Run-time Optimization for Pipelined Systems

Riham Abdel Kader[1], Maurice van Keulen[1],
Peter Boncz[2], and Stefan Manegold[2]

[1] University of Twente, Enschede, The Netherlands,
`r.abdelkader@utwente.nl` `m.vankeulen@utwente.nl`
[2] CWI, Amsterdam, The Netherlands,
`P.Boncz@cwi.nl` `Stefan.Manegold@cwi.nl`

Abstract. Traditional optimizers fail to pick good execution plans, when faced with increasingly complex queries and large data sets. This failure is even more acute in the context of XQuery, due to the structured nature of the XML language. To overcome the vulnerabilities of traditional optimizers, we have previously proposed ROX, a Run-time Optimizer for XQueries, which interleaves optimization and execution of full tables. ROX has proved to be robust, even in the presence of strong correlations, but it has one limitation: it uses full materialization of intermediate results making it unsuitable for pipelined systems. Therefore, this paper proposes ROX-sampled, a variant of ROX, which executes small data samples, thus generating smaller intermediates. We conduct extensive experiments which proved that ROX-sampled is comparable to ROX in performance, and that it is still robust against correlations. The main benefit of ROX-sampled is that it allows the large number of pipelined databases to import the ROX idea into their optimization paradigm.

1 Introduction

The main role of a database optimizer is to explore the search space of execution plans and pick a good one in a small amount of time. For this, the traditional optimization paradigm relies on cardinality estimation techniques and cost models which should accurately estimate the size and the cost of operators. But these estimations are not always accurate. This is caused by, among others, missing or not up-to-date statistics [10], inability to capture data correlations [3], and assumptions that do not reflect real-life situations [5, 10] (*e.g.* attribute value independence). The innacuracy in cardinality and cost estimation propagates exponentially through the plan, possibly causing serious optimization errors [10].

With XQuery, the above problems are more acute. Due to the expressiveness and structured nature of XML, it is hard to build good cost models and concise synopses which accurately reflect both the structure and values in documents.

To overcome the shortcomings of traditional optimizers, we have previously proposed ROX, a Run-time Optimizer for XQueries [12]. ROX focuses on optimizing the execution order of the path steps and relational joins in an XQuery. It does so by interleaving optimization and execution steps, using sampling techniques to estimate cardinalities of operators. Each optimization phase initiates a sampling-based search to identify the sequence of operators most efficient to execute first. The execution step executes the chosen sequence of operators and materializes the result. This allows the subsequent optimization phase to analyze the newly materialized results to update the previously estimated cardinalities. Note that ROX also optimizes the execution direction of steps, that is, it decides if a step should be executed as a forward or a backward axis. By deferring optimization to run-time, ROX is able to acquire accurate knowledge about document characteristics and to detect existing correlations, without any need for statistics or a cost model. The experimental results presented in [12] have proved that ROX is robust in finding good execution plans, even in the presence of strong correlations, while keeping its run-time overhead limited.

It is the alternation of optimization and execution steps followed by the full materialization of results that gives ROX its robustness. But this also makes ROX unsuitable for the pipelined execution style adopted by most database systems. This paper proposes ROX-sampled, a new variant of ROX, that removes this limitation. In a pipelined system, an operator is executed in an iterative fashion, "piping" its output directly into the next operator. This means that only small chunks of data is processed and saved in memory at each iteration. In ROX-sampled, optimization and execution phases process only data samples resulting in less materialized data, making ROX suitable to pipelined systems.

Our proposed ROX-sampled approach raises some questions. Does the use of only small samples during the optimization and execution steps jeopardize its robustness? Will the small generated intermediates be representative enough to detect data correlations? As will become clear later in this paper, ROX-sampled needs, in some situations, to perform redundant operations. Will this reduce the efficiency of ROX-sampled? The paper addresses these questions and presents experiments that investigate the performance and efficiency of ROX-sampled.

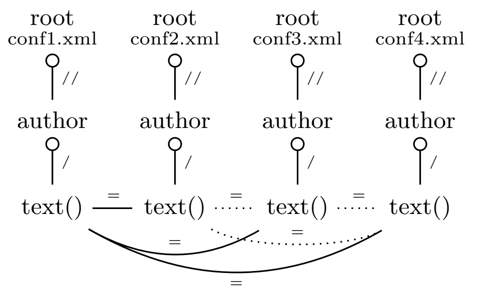
The contribution of this paper is the generalization of ROX to other pipelined execution styles, allowing the large number of pipelined databases to import the ROX idea into their optimization paradigm. The paper starts with preliminaries (Section 2) followed by a brief description of the original ROX, referred to hereafter as ROX-full (Section 3). Then ROX-sampled is explained, including the requirements that a pipelined system should support to efficiently run ROX (Section 4). Finally the conducted experiments are presented (Section 5).

2 Preliminaries

We now describe the foundations on which ROX builds: the join graph, the used physical algorithms and indexes, and the sampling techniques.

2.1 Join Graphs

ROX takes as input a join graph, which represents the to-be-ordered path steps and relational joins in XQuery, without any implications on their execution order. An input XQuery is first completely compiled into a DAG-shaped plan [2], which



```
for $a1 in doc("conf1.xml")//author,
    $a2 in doc("conf2.xml")//author,
    $a3 in doc("conf3.xml")//author,
    $a4 in doc("conf4.xml")//author,
where $a1/text() = $a2/text() and
      $a1/text() = $a3/text() and
      $a1/text() = $a4/text()
return $a1
```

Fig. 1. Join graph of the *4-way join* XQuery returning authors that have published in 4 different conferences.

is then statically optimized in such a way that XPath steps, joins, selections, and projections are grouped together forming a join graph [9]. An XQuery and its join graph are shown in Fig. 1. A vertex in a join graph represents a relation of XML elements, text, or attribute nodes, which is input and output to steps and joins. An edge specifies an XPath step or join relationships between two vertices. A step is depicted as $\circ\!\overset{ax}{—}$ where the label *ax* defines the axis of the step and the circle "∘" denotes the context set of the step. Note that this is only a representational issue; ROX may decide to execute the edge in the reverse direction. The edge $\overset{=}{—}$ depicts a relational join. The dotted edges in Fig. 1 represent join equivalences, and are added by ROX to broaden the search space, allowing more flexibility to find a good plan. The join graph extraction might fail to group all steps and joins in one cluster resulting in a plan containing several join graphs. ROX will then optimize each graph separately. In this paper, we only consider plans with a single join graph.

2.2 Operators and Index Structures

ROX is implemented on top of MonetDB/XQuery where XML documents are shredded into relational tables using a *pre*/*post* numbering scheme [2]. In addition to the standard relational operators, MonetDB/XQuery provides the *Staircase join*, a structural join capable of exploiting the tree properties of the *pre*/*post* plane to execute a single XPath step with linear complexity and at most a single sequential traversal over the XML document [8]. Additionally, MonetDB implements element- and value-indexes, which can fetch XML nodes with a certain qualified name, and text and attribute nodes satisfying a given predicate value. It is also possible, given a set of values, to probe the value indexes to evaluate equi-joins. In MonetDB, indexes are automatically built when loading the documents in the database. The complexity of an index lookup, as well as the cost of finding the count of qualifying tuples, is logarithmic to the index size.

2.3 Sample-based Operations

ROX uses sampling techniques to estimate the cardinality of vertices and edges in the join graph. To limit the time spent on sampling, only physical operators satisfying the *zero-investment* property should be sampled. These operators do not require any investment (*e.g.* sorting) prior to starting execution, and therefore their cost is linearly dependent on the size of the outer operand. All operators used in the ROX algorithm satisfy the *zero-investment* property.

To estimate the cardinality of a vertex in the join graph, the appropriate index is sampled. To estimate the size of an edge, the corresponding operator is sampled using an *index based join sampling* technique [14] 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. The output size of the sampling operation is then extrapolated to estimate the cardinality of the operator. This technique can be used for sampling staircase joins, and equality joins using the value indexes. Although the edge is executed with a sample, it can be an expensive operation if the join hit ratio is high. To avoid situations where large results are generated (the cartesian product in the worst case), the sampled execution of a given operator is stopped when the size of the generated output reaches a *cutoff* limit τ. Consequently, sampling needs to keep track of the number of tuples n of the sampled input S that contributed to its output r, to linearly extrapolate the size of the full sampling result R as $|R| = \frac{|S|}{|n|} * |r|$.

Definitions - Given an edge $e = (v, v')$, we define the following:

- The *weight* of e is an estimation of the size of the operator associated to e.
- $edges^*(v, v')$ is the set of all edges contained in the paths of executed edges branching from v, excluding the path starting with the edge (v, v').

We give an example of the second definition using the join graph in Fig. 1. We refer to the edges $(author^{(1)}, text()^{(1)})$, $(text()^{(1)}, text()^{(2)})$, $(text()^{(2)}, text()^{(3)})$, and $(text()^{(1)}, text()^{(4)})$ with respectively e_1, e_2, e_3, e_4. The superscript (i) denotes the document *conf*i.*xml*. If we suppose that the above 4 edges have already been executed, then $edges^*(text()^{(1)}, author^{(1)})$ is equal to $\{e_2, e_3, e_4\}$.

3 ROX-full: a Brief Description

This section briefly describes ROX-full, a complete presentation is given in [12]. The ROX-full algorithm consists of an initialization phase (Phase 1) followed by a phase where optimization and execution steps are alternated (Phase 2).

Phase 1: This phase initializes the join graph by picking the first samples, and estimating the cardinality of its vertices and edges. For a given vertex v, built indexes are sampled to estimate the number of tuples corresponding to v, and to retrieve a sample of these tuples. The weight of an edge e is computed by first sampling the edge e, and then linearly extrapolating the estimated output size. The input to the sampling operation is chosen from the vertex of e that has the smallest cardinality. In fact, picking the input sample from the smaller table provides a more representative set of the data, leading to a more accurate estimation of the weight of e.

Phase 2: This phase is the core of the ROX algorithm where optimization and execution steps are iterated. During optimization, a search for a *superior* path in the join graph is initiated. The search begins from the edge e with the smallest weight. Although e is the most selective edge, ROX does not proceed with executing it immediately. Instead *chain sampling* is used to search for a potential sequence of operators that is more selective than e. This can be compared to hill-climbing: ROX invests a small amount of time exploring the surrounding of e to avoid the execution of a *local minimum*. Note that if the vertices of edge e do not have branching unexecuted edges, no chain sampling will take place and the edge will be directly executed.

Chain sampling consists of exploring in a breadth first manner the paths of unexecuted edges branching from e, sampling iteratively one edge in each path. By consecutively sampling edges in one path, using the output of one sampling operation as input to the next, it is possible to detect correlations between the joined vertices. The input to the sampling operation of the first edge in each path is picked from the vertex v of e that has the smallest cardinality. At the end of each sampling iteration, a stopping condition searches for a path that is highly selective compared to the others. If such a path exists, chain sampling is stopped and the path is returned for execution. Otherwise and when all the edges in the paths branching from e are sampled, another condition checks for the most selective path and returns it for execution.

The execution phase evaluates the edges in the chosen path, using full tables as input, and materializes the result. Consequently the data in the vertices of the join graph are updated, and the weights of edges are recomputed using the newly materialized data. Optimization and execution steps are alternated until all edges in the join graph are executed.

ROX-full, the first XQuery optimizer that interleaves optimization and execution, proved to be a robust optimizer that improves the state-of-art in XQuery optimization. It does not depend on statistics nor a cost model, and chooses good execution plans even in the presence of strong data correlations while keeping its run-time overhead limited. ROX can handle a large class of the XQuery language by optimizing the join graph as part of a bigger execution plan. Although proposed in the XQuery context and implemented on top of MonetDB/XQuery, the ROX idea can be generalized to other systems and query languages, especially SPARQL in which a large number of self-joins are expressed.

4 ROX-sampled

We now introduce ROX-sampled, a variant of ROX, which makes the ROX idea also suitable for pipelined systems. ROX-full is *not* suitable for pipelined systems because its execution steps process operators with full tables, hence generating large intermediates. Therefore, the execution phases in ROX-sampled does not manipulate full tables. In fact, ROX-sampled follows the same steps as ROX-full, with main difference that only data samples are used throughout the whole algorithm. Therefore unlike ROX-full, where the optimization phase works with

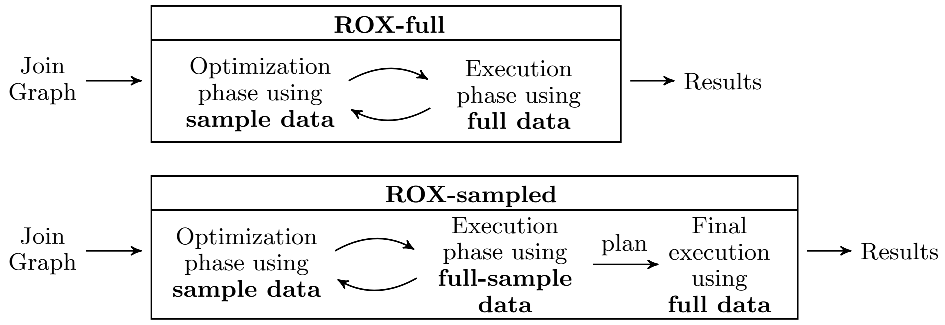

ROX-full
Join Graph
Optimization phase using sample data
Execution phase using full data
Results
ROX-sampled
Join Graph
Optimization phase using sample data
Execution phase using full-sample data
plan
Final execution using full data
Results

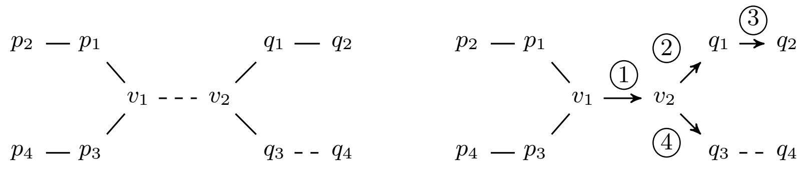
Fig. 2. An illustration of the steps of ROX-full and ROX-sampled.

a data sample while the execution phase processes full tables, both the optimization and execution phases of ROX-sampled manipulate data samples (Fig. 2). This means that the execution phases of ROX-sampled consist of sampling edges instead of executing them with full tables. Note that we use the term *executing* e to refer to the process of sampling e during an execution phase. Although both optimization and execution phases of ROX-sampled consist of sampling operations, each phase might use a different cutoff limit τ in their sampling; execution steps might specify a larger cutoff to allow for more results to be generated.

In ROX-full, two types of relations are associated to a vertex v: the full table $FT(v)$ which contains the XML nodes corresponding to v and which is used as input and output to execution operations, and a sample table $S(v)$ chosen randomly from $FT(v)$ and used in the sampling operations. In ROX-sampled, three relations are associated to a vertex v: the full table $FT(v)$ which contains the XML nodes corresponding to v, a *full-sample* table $FS(v)$ whose content is initially a random sample of tuples picked from $FT(v)$ and afterwards is input and output to execution operations, and a sample table $S(v)$ chosen randomly from $FS(v)$ and used in the sampling operations. Full-samples in ROX-sampled have the same role full tables have in ROX-full; however, they are of a much smaller size. The content of full tables in ROX-sampled is never changed, while, after each execution step, the content of full-sample tables is updated to the result of the processed steps and joins. Each decision made by an optimization phase is executed with the full-sample tables, and saved as part of a final execution plan. When ROX-sampled terminates, the saved plan is executed using the full tables. By limiting the amount of data accessed from base tables, processed and materialized at every execution phase, it becomes possible to apply ROX to pipelined systems.

4.1 Edges with Executed Vertices

An issue arises when executing or sampling an edge e with two executed vertices $e = (v_1, v_2)$. A vertex v is an executed vertex if at least one of its edges is executed. The problem is that a join between two sample sets from v_1 and v_2 does not result in a random sample of the output of the join between v_1 and v_2.



(a) Join graph before the execution of edge $e = (v_1, v_2)$.

(b) The join graph illustrating the execution of the edge $e = (v_1, v_2)$.

Fig. 3. The problem of executing or sampling an edge with two executed vertices.

More precisely, $S(R_1) \bowtie S(R_2) \neq S(R_1 \bowtie R_2)$. We stress that our goal is both to estimate the size of a join or step and to create a representative sample of the operator's output, which results in reliable cardinalities and outputs when used in further evaluations. Next, a solution to the problem is presented for the case of executing e. The same solution will be used for the sampling case.

Again, the problem is that $FS(v_1) \bowtie FS(v_2)$ does not result in a good sample of the join between v_1 and v_2. The solution we propose is *not* to use the *full-sample* of one of the vertices, but instead use the *full table*, *i.e.* $FS(v_1) \bowtie FT(v_2)$. This operation will match the tuples in $FS(v_1)$ with all XML nodes in the document corresponding to v_2. However, $FS(v_2)$ contains the result of all joins and steps that were already executed between v_2 and other vertices. Therefore, to correctly reflect those previous executions, the output of the join between $FS(v_1)$ and $FT(v_2)$ should be used as input to re-evaluate all the executed edges branching from v_2. The solution is illustrated in the join graph of Fig. 3(a) where solid and dashed lines represent respectively executed and non executed edges. First edge e is executed using as input $FS(v_1)$ and $FT(v_2)$, then the result is input to the join with $FT(q_1)$, and so on until all edges in $edges^*(v_2, v_1)$ are re-executed. The execution order is depicted in Fig. 3(b) as labels on edges while the arrows indicate the execution direction (*i.e.* the vertex from which the full-sample data is used as input for the execution). The decision to execute $FS(v_1) \bowtie FT(v_2)$ instead of $FS(v_2) \bowtie FT(v_1)$ aims at reducing the number of redundant evaluations, and stems from the fact that $|edges^*(v_2, v_1)| < |edges^*(v_1, v_2)|$.

For the sampling case of e, the same procedure is applied, but instead of using $FS(v_1)$ as input it uses $S(v_1)$. The goal of sampling e during an optimization phase is to also estimate its cardinality. Therefore, the proposed solution keeps track of the join hit ratio of all the sampled operators, to derive an estimation of the size of e. We omit the details due to lack of space.

4.2 Running ROX-sampled in Other Systems

Now that we have explained ROX-sampled and the MonetDB operators and data structures it uses, we briefly discuss the requirements to run ROX-sampled in other systems. We will focus on two points: picking the initial samples for each vertex, and the sampling of joins.

To pick the initial samples of XML nodes, ROX-sampled uses index lookups. Another method is to have the samples pre-built and saved in the database. This is comparable to collecting statistics, but instead of storing the data characteristics about each attribute, a representative sample of the attribute's values is saved. A good survey about sampling techniques is [6].

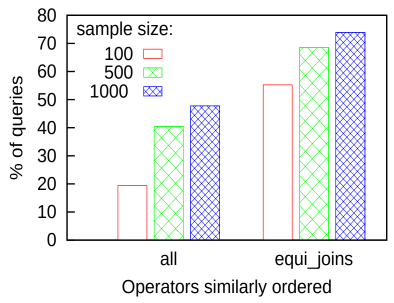
The sampling of edges is performed using an index-based join between a sample and a full table. This requires the existence of an index on one of the joined attributes. Techniques that efficiently sample a join without the use of an index have been proposed in [4]; however, they require the existence of statistics, a requirement we do not want ROX to depend on. Therefore if an index on the joined attribute is not available, a hash-based join can be used. But this means that hash tables must be built on both the input sample and the entire relation. The first is quite cheap. Hashing the entire relation is expensive, but can be a cheap operation if it is used when the join is executed with the full data; however this is not guaranteed to happen as it depends on the generated plan. Note that the hash table will be used in subsequent sampling operations which amortizes the cost of building it. We defer a study of the impact of using hash-based joins on the performance of ROX to later work.

5 Experiments

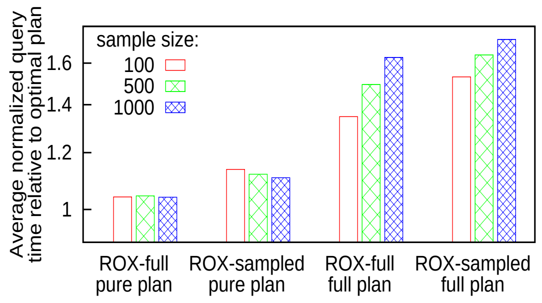
A prototype of ROX is implemented on top of the "Jun2008" release of MonetDB/XQuery[1]. Pathfinder, the XQuery processor of MonetDB [2], generates the isolated join graph for a given XQuery and provides it as input to ROX. For all experiments presented here, we use a PC equipped with two 2 GHz dual-core AMD Opteron 270 processors, 8 GB RAM, running 64-bit Fedora 8.

The experiments use the DBLP XML dataset[2], and the *4-way join* XQuery template shown in Fig. 1. The DBLP document is split up into ∼4500 single XML documents, one for each journal and conference. By replacing the 4 documents in the XQuery by 4 journal and/or conferences chosen from the same or different research areas, ROX-sampled will be tested against queries with different degrees of correlation. 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 research areas. We cluster the document combinations, according to their anticipated correlation, into 3 groups: group 2:2, group 3:1, group 4:0. A group *x:y* contains all combinations of 4 documents such that x number of documents are chosen from the same research area and y number of documents are picked from a different area. Since it is not possible to use all 4500 documents in our experiments, we select 23 "representative" documents from 5 research areas (Database, Data mining, Information retrieval, Bioinformatics, Artificial Intelligence), which results in 831 document combinations. The size of the documents extracted from the original DBLP document ranges from 300 B to 4.8 MB. ROX + MonetDB/XQuery evaluate these queries in less than 50 milliseconds. To achieve more reliable performance measurements, we scale the complete dataset to 45 GB by replicating each article 100 times.

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



(a) Plan comparison.



(b) Query time comparison.

Fig. 4. Comparison between ROX-full and ROX-sampled.

Execution order of operators: Our first experiment runs ROX-full and ROX-sampled on the 831 document combinations using 3 different sample sizes $\tau = \{100, 500, 1000\}$, and compares the chosen execution order of operators. Fig. 4(a) shows the percentage of queries optimized to the same plan by the two ROX. With a sample size equal to 100, only 20% of the plans are similar. This number increases to 48% when a sample size of 1000 is used. We also report the percentage of plans in which equi-joins are ordered similarly. This is of interest since the correlations between the 4 queried documents is detected through the estimated size of the equi-joins. Therefore when the two variants order the equi-joins similarly, it means that they detect and handle the correlations in the same manner. The percentage of plans with the same order of equi-joins grows from 55% to 73% when the sample size increases from 100 to 1000. We conclude that an increase in the sample size reduces the differences between the ROX variants. With a sample size equal to 1000, ROX-sampled is comparable to ROX-full in detecting correlations, and differs mainly in ordering the step operators.

Execution time of plans: The second experiment compares the execution time of the plans generated by the two ROX variants. Fig. 4(b) shows the average normalized execution time relative to the fastest time. For each variant, we time the chosen plan (pure plan), and the full-run which includes the sampling overhead. The execution time of the pure plan of ROX-sampled decreases when a bigger sample size is used. With a sample size of 100, the execution time of ROX-sampled is on average 9% longer than ROX-full, and it decreases to 6% when a sample size of 1000 is used. The execution time of the full run plans increases when a larger sample is used. We note that ROX-sampled has a higher sampling overhead than ROX-full. This is expected since, the time spent in the execution steps of ROX-sampled and to re-execute and resample some edges contributes to the optimization overhead.
Fig. 5 shows the normalized execution times of the four plans. The symbols (+) and (-) denote respectively full run plans and pure plans. We also consider the plan that a classical compile time optimizer would generate. The optimizer is able to accurately estimate the cardinality of operations carried on a single document, but lacks the ability to estimate the correlations existing among several documents. This results in an order of joins that reflects a smallest-input-first

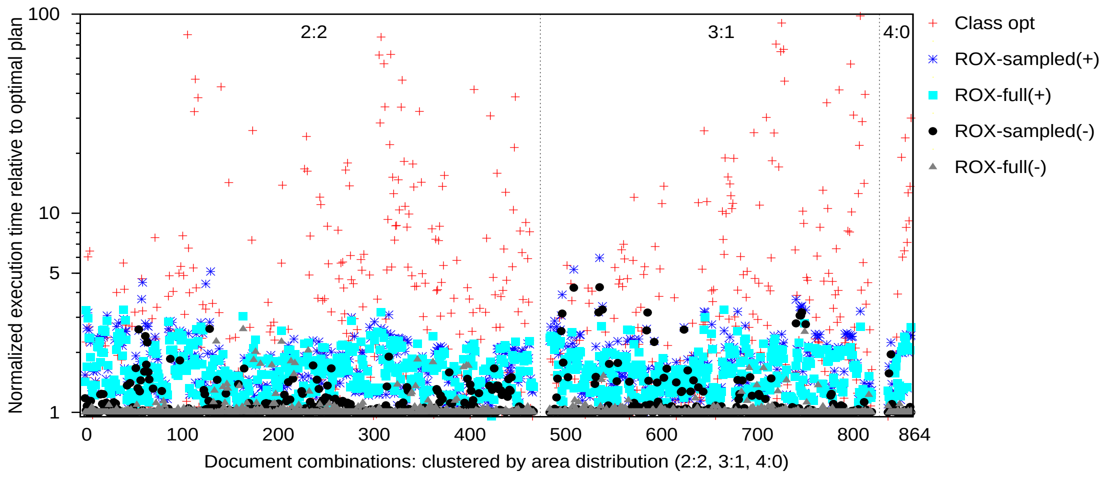


Fig. 5. Execution time of the plans chosen by ROX-full and ROX-samples ($\tau = 1000$).

heuristic where the two smallest inputs are joined first, which is then joined with the third largest input, and so on. The pure plan of ROX-full is the fastest almost all the time, except for very few queries. This is caused by the use of non representative samples during chain sampling which leads to bad execution decisions. A way to solve this is by detecting the error during execution and restarting the optimization phase. ROX-sampled is close to ROX-full, but for very few queries, it can be 3 times slower. The sampling overhead (full run) is on average around 30%. This plot shows that both ROX variants are robust and insensitive to the different correlations, while the classical optimizer shows strong variations.

Impact of the cutoff limit: In our last experiment, we vary the cutoff limit used during the execution steps of ROX-sampled. As explained in Section 4, the optimization and execution steps of ROX-sampled might use a different cutoff limit for their sampling operations. In our previous experiments, sampling during execution steps was performed with an unlimited cutoff limit: all tuples in the sample input were consumed by the sampling operation. In this experiment, the cutoff limit is set to the double of the sample size. The cutoff limit used during the optimization steps in the current and previous experiments is equal to the sample size. Fig. 6 shows the average normalized execution time of ROX-full, ROX-sampled, and ROX-sampled using a cutoff. We notice that the use of a small cutoff results in a small increase in the execution times of ROX-sampled, while the use of a cutoff limit of 2000 does not. Therefore, it is possible to use an appropriate cutoff limit in ROX-sampled without affecting its performance.

ROX-sampled has also been evaluated against XMark documents[3], and proved to be successful in picking a good execution order for the operators in the join graph. One XQuery, containing 15 XPath steps and 2 equality joins, is interesting

[3] http://www.xml-benchmark.org/

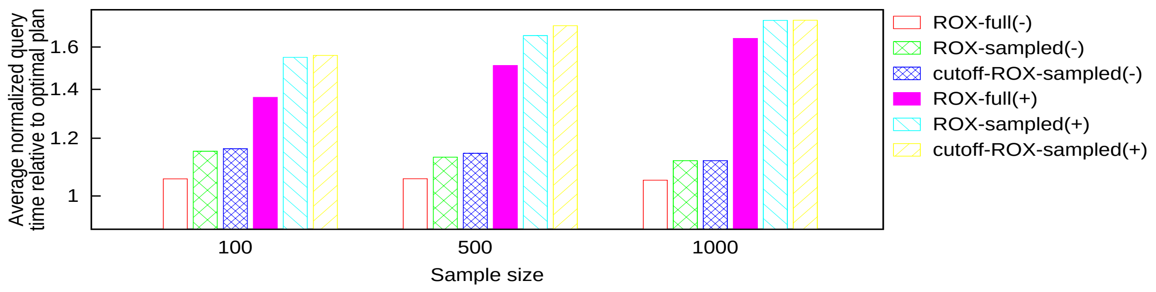


Fig. 6. Impact of the cutoff limit on performance.

to mention with greater detail (the query can be found in [12]). In this query, a correlation exists between the 3 elements: *open_auction*, *current*, *bidder*. Different values in the predicate condition assigned to the *current* element result in a higher or lower cardinality for the other 2 attributes. ROX-sampled proved to be capable of detecting the correlation and its changing effects, and to exploit it in determining the execution order of the operators in the join graph.

6 Related Work

Adaptive query processing has been researched during the last few years. Parametric query optimization [7] generates at compile-time several plans each optimal for a partition of the parameters domain. When at run-time the value of these parameters is known, the appropriate plan is executed. Query re-optimization [11] re-optimizes the plan if during execution, the observed costs differ from the estimates made during optimization. The work in [13] complements the above approach by embedding in the plan *validity ranges* which define the bounds of the estimated values for which the plan is valid. If the observed cardinalities fall outside these bounds, the plan is re-optimized. Other techniques [15] have a feedback loop which adjusts the statistics and cost functions in the database based on observation made during the plan's execution; however their learning curve can be long. The quality of plan chosen by the above three classes of techniques still highly depends on the accuracy of statistics and cost models. They have a reactive behavior and can not detect early enough selective correlations which can speed up performance. On the contrary, ROX is a proactive optimizer which does not depend on any statistics or cost model, and can detect and exploit correlations during optimization. We note that ROX-sampled can use re-optimization techniques similar to [11, 13], if, during the execution of the chosen plan with full tables, the observed cardinalities differ from the cardinalities estimated by the sampling operations.

Eddies [1], a routing based technique, do not depend on statistics or a cost model. They route each tuple to the most efficient sequence of operators based on observed properties. Eddies need to maintain query execution states which can become expensive. They also require the presence of symmetric operators which restricts the number of alternative plans they consider.

7 Conclusion

In this paper, we described ROX-sampled which generalizes the ROX idea to pipelined systems. ROX-sampled is a proactive optimizer which does not depend on statistics nor a cost model, and is robust in face of correlations. We also discussed the requirement to run ROX-sampled on other systems. Extensive experiments were conducted and showed that the performance of ROX-sampled is close to that of ROX-full, especially with larger sample sizes.

As future work, we plan to make ROX dynamic with respect to the time it spends on optimization, *i.e.* able to balance between the sampling overhead and the estimated execution time of the query. Currently, the execution decisions in ROX are based on operators' cardinality. A future extension to ROX would also take into account the execution time of operators. Finally, we want to study efficient ways of integrating operators like Sorting, Distinct and Grouping into the join graph and the optimization and evaluation environment of ROX.

References

1. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.
2. 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.
3. S. Chaudhuri. Query optimizers: Time to rethink the contract? In *SIGMOD*, 2009.
4. S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. *SIGMOD Record*, 1999.
5. S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 1984.
6. O. Frank and R. Doron. Random sampling from databases - a survey. *Statistics and Computing*, 1994.
7. G. Graefe and K. Ward. Dynamic query evaluation plans. *SIGMOD Record*, 1989.
8. T. Grust, M. Van Keulen, and J. Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. on Database Syst.*, 2004.
9. T. Grust, M. Mayr, and J. Rittinger. Xquery join graph isolation: Celebrating 30+ years of xquery processing technology. In *ICDE*, 2009.
10. Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Record*, 1991.
11. N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Record*, 1998.
12. R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. Rox: Run-time optimization of XQueries. In *SIGMOD*, 2009.
13. V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, 2004.
14. F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.
15. M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, 2001.