Join Graph of Q_1



3.2 The third exploration step of the run-time algorithm



3.3 The order of operators' executed by ROX for Q_1



3.4 The order of operators' executed by ROX for Q_1^m

Figure 3: XMark example

XQuery Q_1:

```
let $d := doc("xmark.xml")
for $o in $d//open_auction[.//current/text() < 145],
    $p in $d//person[.//province],
    $i  in $d//item[./quantity = 1]
where $o//bidder//personref/@person = $p/@id and
      $o//itemref/@item = $c/@id
return $a
```

Figure 3.1 shows the Join Graph of Q_1 where the number label on each edge represents the initial weight computed by sampling. The algorithm proceeds by picking the edge with the smallest weight *(current, text<145)*, and chain sampling. Figure 3.2 shows the Join Graph after two alternating exploration and execution steps. The circled numbers represent the order of execution. The weight of the edges *(open_auction, itemref)* and *(open_auction, bidder)* are recomputed using the new updated data in T*(open_auction)*, where *card(open_auction)* = 11.8K. The weights of other edges are unchanged. The same figure presents the third exploration step: the arrows on the edges indicate the chain sampling process using the vertex *open_auction* as *source*. Table 2(a) shows the *(cost, sf)* pair of p_1 and p_2 after each round of sampling, where p_1 is the path segment going though the vertex *bidder* and p_2 is the one traversing the vertex *itemref*. Six sampling rounds are performed, and the *stopping condition* after each iteration is never satisfied. At the end of chain sampling, we have the following:

$cost(p_1) + sf(p_1) * cost(p_2) = 154K + 0.5 * 70.2K = 189.1K$

$cost(p_2) + sf(p_2) * cost(p_1) = 70.2K + 0.94 * 154K = 214.96K$

which means that p_1 should be executed before p_2. The order of execution of all edges in the Join Graph is shown in Figure 3.3. Notice that the order of execution of the edges in path p_1 is not the same as the one followed during chain sampling. In fact, the algorithm will treat this path as a separate Join Graph, optimize it and execute its edges in the most optimal execution order found.

Suppose we modify the query by selecting the *current* elements satisfying the condition *text()*>145. We refer to this query as Q_1^m. The first two exploration and execution steps of Q_1^m are the same as Q_1. The updated data in the Join Graph is as follows: *card(open_auction)* = 12.1K, w*(open_auction, itemref)* = 12.1K, and w*(open_auction, bidder)* = 80.9K. In Table 2(b), we present the *(cost, sf)* pair of p_1 and p_2 estimated during the third chain sampling process. The decision of the chain sampling is, contrary to Q_1, to execute p_2 be-

(a) (b)

round	p_1	p_2	p_1	p_2
	current/text()< 145		current/text()> 145	
1	(29.6k, 2.5)	(11.8k, 1)	(83.7k, 6.89)	(12.1k, 1)
2	(59.2k, 2.5)	(23.7k, 1)	(167.4k, 6.89)	(24.3k, 1)
3	(88.8k, 2.5)	(35.5k, 1)	(251.1k, 6.89)	(36.4k, 1)
4	(118.5k, 2.5)	(47.4k, 1)	(334.8k, 6.89)	(48.6k, 1)
5	(148k, 2.5)	(59.2k, 1)	(418.5k, 6.89)	(60.7k, 1)
6	(154k, 0.5)	(70.2k, 0.94)	(438.2k, 1.6)	(72k, 0.94)

Table 2: value of $(cost, sf)$ of p_1 and p_2 after each round of chain sampling

fore p_1 as shown in Figure 3.4. Note that descendant edges from the root, in both queries, are ignored since these are not necessary to execute to produce the correct result.

As can be seen, the cardinality of *open_auction* elements in both Q_1 and Q_1^m after two execution steps are almost equal (11.8K and 12.1K respectively); however, the estimated number of descendant elements of type *bidder* in Q_1^m is much bigger (80.9K compared to 31.1K). This is logical and to be expected knowing that the bigger the current price of an item, the higher the number of bidders participating in the bid. A compile time optimizer, with the help of statistics, can estimate the cardinality of *open_auction* elements in the two different queries. But it will definitely miss detecting the correlation between the *current* element values and the number of *bidder* elements and therefore miss-estimate the cardinality of the step operator between the *open_auction* and *bidder* vertices resulting in an order of execution that is not optimal.

4. EXPERIMENTS

Given its public availability in open-source and the fact that its Pathfinder XQuery compiler can provide us with an isolated Join Graph [18] as input for ROX, we chose the "Jun2008" release of MonetDB/XQuery[1] as platform for our prototypical implementation of ROX. We implemented our ROX approach in Java; it extracts the Join Graphs that Pathfinder generates from an XQuery, and passes these to its runtime optimization and execution engine.

For all experiments presented here, we use a PC equipped with two 2 GHz dual-core AMD Opteron 270 processors, 8 GB RAM and a RAID-0 disk system. The machine is running 64-bit Fedora 8 (Linux 2.6.24). MonetDB/XQuery is configured with optimization enabled and compiled with GNU gcc 4.1.2.

4.1 Dataset and Sample Query

The XMark example in Section 3 is very suitable to illustrate the principles and potentials of ROX in particular concerning data correlations. However, it represents only one specific case with very limited potential for variation. The goal of our quantitative assessment of ROX is not only to demonstrate its benefits in such particular circumstances. Rather, we want to assess its stability and robustness in a large variety of different constellations.

To achieve this, we use for our experiments the DBLP XML dataset[2], and split it up into ∼4500 single XML documents, one for each journal and conference series covered by DBLP. On this dataset, we use the following XQuery template that asks for authors that have published in 4 different journals and/or conference series:

```
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 Join Graph for the above query template is depicted in Figure 4. The solid edges arise from the original Join Graph as extracted by the Pathfinder compiler from the XQuery query. The dotted lines denoting join equivalences are added by ROX to allow for more flexibility to find a (near-)optimal plan. During a chain sampling phase, ROX explores up to 15 different path segments with a length ranging between 2 and 4 edges.

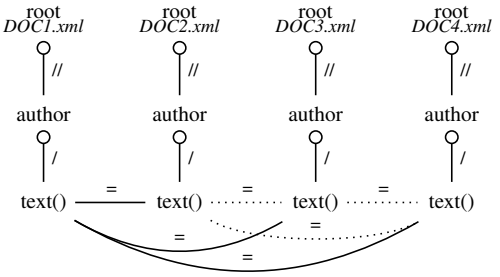


Figure 4: Join Graph of the DBLP Query

The idea is that by choosing the 4 documents from one or multiple research areas, we can vary the degree of correlation in the query. It is in general more likely that authors publish in various journals and/or conferences of one research area, than that an author publishes in multiple different research areas.

The original DBLP dataset consists of a ∼450 MB XML document covering ∼4500 journals and conference series, each ranging from 300 B to 4.8 MB in size. Even when choosing the 4 largest journals/conferences, ROX + MonetDB/XQuery manage to evaluate our query in less than 50 milliseconds. To achieve more reliable performance measurements, we scale the complete dataset to 4.5 GB, and 45 GB by replicating each article $n \in \{10, 100\}$ times, respectively. To avoid duplicates and to maintain the original data distribution and correlation, we suffix the titles and author names of each replicated article with a serial number from $[0, \ldots, n]$.

Since it is not possible to use all 4500 documents in our experiments, let alone all 409515972723000 combinations of 4 documents, we select 23 "representative" documents from 5 research areas with a total size of 15 MB (original) to 1.5 GB (scaled 100x). Table 3 lists the documents and some of their characteristics.

4.2 Query plans

In order to assess the quality of query plans generated by ROX, we implemented a small tool that enumerates all plans that ROX could potentially consider. The tool varies the order of equi-joins, the placement of location steps among the equi-joins, the direction of path steps, the use of indices, and the join- and step-algorithms and their implementations. In this way, it enumerates a total of 88880 different physical plans for our 4-way join DBLP query. Obviously, we cannot compare the ROX-generated plans to each of the 88880 alternatives. Hence, we introduce a two-level categorization of the plans. The first and most significant level is the equi-join order. For brevity, we use the term 'join' to refer to the equi-joins only. For our DBLP query, there are 18 different join orders as listed in the legend of Figure 5. We assume that individual joins are (logically/semantically) symmetric, but we distinguish linear and bushy plans. Parentheses in the join order nota-

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

journal / conference name	research area(s)[3]		# author tags × 1	# author tags × 100	document size × 1	document size × 100
Fuzzy Logic in AI	AI		62	6200	12 KB	1.2 MB
AI in Medicine	AI		2264	226400	332 KB	33 MB
AAAI	AI		6832	683200	1.1 MB	105 MB
CANS	AI	BI	214	21400	32 KB	3.1 MB
BMC Bioinform.		BI	3547	354700	440 KB	44 MB
Bioinformatics		BI	15019	1501900	2.1 MB	205 MB
BIOKDD	DM	BI	139	13900	22 KB	2.1 MB
MLDM	DM		575	57500	99 KB	9.9 MB
ICDM	DM		2205	220500	348 KB	35 MB
KDD	DM		3201	320100	460 KB	46 MB
WSDM	DM	IR	95	9500	13 KB	1.2 MB
INEX		IR	342	34200	54 KB	5.4 MB
SPIRE		IR	724	72400	124 KB	13 MB
TREC		IR	2541	254100	304 KB	31 MB
SIGIR		IR	4584	458400	811 KB	81 MB
ICME		IR	5757	575700	828 KB	83 MB
ICIP		IR	7935	793500	1.2 MB	113 MB
CIKM	DB	IR	3684	368400	629 KB	63 MB
ADBIS	DB		947	94700	294 KB	29 MB
EDBT	DB		1340	134000	389 KB	39 MB
SIGMOD	DB		5912	591200	1.8 MB	173 MB
ICDE	DB		6169	616900	1.7 MB	163 MB
VLDB	DB		6865	686500	2.1 MB	204 MB

Table 3: Research areas, documents and their characteristics

tion indicate precedence, and hence bushiness. The second categorization level is the placement of steps among the joins. In total, there are 804 different ways to place the `author/text()` steps among the joins. For our experiments, we limit our considerations to 3 canonical plans each exhibiting a specific step placement: SJ=$S_aS_bS_cS_dJ_aJ_bJ_cJ_d$ means that the steps for all 4 documents are executed before the joins in the same order of the joins execution; JS=$S_aJ_aJ_bJ_cJ_dS_bS_cS_d$ means that one step is executed first to provide the initial input for the join sequence, then all joins are evaluated, and the remaining 3 steps are executed last; S_J=$S_aJ_aJ_bS_bJ_cS_cJ_dS_d$ means that after the initial step and join, a step corresponding to a certain document is executed right after the document has been joined to the already generated intermediate result.

In addition to the enumerated and categorized plans, we also assume a "*classical*" compile time optimizer equipped with an accurate cardinality estimation module. This means that, in our DBLP example, the optimizer can correctly estimate the result size of an operator executed in the context of a single document, but will be unable to estimate the cardinality of operations joining two different documents. In the latter case, the optimizer falls back on a simple "smallest-input-first" heuristic to determine an appropriate join order. This results in a linear join order such that the two smallest sets of *author/text()* value are joined first, which is then joined with the second largest set, and finally joined with the largest one.

4.3 Performance for Different Plan Classes

Our first experiment is a simple demonstration how join orders influence (intermediate) join result sizes (and hence execution costs). We select 4 conferences — VLDB, ICDE, ICIP, & ADBIS; ICIP from IR, the others from DB — and calculate the sum of all (including intermediate) join result sizes for all join plans. Figure 5 shows the results for the × 100 scaled dataset. Due to the correlation among the 3 DB conferences, all join orders that consider the IR conference ICIP only at the end, need to process considerably (up to 3 orders of magnitude) larger intermediate data volumes than those join orders that start with ICIP. ROX manages to find the join order that creates the smallest size of intermediates, while the classical optimizer is not able to recognize and avoid the correlation.

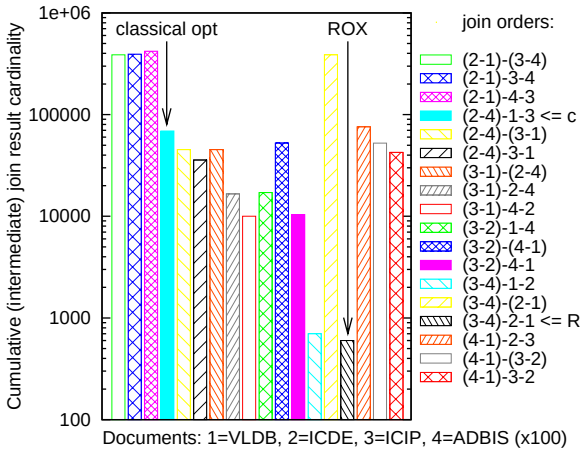


Figure 5: Impact of join order on intermediate result sizes

In our second experiment, we assess the elapsed execution time of different plans for a large variety of document combinations. We form the following 3 groups of document combinations: group 2:2 contains all combinations of 4 documents such that there are two pairs of documents from two different research areas; group 3:1 has 3 documents from the same area and one from a different area; group 4:0 has all documents from the same area. The idea is that these groups roughly cluster the document combinations according to their anticipated correlation. Omitting document combinations that yield empty results with our sample query, group 2:2 contains 469 combinations, group 3:1 contains 337 combinations, and group 4:0 contains 25 combinations. We calculate for each of the 831 combinations of 4 documents a "correlation" measure which represents the standard deviation of the join selectivity of all pairs of documents. For the document combination $D = \{d_1, d_2, d_3, d_4\}$, the correlation C is computed as follows:

$$\begin{aligned} \mathit{js}(d_i, d_j) &= \frac{|d_i \bowtie d_j| * 100}{\max\{|d_i|, |d_j|\}} \\ \mathit{mean} &= \mathrm{avg}\{\mathit{js}(d_i, d_j)\} \\ \mathit{diff}(d_i, d_j) &= (\mathit{js}(d_i, d_j) - \mathit{mean})^2 \\ C &= \mathrm{avg}\{\mathit{diff}(d_i, d_j)\} \\ &\text{for all } d_i, d_j \in D \wedge i < j \end{aligned}$$

For all 831 combinations, we identify 4 join-order classes to compare with: the join order that yields the *smallest* cumulative intermediate result size, the join order that yields the *largest* cumulative intermediate result size, the join order chosen by the classical optimizer, and the join order chosen by ROX. For each of these classes, we compare the performances of the SJ, JS, S_J "canonical" plans to the ROX plan. Note that the plans considered from the ROX join-order class are not the same as the ROX plan, because, although the executed join order is equal, the execution order of the XPath steps (and choice of step direction) follows the "canonical" forms, and hence is not optimized adaptively, as the "real" ROX does. Figure 6 presents the elapsed time for the different join-order classes and the ROX plan normalized to the fastest plan, for each of the 831 document combinations. In the plot, the document combinations are clustered by the considered area distributions 2:2, 3:1, 4:0, (separated by vertical dotted lines), and within each cluster ascendingly ordered according to their computed correlation value C. For each join-order class, except the *largest*, the normalized execution time of only the fastest of the SJ, JS, and S_J plans is plotted. For the *largest* join-order class, the time of slowest plan is drawn.

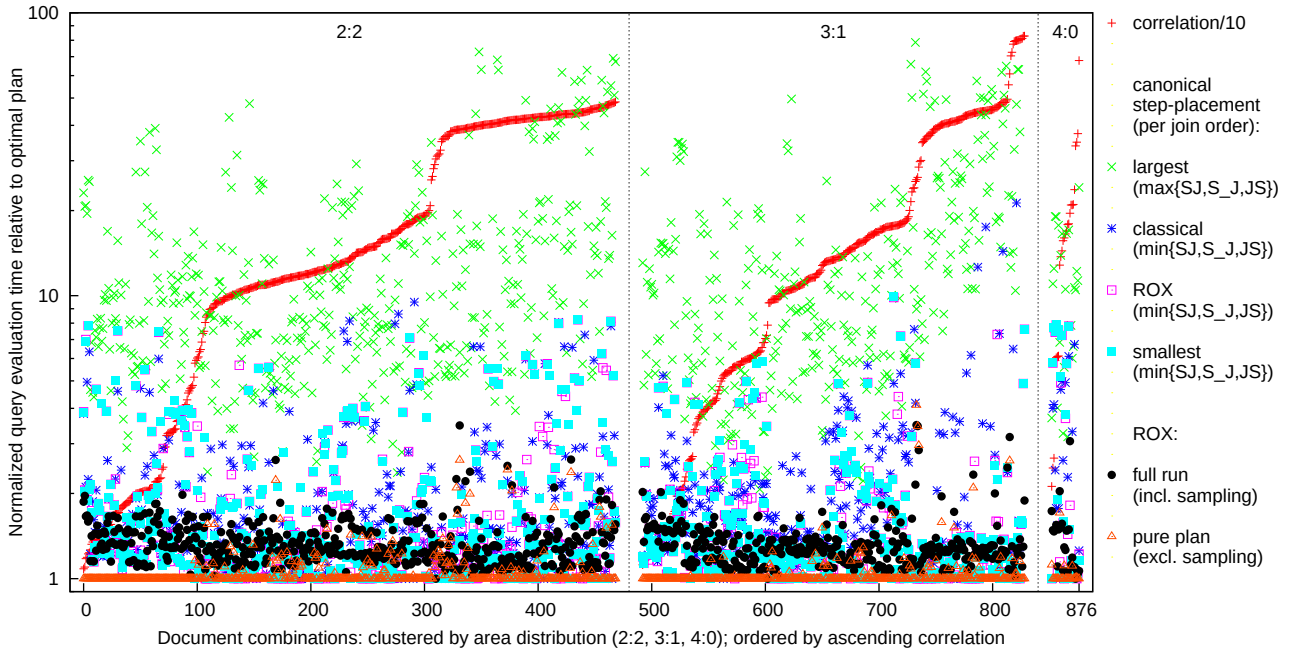


Figure 6: Elapsed Time of ROX vs Four Plan Classes

The reader of this plot should not try to examine specific cases but should observe the trend in the performance of the compared plans. The almost straight line of triangles at the bottom of the scatter plot shows that the plan found by ROX ("pure plan") is almost invariably the fastest plan. By comparing the circles with the triangles, that is, the ROX time including sampling with the pure plan, we also see that on average the overhead imposed by sampling is around 30%, and almost always lower than a factor two. Note that ROX behaves roughly the same across the various types of queries (2:2, 3:1 and 4:0) showing that it is *insensitive* to correlation. The classical plan on the other hand shows strong variation, it frequently exceeds the optimum by an order of magnitude or more, reaching even two orders of magnitude with high correlation in group 3:1. On average, the classical plan exceeds the ROX results by a factor 3.4 in group 2:2, factor 6 in group 3:1, and even factor 7.9 in group 4:0. The latter is rather unexpected, and obviously related to the unexpectedly high correlation in the 4:0 group. The ROX join-order plans (*i.e.* the right equi-join order, but "canonical" order of XPath steps), shown with empty squares, are in general very close to ROX, but for certain document combinations where apparently the XPath steps matter, the performance is much worse. Overall, our ROX optimizer counters the unexpected correlation effectively, demonstrating its robustness and reliability.

4.4 Scaling Document Sizes

While focusing on the $\times 100$ scaled dataset in the previous experiments, we now scale the dataset size to analyze the impact of the document sizes on the performance of ROX. The hypotheses is that the overhead of sampling might become more visible with less data, while the plan quality should stay the same.

Again, we compared the ROX full run plan (incl. sampling) and the ROX-generated plan (excl. sampling) to the plans corresponding to the *smallest*, *largest* and *classical* join-order classes on the same 831 document combinations. As before, the plan used in the experiment for both the *smallest* and *classical* join-order classes corresponds to the fastest canonical step placements (SJ, JS, S_J), while we use the slowest for the *largest* class. Figure 7 shows the relative cost of each of the 5 plans compared to the optimal plan on three dataset, the original DBLP set (scale $\times 1$), as well as scale $\times 10$ and scale $\times 100$. While the plain ROX generated plan is close to optimal, the full ROX run is almost twice as slow for small documents. With larger documents the sampling overhead shrinks considerably. In fact in the $\times 1$ dataset, the most expensive query executes in 50 milliseconds, therefore less time should be spent on optimization as almost any plan from the search space would run sufficiently fast. This adaptive decision on the time spent on optimization is considered as a future extension to ROX.

4.5 Impact of Sample Size

In our next experiment, we analyze the impact of the sample size on the sampling costs during the ROX query evaluation.

We run ROX on the 831 document combinations as before, using the $\times 100$ dataset and sample sizes of 25, 100, and 400 tuples. With R denoting the execution time of the full ROX run (incl. sampling) and r denoting the execution time excluding the sampling cost, we define the relative sampling overhead in % as $100 * (R-r)/r$. Figure 8 shows for each sampling size the average overhead per document group. As expected, the overhead increases with the sample size. The difference between 25 and 100 is marginal, while samples of 400 tuples cause significantly more overhead as samples of 100 tuples. This observation supports our initial intuitive choice of using sample size 100 in the previous experiments.

5. RELATED WORK

Accurate estimation of the cardinality of intermediate query results (for optimization purposes) has been extensively researched. Focusing on XML context, while several techniques have been developed [1, 8, 13, 14, 28, 30, 31], these still do not cover the full problem of XQuery intermediate result size estimation. These works all propose to build a synopsis and/or histogram for each XML document that captures the document structure and element

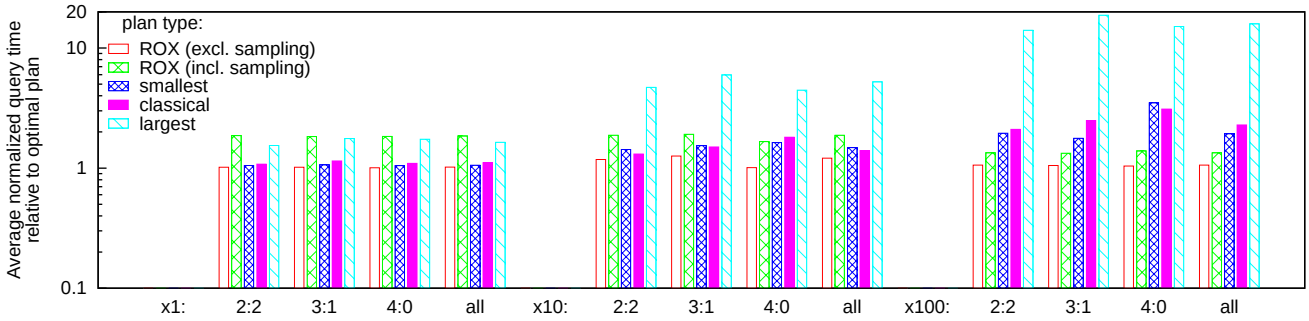


Figure 7: Scaling Document Sizes

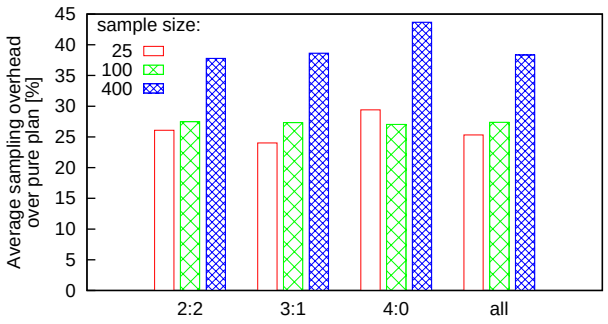


Figure 8: Impact of Sample Size τ on Sampling Overhead

values (in various forms). Some techniques cover the cardinality estimation of only a subset of the XPath language, others do not support queries with value constraints, and some can not efficiently handle updates to the document or recursive data. Moreover and generally speaking, cardinality estimation techniques, are based on the attribute value independence heuristic, which assumes independence between the values of different attributes and elements.

The independence assumption has originally been made in the relational domain, and overcoming it has become an active topic in query optimization research. The proposed solutions range from keeping and exploiting statistics over query expressions [5] or keeping table samples [3]. A related approach is to monitor query plan performance and use the resulting information as query optimizer *feedback* [20]. These efforts all fall under the umbrella of improving optimizer robustness. In this more general area, there have also been proposals to exploit cost model error distributions to direct the query optimizer not necessarily to the minimum cost plan, but instead look for a plan whose lower-bound performance (with the worst-case error) is minimum [3, 9].

The use of run-time techniques to mitigate the above problems has led to various proposals in the area of Adaptive Query Processing, where the general principle is that the query plan is determined, or can be changed, while the query is executing. A good survey of this area is [12], where each of the described techniques has its own limitations. For example, conditional plans [11] only protect against a priori known correlations, and can not detect them. Further, unlike ROX, those "switch-plans" can only cover a handful of alternatives. Another more popular proposal is Eddies [2], where operators in the query evaluation plan are reordered on a tuple-by-tuple basis. This is achieved by routing arriving tuples into the most efficient sequence of operators. Several tuple routing policies have been proposed, and each has its own shortcomings. Further, Eddies present two main drawbacks: they need to maintain query execution states which can become expensive, and they rely on only symmetric operators which makes them restrictive in the number of candidate plans considered for optimization.

Dynamic (also known as parametric) query evaluation optimizes the query at compile time into several candidate plans. Each such plan is optimal for a set of possible values that certain parameters can take at run-time. When at run-time the values of these parameters are known, the appropriate optimal plan is picked and executed. This technique was first proposed in [16] where a *choose-plan* operator is introduced to connect the candidate plans and choose the one to execute. One step further is to re-run the optimizer at run-time when the current plan, due to unexpected data selectivities or changing system resources, is no longer optimal. This Mid-Query Re-Optimization approach [24] triggers re-optimization when query execution is delayed by more than re-optimization time. A follow-up proposal [25] computes *validity ranges* which trigger re-optimization when the real observed cardinalities exceed these ranges. These approaches, however, take action depending on a comparison of observed with predicted cardinalities, with a cost model that does not necessarily anticipate correlations. ROX goes beyond these approaches by continually intertwining optimization and execution, and effectively basing all decisions on observed (sampled) cardinalities rather than a cost model, making it much less vulnerable to cost estimation errors and correlations.

6. FUTURE WORK

We finally sketch future directions to extend and enhance the ROX algorithm.

First, since ROX intertwines (sampling-based) query optimization work with query evaluation, it becomes possible to strike a balance between these two query evaluation cost factors. Static query optimization always runs the risk of spending too much time on optimization, such that it would have been faster to go with a maybe slightly worse plan that was found early, or spending too little time on optimization failing to avoid a very bad plan. A future adaptation of ROX can decide to invest more or less resources in chain sampling depending on the execution cost observed so far.

Second, while the algorithm described here only looks at the result sizes of the sampled operators, an alternative is to also measure the *execution time* of the sampled operators. To a limited extent, such functionality is already present in the current ROX prototype, which after deciding to execute an edge, tries all applicable physical operators on a sample to see which one is fastest. In XML query processing, depending on the document structure, there may be significant differences in running e.g. a `child` or `parent` staircase join. A future adaptation of ROX may use the actual execution time of a sample in the calculation of the weight of an edge, such that deciding which path segment to execute naturally takes into account many more characteristics of operator execution.

Third, while the current approach of fully materialized execution of edges fits the chosen MonetDB experimental framework, it does carry the risk of large and unnecessary intermediate result materialization, even though current RAM sizes allow materializing strategies on many of today's "large" problems. A possible extension of ROX could identify, while chain sampling, path segments that generate large intermediates, and run such sub-chains in a pipelined fashion, thus improving scalability. Another approach would be to run ROX with samples instead of the complete data, therefore materializing small sizes of intermediates. Preliminary experiments have proven this to be a promising idea.

Fourth, it is possible to incorporate into ROX the use of physical operators that do not conform to the "zero investment" policy (see Section 2.3). In fact, the algorithm does so, and at times executes an edge with materialized ends using e.g. a hash-join; however it currently does not sample such an operator in advance. By exploiting some of the more advanced proposed sampled (join) query processing techniques (e.g. [7]) it should become possible to extend the physical operator collection supported by ROX.

Finally, as mentioned in Section 2.1 we want to study efficient ways of integrating operators like Sorting, Distinct and Grouping into the Join Graph and the runtime optimization and evaluation environment of ROX. This extension would make the ROX algorithm applicable to areas such as SPARQL[4] and generic SQL processing.

7. CONCLUSION

In this paper, we have described ROX, a new XQuery processing approach that intertwines run-time query optimization with partial query evaluation. ROX uses a new adaptive *chain-sampling* mechanism which does not depend on a particular cost model. As such it does not suffer from the deficiencies of the current state-of-the-art in XQuery cost estimation, and additionally addresses a problem, also widely recognized in the relational context, of data correlations and their adverse affect on optimizer decisions. Further, the use of Join Graphs as part of a bigger execution plan gives ROX the possibility to handle a large class of XQuery queries.

We illustrate the power of the ROX optimizer both on XMark example queries as well as using extensive experimentation on the DBLP dataset. These latter experiments give us good indications that ROX sustainably achieves significantly better query execution plans than classical query optimizer approaches when data correlations come in to play, while its strict control over sampling efficiency keeps the optimization and sampling overhead low.

In future work, we plan to improve the ROX algorithm further by enhancing the sampling and indexing techniques on which it depends and making the algorithm even more robust to correlations and amendable to pipelining.

8. REFERENCES

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

[2] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[3] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: a Principled and Practical Approach. In *SIGMOD*, 2005.

[4] 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.

[4] `http://www.w3.org/TR/rdf-sparql-query/`

[5] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *SIGMOD*, 2002.

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

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *SIGMOD*, 1999.

[8] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *ICDE*, 2001.

[9] F. Chu, J. Halpern, and J. Gehrke. Least Expected Cost Query Optimization: What Can We Expect? In *PODS*, 2002.

[10] R. Cole and G. Graefe. Optimization of Dynamic Query Execution Plans. In *SIGMOD*, 1994.

[11] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting Correlated Attributes in Acquisitional Query Processing. In *ICDE*, 2005.

[12] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Found. Trends databases*, 1(1):1–140, 2007.

[13] D. Fisher and S. Maneth. Structural Selectivity Estimation for XML Documents. In *ICDE*, 2007.

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

[15] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[16] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *SIGMOD*, 1989.

[17] T. Grust. Purely Relational FLWORs. In *<XIME-P/>*, 2005.

[18] T. Grust, M. Mayr, and J. Rittinger. XQuery Join Graph Isolation. In *ICDE*, 2009. arXiv:0810.4809.

[19] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath Evaluation in Any RDBMS. *TODS*, 29(1), 2004.

[20] P. Haas, F. Hueske, and V. Markl. Detecting Attribute Dependencies from Query Feedback. In *VLDB*, 2007.

[21] P. Haas, J. Naughton, S. Seshadri, and A. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.*, 52(3):550–569, 1996.

[22] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. How To Recognize Different Kinds of Tree Patterns from Quite a Long Way Away. In *PLAN-X*, 2007.

[23] Y. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, 1991.

[24] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[25] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing Through Progressive Optimization. In *SIGMOD*, 2004.

[26] F. Olken and D. Rotem. Random Sampling from Databases - A Survey. *Statistics and Computing*, 5:25–42, 1995.

[27] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATH: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

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

[29] S. Seshradi. *Probabilistic Methods in Query Processing*. PhD thesis, Univ. Wisconsin, 1992.

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

[31] Y. Wu, J. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *EDBT*, 2002.