ROX: Run-Time Optimization of XQueries

Riham Abdel Kader

PhD dissertation committee

Chairman and Secretary

Prof. dr. ir. A. J. Mouthaan	University of Twente

Promotor

Prof. dr. P. M. G. Apers	University of Twente

Assistant Promotor

Dr. ir. M. van Keulen	University of Twente

Members

Prof. dr. J. C. van de Pol	University of Twente
Dr. ir. R. A. de By	University of Twente
Prof. dr. A. K. Elmagarmid	Purdue University, USA
Prof. dr. M. L. Kersten	CWI, The Netherlands
Prof. dr. T. Grust	Universität Tübingen, Germany

CTIT Dissertation Series No. 10-184
Centre for Telematics and Information Technology (CTIT)
P.O. Box 217 - 7500 AE Enschede - The Netherlands

SIKS Dissertation Series No. 2010-54
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The research in this thesis was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 612.066.410.

ISBN: 978-90-365-3111-5
ISSN: 1381-3617, No 10-184
DOI: http://dx.doi.org/10.3990/1.9789036531115
Printed by: Wöhrmann Print Service, The Netherlands

© 2010 Riham Abdel Kader, Enschede, The Netherlands
© Cover design by Riham Abdel Kader
All rights reserved. No part of this publication may be reproduced without the prior written permission of the author.

ROX: RUN-TIME OPTIMIZATION OF XQUERIES

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation
committee to be publicly defended
on Thursday, November 25, 2010 at 15:00

by

Riham Abdel Kader
born on February 8, 1983
in Beirut, Lebanon

This dissertation is approved by:

Prof. dr. Peter M. G. Apers (promotor)
Dr. Maurice van Keulen (assistant-promotor)

Acknowledgments


```
1  <My_warm_thanks_go_to>
    <My_supervisors>
     <name> Peter Apers <!--Known as the big boss, I
        have found Peter to be a kind man always with
        a smile, and always there when any of his
        students is in need for help. Peter, I have
        always enjoyed and found interesting the
        pleasant conversations we used to have
        together.-->
     </name>
5    <name> Maurice van Keulen <!--for his patience,
        support, and belief that the conducted
        research will lead to the intended results.
        Maurice, with his continuously positive
        attitude, has been a reminder that even when
        the research path seems foggy and unclear,
        there always are bright spots on the way
        ahead.-->
     </name>
    </My_supervisors>
    <Cooperation>
     <name> Peter Boncz <!--I want to thank Peter for
        initiating the collaboration with CWI which
        has resulted in the birth of many of the ideas
         behind ROX. Working with Peter has been an
        eye opener, and I have learned a lot from him.
         Peter, you have been and still are an
        inspiration, and I fear that thanking you with
         these few words is not enough.-->
10   </name>
     <name> Stefan Manegold <!--Stefan has also been
        part of my cooperation with CWI. We have had
```

```
        several long and interesting discussions which
         were always fruitful. I owe Stefan some of
        the experimental and scripting skills I have
        developed during my PhD studies.-->
     </name>
     <name> Martin Kersten and his group <!--I would
        like to thank the Database group at CWI for
        making me feel welcomed whenever I was around.
         I also want to thank Martin for allowing me
        to use the CWI machines to conduct the ROX
        experiments, some of which are shown in this
        thesis.-->
     </name>
15   <name> Torsten Grust and his group <!--for the
        interesting discussions we have had at the
        start of my PhD studies during which the join
        graph idea has emerged.-->
     </name>
    </Cooperation>
    <DB_group>
     <!--During the 4 years of my PhD studies, the
        faces in the DB group kept on changing, some
        faces left and some new joined. Throughout it
        all, the DB group remained a cooperative and
        "gezelig" team of colleagues with whom I
        enjoyed working. Two persons of the group I
        would like to distinctively mention are Harold
         and Rongmei, with whom I shared many lunches
        and diverse conversations. Simply said: Harold
         and Rongmei, you have been good friends,
        helping me sailing through the Dutch waters
        whenever the waves became considerably
        high.-->
20  </DB_group>
    <Committee>
     <!--I would like to thank every member of my PhD
        committee for accepting to be in the
        committee, and for taking the time to read my
        thesis. I distinctively thank Jaco van Der Pol
         and Rolf de By for their detailed comments
        and feedback.-->
    </Committee>
   </My_warm_thanks_go_to>
```

And for those who find the above XML fragment a small labyrinth of technical words, I present my thanks in the following.

I start with Ida, the secretary and *heart* of the DB group. Ida and I have shared one common thing: being, for a part of my PhD years, the only ladies in the group. I think this has made our relation more special. Ida, I will miss you, not only as a hardworking, efficient, and helpful secretary, but also as a person I enjoyed to talk to and to share my news with.

I also want to thank my friends in Lebanon and The Netherlands. They have always been there, and believed from the beginning in my ability to follow and complete the PhD path.

Finally and most importantly, I thank my family, and especially my parents, for being supportive and caring throughout the four PhD years. Mom and dad, thank you for believing in my success and helping me seeking my PhD dream.

Riham Abdel Kader
Eindhoven, November 2010

Contents

Acknowledgments **i**

Contents **v**

1 Introduction **1**
1.1 Challenges in Database Optimization 2
1.2 Thesis Contributions 6
1.3 Thesis Structure 8

2 Background and Related Work **9**
2.1 Background 9
2.2 Related Work 15
2.3 Conclusion 26

3 Foundations and Formalization **27**
3.1 Join Graphs 27
3.2 Sampling Techniques 40
3.3 Notation 53
3.4 Conclusion 58

4 ROX: Run-time Optimization of XQueries **59**
4.1 Introducing ROX 60
4.2 The ROX Algorithm 66
4.3 Chain Sampling 75
4.4 Chain Sampling Implementation 104
4.5 The Power of the Run-time Optimizer 109
4.6 Conclusion 120

5 Prototype and Experiments **123**
5.1 Prototype Platform: MonetDB/XQuery 123
5.2 Sampling Operations 128
5.3 Overview of Experiments 135

5.4 XMark Experiment 136
5.5 DBLP Experiment 138
5.6 Conclusion 160

6 ROX-sampled: Towards a Pipelined Execution **163**
6.1 General Description of ROX-sampled 163
6.2 The ROX-sampled Algorithm 180
6.3 Experiments 193
6.4 Implementing ROX-sampled in Pipelined Database Systems 201
6.5 Conclusion 202

7 Conclusions **205**
7.1 Revisiting the Research Questions 206
7.2 Additional Strong Aspects of ROX 209
7.3 Future Research Directions 210

Bibliography **213**

Siks dissertations **225**

Summary **237**

Samenvatting **239**

1 Introduction

The emergence and spread of the Internet and distributed systems has increased the need for a flexible way to represent data, easing its exchange and integration with other data sources. This has led to the introduction of the semistructured language XML (eXtensible Markup Language). XML has been defined and standardized by W3C [1], and is a flexible language widely used to publish and exchange data on the web and in distributed systems. Several implementations to support XML have been developed. Some of these choose to store XML on top of relational database systems. These proposals have proved to be successful as they can make use of the mature relational technology. To access the stored data, XPath [5] and XQuery [6], the most common languages to query XML, can be used. An XPath expression can be seen as a sequence of steps, where each step corresponds to a join operation between two tables of XML nodes. While XPath can only select XML nodes from a document, XQuery is a superset of XPath allowing selection of nodes, updating of existing data, and construction of new XML fragments.

As with queries in the relational context, the processing of XQueries in an XML database system should be preceded by an optimization phase which aims at finding the cheapest execution plan based on some predefined cost functions. Query optimization in relational database systems has been thoroughly researched in the last four decades, resulting in a multitude of optimization techniques. These techniques, although quite mature, still face some challenges as we will see in the next section. The problems in relational optimization techniques are aggravated when imported into the XML context, because of the nature and complexity of XQuery. These challenges, in both the relational and the XQuery cases, might lead to poor plans chosen for execution, creating a pressing need to investigate new and more robust optimization techniques.

Thesis Focus

The subject of this thesis is the optimization of the execution of XQueries in a relational database system, with a focus on optimizing the execution order of XPath steps and joins in a plan. This is referred to as the *join ordering* problem. The *join ordering* problem is a classical problem extensively

researched in the relational context. In fact, one of the most expensive but heavily used operator in query processing is the *join* operator. Therefore, one of the main tasks and focus of a relational optimizer is to determine the execution order of the joins in a plan such that the processing is as efficient as possible. The problem is not trivial with a search space consisting of $n!i^n$ plans where n represents the number of joins and i the number of available physical implementations for the join operator. The complexity of this problem is intensified in the context of XQuery, where every XPath step adds an extra join operator to the formula, since as mentioned earlier an XPath step consists of a join between two tables. We stress that in some situations multiple axis steps in an XPath expression can be grouped into a single selection operation. This is the case when mapping techniques like for instance shared and hybrid inlining [115] are used to shred the XML document into relational tables, or when certain types of XML indices [39, 81, 82] are built. In this thesis, we consider the case where each XPath axis step is conceptually seen as a join, and also physically mapped into a join operator.

In short, this thesis is about query optimization techniques, addressing the challenges that are behind the deficiencies of current optimizers. We focus on the join ordering problem in the context of XQuery, while aiming at overcoming the more general join ordering challenges relational optimizers are still facing.

To tackle the above problem, we propose a solution which deviates from the traditional approach used by current optimizers. Our strategy is to *move the optimization phase to run-time*, making it possible for the optimizer to use and benefit from the available information about the queried documents and the constructed intermediate results.

In this chapter, we first review some of the important challenges still faced by traditional relational optimizers. We then present the research questions that will be studied in this thesis and briefly describe our proposed solution.

1.1 Challenges in Database Optimization

Challenges in Traditional Optimizers

Nowadays, database systems are exploited in several domains, from small institutions (schools, libraries) to big organizations (banks, hospitals, universities, governmental departments). The usage of the database system ranges from the execution of simple queries to retrieve specific pieces of information, to the generation of reports, to running complex queries that analyze the stored data to extract information useful to the business intelligence of the enterprise. With the constant increase in the amount of data stored in a database system and in the complexity of queries,

good optimizers are highly needed. Are traditional relational optimizers up to the task? In fact, in some cases these optimizers are successful in building good execution plans, and in other cases they generate plans that are far from optimal [28]. Moreover, relational optimizers have proven to be insufficient in the context of XML, especially that the optimization of XQueries brings in its own challenges. This section presents some of the most important causes behind the deficiencies, addressed in this thesis, of current optimizers.

1.1.1 Cardinality and Cost Estimation

The success of traditional optimizers relies on good cost models, which in turn are highly dependent on accurate cardinality estimation techniques. In a database system, techniques to estimate the cardinality of operators are based on collected statistics. Although a lot of work has been done on building good statistics, these are often an inaccurate not up-to-date reflection of the actual content of the database [71, 120].

Still even when the statistics are erroneous, the cardinality and cost estimations derived by the optimizer for a single operator are usually accurate enough. When estimating the cardinality of a sequence of operators, the optimizer uses the estimated characteristics of the intermediate result of one operator to estimate the cardinality of the subsequent operator. Small errors in the maintained statistics can lead to estimation mistakes which degrade and propagate exponentially through the plan. This may result, in case of big queries, in large estimation errors, and hence bad plans chosen for execution [71].

A cost model also determines the cost of an operation by estimating its CPU and I/O costs. Therefore, an accurate cost model should take into account the physical implementation of operators, and the cost of accessing data from disk. The latter is a non-trivial task since it requires the knowledge of the physical location of data and index pages, and the cost of a disk access. Although a lot of effort in the field has led to better cost models, cost estimations are still a challenge in query optimization [27, 28].

In the XML context, contrary to the relational context where only value statistics are collected, XML statistics should capture both the structure of the document and the value of the nodes, making the collection of adequate statistics a hard task [103]. A lot of work has been conducted to collect statistics from XML documents. Most propose to summarize the data content and structure into a synopsis; however, building a concise and still accurate representation of the relationships between the different nodes in the XML document it still considered a challenge. Without good statistics, cardinality estimation in XML fails to be accurate.

Cost modeling in XML is still a poorly researched topic. In fact de-

veloping cost models for XML query processing is much harder than building cost models in the relational context [131]. Advances in the field are hindered by the complexity of XML-specific operators, the data access of which is very hard to predict and to model.

1.1.2 Detection of Correlation Between Attributes

Even though a lot of work on building better statistics has been done, capturing correlations between attributes remains a challenge. In fact, summarizing the number of distinct pairs of values gives only very shallow knowledge about the existing correlations. Additionally using 2-dimensional histograms suffers from the large space of possible combinations, and picking the columns on which to build the histogram is non trivial as it requires a pre-knowledge of the correlated attributes [27, 28, 120]. Note that trying to capture the correlation between three or more attributes increases the complexity of the problem exponentially.

When no information about the correlation between two attributes is collected, the optimizer assumes that the values of both attributes are independent. The *attribute value independence* (AVI) assumption is used in cardinality estimation techniques to simplify the estimation process. Obviously, the AVI assumption does not hold in real-life data, leading to large errors in estimations [36, 71, 120].

Correlation between attributes does occur frequently in databases. The inability to detect it during optimization, and assuming attribute value independence, may result in cardinality estimations with orders of magnitude off from the real values, which in turn leads to picking bad plans for execution [36]. Similarly, in the XML context the problem of not detecting correlations may result in bad optimization decisions.

1.1.3 Large Queries

When queries include a small number of joins, it is possible to optimize the ordering of the joins using an exhaustive search algorithm possibly enhanced by pruning techniques that disregard categories of plans that are most likely bad. When optimizing the ordering of a large number of joins, these algorithms become prohibitively expensive when enumerating the large number of possible orderings [119].

In the XML context, an XPath expression can be seen as a sequence of steps, where each step corresponds to a join operation between two tables of XML nodes. Therefore, an XQuery, which usually consists of several XPath steps, contains on average more joins than relational queries. In the XMark benchmark [4], the number of joins in an XQuery ranges between 5 and 32. The number of joins in XQueries issued in real life is

expected to be even larger. This large number of joins in an XQuery makes its optimization a challenge.

1.1.4 Precompiled Plans

In some situations, traditional relational database systems precompile parametric queries into execution plans, and store these plans for future processing. Types of queries that are usually precompiled into execution plans are frequently issued queries, and queries that have a long and expensive optimization phase. Examples of such queries are stored procedures, and queries that originate from template forms. The reason for not re-compiling these queries every time they are initiated is to avoid spending time on optimization. Since such a query contains parameters, the value of which is specified at run-time, the optimizer compiles the query into the plan that performs well for the widest range of the parameters value. Nevertheless, the performance of a candidate plan may vary significantly with the different parameter settings. Therefore, in case the values of some parameters in an issued query fall outside the range optimized for, the pre-compiled plan will be sub-optimal and may result in a poor execution performance [75].

Parametric queries are even more common in the XML context. In fact, XQuery is often viewed and used as a functional programming language. As a result, users write queries which include their own defined functions, or even import libraries containing a large number of user-defined functions [2]. The input to these functions can be variables, context nodes and even the to be queried document. In this case, a precompiled plan may, for some values of the parameters, result in an unacceptable execution time.

1.1.5 Absence of Statistics

Users nowadays issue queries that involve data sources stored on remote machines (*e.g.* from the web) for which no statistics can be built [130]. More specifically in XQuery, documents may be accessed using the `fn:doc(`*url*`)` construct, which allows to specify the name of the to be queried document at run-time. In these cases, access to statistical data at compile time is not possible. Without any knowledge about data cardinalities, the optimizer will derive an execution plan based on generic heuristics, with a high chance of making bad choices.

We conclude this section by noting that current optimizers have become a module with quite a complex logic. In fact, it has been proven that in general it is hard to predict the decisions made and the execution plans chosen by optimizers [111].

1.2 Thesis Contributions

This section presents the contributions of this thesis. We start by giving the research questions that will be the subject of investigation in this thesis. We then give a brief description of the solution we propose.

1.2.1 Research Questions

The focus of this thesis is to design a robust query optimization technique that can find a good execution order for the XPath axis step and join operators in a given XQuery, while overcoming the challenges described in Section 1.1. To achieve this, we need to answer the following research question which is in fact the center of investigation in this thesis:
Main research question: *How to develop an XQuery optimizer which has the following properties:* ***autonomy****,* ***robustness*** *in always finding a good execution plan, and* ***efficiency****.*

To construct the required *autonomous*, *robust* and *efficient* optimizer, the above main research question is divided into the next three sub-questions:

- ***Research question 1:*** *How can an optimizer accurately estimate the cardinality and cost of operators without relying on any a priori collected statistics and cost model?*
- ***Research question 2:*** *How can the correlation existing between several attributes be detected and exploited?*
- ***Research question 3:*** *How can the proposed optimizer guarantee a good quality of decisions?*

Additionally, our proposed optimizer should be suitable for the different existing database system architectures, leading to our fourth research question:
Research question 4: *How can our proposed optimization technique be applied to different database system architectures (full materialization and pipelined execution strategies)?*

1.2.2 Approach

In this thesis, we adopt a fundamentally different approach to query optimization. We propose ROX, a Run-time Optimizer for XQueries, which radically departs from the traditional path of separating the query compilation and query execution phases. This section starts by giving a general description of the ROX approach and then explains the way ROX satisfies the properties enumerated in Section 1.2.1.

General Description: ROX performs the optimization of queries at run-time. It 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 the cardinalities and costs 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.

Autonomy and no dependence on statistics and cost model: By deferring optimization to run-time, it becomes possible to accurately observe the characteristics and size of intermediate data, and to accurately estimate the cost of operators. ROX uses sampling techniques to accurately estimate the cardinality and cost of the different operators, which makes it autonomous and independent of any a priori collected statistics and cost model. Note that the alternation between optimization and execution steps allows the optimization phases of ROX to use the newly materialized intermediate results as input to their sampling operations. This makes it possible to update the previously estimated cardinalities, and results in more accurate cardinality estimations.

Robustness and correlation detection: The alternation of optimization and execution steps followed by the full materialization of results is the main factor behind the robustness of ROX. Another reason that makes ROX robust is that our approach uses a *chain sampling* technique to avoid a local optimum during its search for the sequence of operators to execute. Moreover, ROX can detect the correlations between two attributes by sampling the operator that joins the two tables corresponding to the attributes. By sampling a sequence of operators, ROX can detect if any correlation exists between several attributes. ROX does not only detect the existing correlations, it also naturally exploits these correlations in its decisions which operators to execute next.

Efficiency: 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. As we have explained in Section 1.1.4, current optimizers precompile some plans to avoid the long re-optimization phase of their corresponding queries, and therefore might run into the risk of executing a bad plan. ROX can overcome this problem by controlling the amount of time spent on optimizing a query.

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. ROX can decide to invest more or less resources in optimization based on an estimation of the execution cost of the query (which is re-estimated after every optimization and execution step).

Although the ROX approach is explained in the context of XQuery, we emphasize that the proposed optimization strategy is general enough to be used for other query languages, like SQL and SPARQL.

1.3 Thesis Structure

Chapter 2 starts by giving a background overview on query optimization in traditional relational compile-time database systems, and then reviews some of the related work in the literature.

In Chapter 3, we introduce the building blocks of the ROX approach: join graph, and sampling and estimation techniques. We also present the notations that will be used in the subsequent chapters.

The ROX optimizer is described in detail in Chapter 4. In this chapter, we direct our focus on explaining ROX in the context of database systems that support full materialization. We start by presenting the algorithm of the ROX approach, and then describe the chain sampling process while showing the differences between the theoretical and implemented versions.

Since a prototype of ROX is implemented on top of the relational database system MonetDB/XQuery [3, 20], Chapter 5 gives a quick description of the storage structure and operators in MonetDB, and some details about the implementation of the sampling and execution of operators in MonetDB/XQuery. Finally, experiments evaluating the performance of ROX are presented.

Chapter 6 presents a variant of the original ROX algorithm that is suitable for database systems with a pipelined execution scheme. First, the main differences between the two algorithms will be described. Afterwards, the algorithm of the new ROX variant is given and explained in details. Last but not least, experiments evaluating the new ROX variant and comparing it to the original one are presented.

We end with a summary and a conclusion of the thesis in Chapter 7, in which we also present some possible future research directions.

2 Background and Related Work

The first part of this chapter serves as a short background for the subject of query optimization in database systems, and can therefore be skipped by readers who are familiar with the topic. In the second part of the chapter, we give an overview of the related work.

2.1 Background

Database Management Systems (DBMS) were introduced to satisfy the need to organize, store, query and manipulate large amounts of data. In the early database systems, data was represented in a tree-structured file or a graph/network model. The architecture of these databases required pre-knowledge of the data organization in the machine to access it. The necessity to efficiently access and search the stored data without the knowledge of its internal representation (referred to as data independence) was the motivation behind the *relational model* proposed by Edgar Codd at the beginning of the 1970's.

Besides the notion of data independence, the power of the relational model arises from the proposed relational storage structure and algebra. The relational model structurally organizes data in n-ary *relations* (or tables). The relational algebra defines a set of operators with which relations can be manipulated. Examples of operations are: selecting from a relation tuples that satisfy a given predicate, joining matching tuples from two different relations, and grouping tuples in a relation that partially share similar data values.

Users interact with the database by issuing declarative queries, written in a high-level language, in which they only need to specify the needed data without worrying about the location and structure of the data or how their request will be answered. It is the task of the database system to find and retrieve the requested data, and construct the result to be returned to the user. The process of answering a user's query consists of several phases (shown in Figure 2.1):

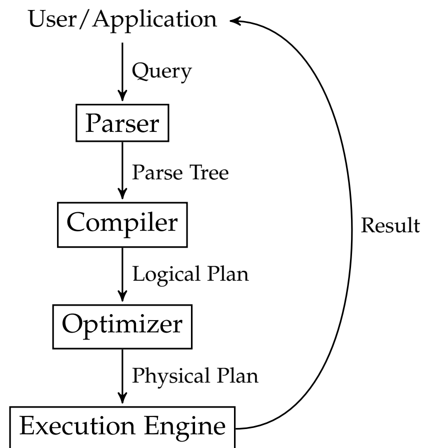


Figure 2.1 Phases of query processing

Query Execution Phases

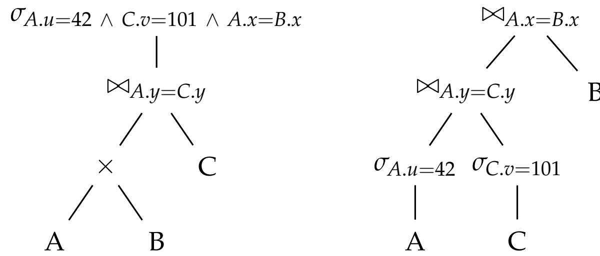
- ***Parsing:*** The parsing phase is responsible for constructing an internal representation of the query, called parse tree. The parse tree corresponds to the query's structure.
- ***Compilation:*** Compilation consists of converting the parse tree into an initial logical plan which consists of a sequence of algebraic operators that can produce the query's result.
- ***Optimization:*** This phase optimizes the logical plan into an equivalent plan that is estimated to be cheaper to execute. In this thesis, a cheap plan refers to a plan that executes in a small amount of time, hence the optimization phase consists of a search for the fastest plan.
- ***Execution:*** In this phase, the plan generated by the optimizer is executed, the result of the query is constructed and returned to the user.

2.1.1 Optimization in Databases

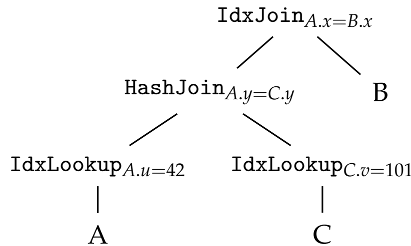
Optimization is the hardest and the most complex phase of processing a query. To find the cheapest (*i.e.* fastest) plan, the optimizer has to make two kinds of decisions:

Optimizer's Tasks

1. ***Choosing the type and execution order of operators -*** The type and the order of the operators in the plan largely determine the amount of investment needed to execute the query. Using algebraic rewriting rules, the optimizer can *reorder* a set of operators to make their execution more efficient. Other rewriting rules *replace* a set of operators in the plan with a more efficient sequence of operators, *e.g.* substituting



a Two different algebraic plans containing selection, cartesian product, and join operators. The two plans correspond to the same query and return the same result although their operators are ordered differently. We say the two plans are equivalent. It is the task of the optimizer to reorder the operators to find a better plan. If the select operators are highly selective, the optimizer should push them below the join to reduce the size of data generated and processed. Moreover joins can be reordered such that the most selective join is executed first. Note that the cartesian product ($\times$) and selection ($\sigma_{A.x=B.x}$) operators between the two tables A and B are replaced with a join operator ($\bowtie_{A.x=B.x}$).



b The physical plan chosen by the optimizer. The optimizer decides to implement the select operators as index-lookups. One of the joins is executed as hash-based join, while the other uses an index-based implementation.

Figure 2.2 Optimizer's tasks: determining the best order and physical implementation of the operators in the plan.

a selection and a cartesian product with a join. An example is shown in Figure 2.2a. The two plans are equivalent, *i.e.* they correspond to the same query and return the same result. If the selection predicates are highly selective, it is more efficient to push the select operators below the joins. This reduces the size of generated data, and hence the processing time of the joins. Similarly the joins can be reordered such that the most selective join is pushed further down. Note that the cartesian product ($\times$) and selection ($\sigma_{A.x=B.x}$) operators between the two tables A and B are replaced with a join operator ($\bowtie_{A.x=B.x}$).

2. ***Choosing the physical implementation of operators -*** An algebraic operator denotes the type of operation to be performed on a set of relations but does not specify the way it should be carried out. In fact, for each algebraic operator in the database, there exist several corresponding physical operators. A physical operator is simply the algorithm that implements the functionality of the algebraic operator. For instance, the join operator can be implemented as a nested loop join, a sort-merge join, or a hash-based join. When indices are available, a select operator is implemented as either a table scan or an index lookup depending on the selectivity of its predicate condition. It is the job of the optimizer to map each operator in the plan to the appropriate physical implementation such that the resulting physical plan has a fast execution. Figure 2.2b depicts one possible physical plan corresponding to the second logical plan shown in Figure 2.2a. The select operators are implemented as index-lookups. A different physical operator is chosen for each of the joins: one is executed as a hash-based join, while the other uses an index-based implementation.

To perform the above two tasks, the optimizer is usually provided with the following:

1. ***Search space of plans:*** The search space includes all equivalent plans that can be generated for a single query. Each plan corresponds to one possible ordering of the operators and one of the different combinations of physical implementations supported for each operator.

2. ***Enumeration algorithm:*** An enumeration algorithm explores the search space looking for good candidate execution plans.

3. ***Cost model:*** To assist the enumeration algorithm in assessing how good a candidate plan is, the cost model computes for a given plan in the search space an estimation of its cost. The cost of a plan is a quantitative estimation of the resources consumed while executing the plan. The cost function consists of a weighted formula of the combination of resources chosen to be measured (*e.g.* CPU time, I/O cost, memory, ...).

A good quality optimizer requires an efficient and effective enumeration algorithm and a highly accurate cost model. Developing these two components with the above properties is not a trivial task. We explain these two components next, but before we proceed, we shall stress that an optimizer is not required to find the optimal plan for every submitted query. In fact, a good optimizer is one that satisfies the following two properties:

- **Robustness:** A robust optimizer ensures that a good (possibly sub-optimal) execution plan is chosen for any input query, and reliably avoids picking an expensive plan.

- **Efficiency:** An efficient optimizer should not spend a considerably large amount of time optimizing a query. Ideally it should balance between the time spent on optimizing a certain query and the time needed to execute the chosen plan.

2.1.2 Enumeration Algorithm

An efficient enumeration algorithm is one that picks from the search space the most promising candidate plans in a short amount of time. To make the search efficient, it should explore the search space in a selective way, disregarding whenever possible the likely expensive categories of plans. Next, we briefly describe a few of the existing categories of enumeration algorithms:

1. ***Exhaustive top-down:*** This strategy starts from the root operator of the plan and walks its way down. For each encountered operator, all possible implementations are considered, then combined with the possible alternatives for its children operator(s). When all required operators are added to the plan, the execution with the cheapest cost is picked.
2. ***Exhaustive bottom-up:*** It starts by considering all different ways to access each single relation. Then all possible plans are generated by iteratively considering every remaining operator and appending it to each of the already produced plans. When all required operators are added to the plan, the execution with the cheapest cost is picked.
3. ***Greedy bottom-up:*** This strategy is similar to the exhaustive bottom-up approach; however, at each step, only the cheapest enumerated plan is kept and extended with the next operator.
4. ***Heuristic-based:*** This approach generates an execution plan by making a sequence of decisions based on heuristics. Examples of commonly used heuristics are: *(i)* if available, use indexes to scan a relation, *(ii)* first join the two relations with the smallest estimated result cardinality, *(iii)* use an index-based join when one of the joined relations has a corresponding index, *(iv)* use a sort-merge join if one of the join's input is sorted, *(v)* consider only left-deep plans.
5. ***Branch-and-Bound:*** With this approach, a plan is first generated using the heuristic-based technique. Then, using algebraic rewriting rules, the operators in the plan are reordered to generate a better plan. Branch-and-bound pruning is used to disregard any permutation that generates a plan with a cost higher than the best so far.

Some of the above techniques apply an exhaustive search to the space of plans. Although this guarantees that the best plan is found, for non-simple queries, the search space can be prohibitively large, which can result in a

slow and unacceptable performance of the optimizer. The other techniques explore only part of the search space and run the risk of disregarding good plans. As we will see in Section 2.2.1, other more sophisticated enumeration algorithms have been proposed (*e.g.* randomized search techniques).

2.1.3 Cost Model

The cost of a plan is derived by combining the estimated cost of all its operators. The execution cost of an operator is affected by two factors:

1. The amount of data it has to process.
2. The CPU and I/O costs of the operator. The first refers to the computational complexity of the algorithm implementing the operator's functionality, while the latter denotes the number of disk blocks that are accessed during the execution of the operator.

To estimate the amount of data an operator has to process, the cost model should be able to access *statistical information* collected on base tables, and to *estimate the size and characteristics of intermediate data returned by operators*. We will shortly describe the techniques used to achieve this. The CPU cost of an operator is derived from the implementation of the operator using algorithmic complexity theory. Given some knowledge about its data input, the I/O cost of an operator is determined using cost formulas derived and integrated in the cost model.

Statistical summaries For every table in the database, statistical information is collected. A piece of statistics is related to either the whole table or to one of the table's attributes. Examples of statistics are:

- Statistics about a table: Statistical information concerning a table consists of, among others, the total number of tuples, the average size of a tuple, and the number of blocks used to hold all the tuples of the table.
- Statistics about an attribute: Statistical information concerning an attribute consists of, among others, the highest/lowest (or second highest/second lowest) values of the attribute, and the value count (number of distinct values) of the attribute. To capture the value distribution of an attribute, histograms can be built in which, for example, the frequency of the distinct values of the attribute can be collected.

Statistics can also be collected for multiple attributes belonging either to the same table, or to several different tables. This is an important kind of statistics, since it allows the optimizer to capture correlations between two or more attributes. Unfortunately this type of statistics is not easy to build. One proposed technique to capture the correlation between two

attributes is to summarize the number of their distinct pairs of values. A more detailed representation uses 2-dimensional histograms. As we have seen in Section 1.1.2, these two techniques still have some drawbacks. In general, the challenge in collecting statistical information that allows accurate cardinality estimations is in having a representative view of the stored data while keeping the size of the statistics as small as possible.

Result Size Estimation techniques These techniques are used to estimate *the selectivity of operators*. This is accomplished by trying to accurately propagate the statistical information collected about base tables through the operators in the plan to estimate the cardinality and in some cases the value distribution of the output data of each operator. To simplify this complex task, cost models usually make the following assumptions:

- Uniformity of the distribution of values within an attribute's domain.
- Attribute Value Independence (AVI) which supposes that the value of an attribute is independent of that of (an)other attribute(s) belonging to the same or different relations.

We have given in this section a quick introduction to query optimization in classical database systems. We now proceed with reviewing some of the most important work related to ROX, our run-time optimizer for XQueries.

2.2 Related Work

Since the join operator is one of the most expensive operators in query processing, but also a heavily used one, the focus of ROX, as other optimizers, is on solving the join ordering problem. We stress that, due to the existence of many XPath steps in a typical XQuery query, a relational XQuery plan contains more joins than a typical relational SQL plan, making the optimization of the order of joins even a bigger necessity in XQuery. In this section, we present, on the one hand, existing solutions that optimize the order of joins at compile-time, and, on the other hand, suggested techniques for run-time optimization. As this thesis uses sampling techniques to estimate the cardinality and cost of operators, we refer the reader to [100] for a survey on sampling techniques. The sampling techniques used in ROX are described in Sections 3.2 and 5.2.

2.2.1 Join Ordering at Compile-Time

In this section, we give an overview of the existing compile-time join enumeration algorithms in the relational context, and then quickly review the proposed XML query optimization techniques.

The join ordering problem has been extensively researched over the last three and a half decades, leading to the development of several compile-time optimization algorithms. Given an n-way join query and i different ways of evaluating a join, these algorithms should pick the optimal order of joins and the best join implementation to use among the $n!i^n$ possible choices. To accomplish this, compile-time optimizers explore the search space of plans using different enumeration strategies, and a cost model that associates an execution cost to each candidate plan.

Relational Join Ordering Using Deterministic Algorithms

The pioneering work in enumeration algorithms for ordering joins is the optimizer proposed in System-R [113]. It uses heuristics and dynamic programming to exhaustively explore in a bottom-up manner the search space of candidate plans while taking into consideration "interesting orders". To limit the search, the approach focused only on left-deep plans, and considered cartesian products only after determining the optimal order of joins. The System-R optimization framework is not easily extendible with new logical transformations (beyond join ordering) and new physical operators. This has led to the development of the extensible transformation-based optimizers: Starburst [62] and Volcano [54].

In Starburst, a query is optimized in two rule-based phases. The first [102] consists of rewriting the query by, for instance, merging nested subqueries, pushing down projections, and reordering the selections. This phase does not have access to the cost model and is heuristics-based. The second phase optimizes the order of joins in the query by exhaustively exploring the search space in a bottom-up fashion while using dynamic programming, pre-defined grammar-like rules and a cost model [87, 90]. The Starburst optimizer can take into consideration cartesian products and both left-deep and bushy plans.

The Volcano optimizer evolved from Exodus [53] and uses two types of rules to optimize a query. The first consists of transformation rules that map an algebraic expression into an equivalent one, while implementation rules, the second type of rules, are used to map the operators in an algebraic expression into the best corresponding physical algorithms. The enumeration algorithm uses dynamic programming in a top-down fashion with memoization, and considers left-deep and bushy plans. Memoization avoids the execution of redundant work: an optimization task that has already been accomplished is not performed a second time. The advantage of this top-down approach is its ability to perform early pruning of sub-plans that are known to be suboptimal. DeHann et al. [40] proposed a top-down enumeration algorithm that is not based on transformations, hence allowing the dynamic programming optimizers to exploit the benefits of the top-down approach. Their approach considers bushy trees

without cartesian products. The Volcano and Starburst transformation rule-based optimizers suffer from one disadvantage of using a large amount of memory to store all enumerated plans [40, 101].

Dynamic programming has also been used by other work as an enumeration technique to exhaustively explore the search space of joins. The generalization of the System-R approach to bushy plans is given in [97]. Vance et al. propose to use dynamic programming to optimize multi-way joins while considering bushy plans and cartesian products [126]. The techniques presented in [18, 49, 47, 98, 109, 112] suggest different dynamic programming algorithms to efficiently enumerate possible join ordering while considering bushy trees excluding cartesian products and including other types of joins that in general are not freely re-orderable, *e.g.* outer-joins and anti-joins.

Join order enumeration algorithms targeting specific join graph shapes have also been proposed. For acyclic graphs with n joins and a cost model satisfying certain properties [68], the work in [85] suggests the KBZ algorithm which can return the optimal plan in $O(n^2)$. An extension to cyclic join graphs has been proposed in the same paper but it no longer guarantees the generation of an optimal plan. For the same class of join graphs and cost models, Cluet et al. [37] present a theoretical study of the enumeration problem when considering left-deep plans and cartesian products.

Some heuristics-based join enumeration techniques have been proposed. We first mention those that adopt a greedy approach, building the execution plan step-by-step, adding one join at a time [43, 86]. Techniques mixing dynamic programming and heuristics/greedy algorithms have also been suggested to handle larger queries [84, 99]. Kossmann et al. [84] apply dynamic programming iteratively during the query optimization: it uses dynamic programming to optimize the order of a subset of the joins in the query, then restarts dynamic programming using the already selected plan as building block to order a second set of joins, and repeats the process until all joins are part of the final plan. The work in [99] proposes to simplify the initial join graph of a complex query by ignoring non-promising join edges and then explores the simpler and smaller graph using the exhaustive search described in [98].

Some of the relational compile-time optimizers described above explore the search space of plans exhaustively resulting in an unacceptably large amount of time spent on optimization. The other approaches use heuristics to reduce the number of enumerated plans, and hence risk disregarding the (near-)optimal ones. Moreover, the quality of produced execution plans highly depends on the quality and accuracy of collected statistics and cost model. As we have explained in Chapter 1, the latter are often not accurate and not up-to-date which might result in bad plans picked for execution.

Relational Join Ordering Using Randomized Algorithms

To address the prohibitively exhaustive search of the early dynamic programming algorithms when optimizing complex and large queries, *randomized optimization techniques* have been proposed. These algorithms view the search space of plans as points in a high-dimensional space. The different plans are connected by transformation rules named *moves*, *e.g.* join commutativity and associativity. The algorithms navigate in the space by performing random moves along the edges between the plans. The advantage of randomized optimization techniques is that they have a constant space overhead, and although slower than heuristics and dynamic programming approaches when optimizing simple queries, they are faster for large queries. A survey of the existing randomized optimization techniques can be found in [119]. We present some of the most important algorithms next.

Iterative Improvement [72, 73, 123, 122] starts with a random plan and iteratively performs a move to a new plan with a lower cost. When none of the neighboring plans have a lower cost, the whole process is re-initiated with another new random plan. This is repeated until a time limit is reached, then the plan with the lowest cost is returned.

The problem with Iterative Improvement is that it might get trapped in local minima. *Simulated annealing* [73, 76, 123, 122] is a variant of Iterative Improvement which overcomes this drawback. Starting with a random plan, it makes several random moves always accepting those that yield a lower cost plan. To avoid a local minimum, it sometimes climbs the hill by following, based on some probability, moves that result in a plan with a higher cost. This probability decreases as optimization proceeds until reaching the value 0. When optimization ends, the plan with the lowest cost is returned for execution.

The *Two Phase Optimization* [72, 73] algorithm combines the above Iterative Improvement and simulated annealing approaches. The technique consists of first finding a local minimum and then exploring the space around it in search for a global optimum. This is performed in two phases. The first phase starts by applying Iterative Improvement for a short time to find several local optima. Then the algorithm proceeds by starting simulated annealing using the cheapest plan found during the previous phase. A low probability is used to avoid climbing considerably high hills.

The quality of the plan generated by the Iterative Improvement and simulated annealing enumeration algorithms depends highly on the quality of the starting random plan. To guarantee better results, Swami et al. [122] have proposed to combine the two techniques with heuristic algorithms which generate a good quality initial plan to be fed to the randomized optimization approaches. The work in [48] proposes to explore the space of plans by performing a randomized walk without applying tree trans-

formations. As such, the considered candidate plans are chosen uniformly at random from the search space and compared on their estimated cost, eliminating the overhead of performing tree transformations.

In addition to the above optimization techniques, genetic algorithms have been proposed to solve the join ordering problem [17, 51, 119]. These algorithms simulate the biological phenomenom of evolution in their search for an optimal plan. The idea is to start with a random population of plans, each with its own cost. Pairs of plans are picked from the population and "crossed-over" generating new "offspring" plans which hold some of the features of their parents. During the cross-over, "mutations" can be introduced in the offspring plans. The parents and children with the least cost survive to the next generation. The algorithm terminates when the entire population consists of the same "fittest" solution or after a predetermined number of generations. The fittest solution of the last generation is returned for execution.

Although the randomized algorithms described above succeed in efficiently optimizing large queries, they still depend on statistics and a cost model to assess the quality of enumerated plans. However, the statistics and cost model are sometimes inaccurate resulting in bad optimization decisions. Moreover, as these algorithms have a random behavior, they provide no guarantee on the quality of the produced execution plan, especially that the latter highly depends on the quality of the starting random plan. Even when the starting plan is generated using a deterministic enumeration algorithm, it might still be far from optimal due to inaccurate statistics and cost model used during the deterministic plan enumeration and selection.

Join Ordering in XML

To optimize queries in the XML context, some researchers have suggested to reuse the already mature relational optimization techniques. However, these techniques have proved to be insufficient [16] and new XML-specific optimization solutions have been proposed. We quickly review these next. Before we proceed, we point to the fact that, unlike the relational case, different XML database systems adopt different algebras (*i.e.* relational algebra [20], tree algebra [78]) and structural operators (*i.e.* staircase join [57], stack tree [12], TwigStack [26], . . .). This adds to the complexity of the optimization problem in XML.

Accurate estimation of the cardinality of intermediate query results (for XML optimization purposes) has been extensively researched, resulting in a multitude of techniques [10, 34, 45, 46, 104, 105, 106, 125, 127, 128]. However, these still do not cover the full problem of XQuery intermediate result size estimation. They all propose to build a synopsis and/or histogram that captures the XML document structure and element 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 cannot 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, resulting in inaccurate estimations in some cases.

Similar to the join operator in the relational context, the structural join which evaluates one or more XPath steps is a central operator in XML query evaluation. Therefore, some work has focused on proposing techniques to determine the best execution order of structural joins in an XQuery. Wu et al. [129] present several structural join ordering algorithms, implemented in the Timber XML database system [77]. They stress that some of the heuristics, *e.g.* limit the search to left-deep plans, introduced to solve the relational join ordering problem, result in sub-optimal plan if applied in the context of XML. Stating that exhaustive search through dynamic programming can be expensive, they propose a new dynamic programming algorithm with pruning. Another dynamic programming algorithm with aggressive pruning combined with two different heuristics is suggested. Finally, they present an algorithm which focuses on the generation of only fully-pipelinable plans.

May et al [92] propose a join ordering algorithm for Natix, a native XML database system. They show that the optimizer can find better plans if the search space is extended to also include non left-deep plans and to disregard document ordering preservation during query optimization. In the latter case, a sort operator is added to the final plan to restore the correct order. Tested in the context of Natix, the work in [80] focuses on optimizing the evaluation of XPath steps by ordering the execution of navigational primitives such that the I/O operations are performed as efficiently as possible. All operators requiring I/O access are grouped and executed together employing efficient I/O strategy, *e.g.* either sequential scans or asynchronous I/O.

To reduce the search space of plans and the complexity of the enumeration algorithm, optimization in the Lore system decides greedily about the choice of physical operators and makes use of some heuristics rules [95]. In [94], McHugh et al. describe 6 enumeration algorithms and a few post-optimization techniques to efficiently optimize branching path expressions in the context of the Lore system. The proposed algorithms adopt a top-down approach and use aggressive pruning heuristics along with greedy choices focusing on the generation of left-deep plans. As various pruning techniques are employed, each algorithm examines a different subset of plans from the search space.

Grust et al. [60] suggest to outsource the task of ordering the joins in an XQuery to relational database systems. They start by noting that the

performance of relational optimizers is unsatisfyingly low when handling XQueries. This is due to the fact that XQueries usually compile into plans having odd shapes in which joins are scattered around with blocking operators in between, disallowing the optimizer to freely reorder the joins in the plan. Therefore, they present a method that reorders the operators in the relational XQuery plan such that blocking operators are moved towards the root of the plan and joins are grouped into clusters named join graphs. These join graphs provide an order-free representation of the joins in an XQuery. They have shown, through experiments, that relational database systems are capable of efficiently optimizing the isolated join graphs into join trees that execute fast while breaking-up and stitching complex path expressions. Although this work has shown that the optimization of joins in XQuery can be delegated to a relational optimizer, we stress that the problem of inaccurate and not up-to-date statistics and cost model and their impact on the quality of generated plans persists and is still not solved.

As can be seen from the above presentation, the optimization techniques targeting join ordering in the XML context are still few and considerably primitive. Efficient and robust ordering algorithms are highly needed. Moreover, we notice that the proposed techniques, except for [60], do not take into account relational joins, *i.e.* XPath steps and relational joins are not optimized indifferently of each other. ROX, our Run-time Optimizer for XQueries, can robustly and efficiently determine a (near-)optimal ordering of the joins in an XQuery, while seamlessly optimizing the order of both relational joins and structural joins. In fact, the isolated join graphs of [60] are used as input to the ROX optimization process. ROX not only re-orders XPath steps and relational joins, it also breaks-up and stitches complex path expressions and determines the optimal execution of the step by reversing its axis if necessary.

The quality of the decisions made by the previously described relational and XML compile-time optimizers highly depends on estimated values about, among others, document characteristics, intermediate results cardinality, and system load. As explained in Chapter 1, the accuracy of the estimations is never guaranteed, possibly resulting in poor quality decisions and therefore bad plans picked for execution. Moreover, these optimizers cannot detect correlations among the queried data resulting in more bad optimization decisions. To overcome the above problem, other types of optimization approaches have been proposed to operate at run-time to benefit from the available accurate result size and cost observations. The next section gives an overview of the available adaptive and run-time join ordering algorithms.

2.2.2 Run-time Optimization

The use of run-time techniques to mitigate the problems faced by compile-time optimizers 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 can be found in [41]. We present here the most important of the proposed techniques.

Dynamic Plans

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 might take at run-time. These parameters might include the value of variables in the query, the amount of system resources available at run-time, and the data characteristics. When at run-time the value of these parameters is known, the optimal plan is picked and executed. The choice of the appropriate (sub-)plan is performed before the start of execution or during execution at materialization points. The latter allows the use of intermediate results characteristics to decide which sub-plan to execute. The dynamic plans technique is suitable for scenarios where the queries are compiled once and executed repeatedly, possibly with different parameter values. The main challenge with this approach is in the explosion of the number of plans when considering the space of parameters values, and in the decision which of the plans to keep until run-time.

Dynamic plans were first proposed in [38, 55] where a *choose-plan* operator is introduced to connect the candidate plans, and to choose the optimal one to execute based on run-time information. The choose-plan operator might be inserted anywhere in the execution plan. The work in [38] devises a search strategy based on dynamic programming to determine the dynamic plans and insertion points of the choose-plan operators. The work in [74, 75] uses parametric optimization to optimize queries while focusing on the buffer size as the unknown parameter. They propose to use randomized algorithms instead of dynamic programming to generate the dynamic plans. Contrary to the previous work, this technique does not incur any run-time overhead.

There exists a multitude of works [50, 66, 67, 107, 110] that analyze the parametric query optimization problem and its complexity while focusing on cost functions that might be easier to optimize for. The studied cost functions were linear [50, 66, 107], or piecewise linear [66], or non-linear [50, 67, 107] in the given parameters.

Performing parametric query optimization for infrequently executed queries is not cost effective. To solve this problem, Bizarro et al. [19] have

recently proposed to progressively perform the parametric optimization of a query. Whenever the same query is submitted with different parameters, the optimizer decides whether to reuse a previously generated plan or to optimize the query for the specific values of the parameters and add the produced plan to the complete parametric plan. A *competitive* technique similar to the ones described already was proposed in the DEC RDB system [13]. It consists of running multiple access methods in parallel to determine the most promising one with which to execute the query. Once the winner is chosen, all other executions are stopped.

Mid-Query Re-Optimization

Instead of generating several potentially optimal plans at compile-time, and decide at run-time which ones to choose, re-optimization techniques generate a single plan, and during the execution phase re-run the optimizer if the current plan, due to unexpected data selectivities or changing system resources, is detected to be sub-optimal. The approach consists of inserting *checkpoint* operators in specific locations in the plan to monitor at run-time the flowing tuples and collect statistics about essential cost factors (*e.g.* result size and selectivity of operators). If the statistics observed at run-time do not coincide with the estimated values, re-optimization is initiated. The main challenge in this approach is to determine where to place the checkpoints, which statistics to collect, the necessary gap between the actual and the estimated values for which re-optimization should be triggered, and the best way to switch plans.

The approach in [79] uses heuristics to determine whether to trigger the re-optimization of a query. It computes for each estimate made by the optimizer an inaccuracy potential level, and inserts checkpoints in the plan at these operators that have a high inaccuracy potential level. A follow-up proposal [91] studies several kinds of checkpoints: *lazy* placed above materialization points, *eager* placed anywhere in the plan, *forced materialization* placed in useful location in the plan introducing new materialization points. It computes for each checkpoint an approximate *validity range* that defines an upper and lower bound outside which the sub-plan is considered sub-optimal. Babu et al. [15] propose an alternative to validity ranges called *bounding boxes*. The idea is to take a proactive approach to re-optimization by predicting, preparing and planning, during the initial optimization itself, for a possible re-optimization at run-time. To achieve this, they compute, for each uncertain cardinality estimate, a bounding box that represents the uncertainty in the estimate, and that is supposed to cover the actual cardinality. The bounding box is then used to define a set of "switchable plans" that are robust to possible errors in the estimations, hence reducing the need for re-optimization. Switchable plans are plans among which it is easy and cheap to switch. The techniques presented

in [89, 96] study the problem of re-ordering left-deep pipelined joins at run-time. The first re-orders the inner and outermost tables of indexed nested-loop joins while the second considers only inner tables of both indexed nested-loop and hash-based joins. Li et al. [89] monitor join selectivities throughout query execution and reorder operators to execute the more selective joins first; however, they assume independence between the selectivity of join operators. In [96], during the execution of a given plan, alternative plans are also sampled to estimate the selectivities of joins and then decide if re-ordering is required. This approach might need to sample a large number of alternative plans. The work in [42, 124] studies the re-optimization of the execution order of operators in subplans of federated queries.

2.2.3 Learning Techniques

We quickly mention some other proposed techniques that, similar to the re-optimization approaches, monitor the execution of a plan, not to re-optimize it, but to feedback their observations to the optimizer to adjust the statistics and cost model [11, 24, 25, 31, 32, 63, 118, 120]. Chaudhuri et al. [30] stress the fact that the aforementioned techniques are limited in their ability to monitor the execution of the plan and hence to improve the optimizer's decisions. They propose to increase the learned knowledge by monitoring, in addition to the execution plan, several alternate execution paths. These efforts all fall under the umbrella of improving optimizer robustness; however the learning curve might be long before the estimates become accurate.

The classes of techniques described above are all plan based. The first type of techniques constructs several plans and picks the one to execute based on run-time information. The second class of techniques constructs a single plan and re-optimizes it when the data characteristics and cost observed at run-time differ from the estimates made during optimization. The third group of techniques constructs a single plan, monitors its execution to learn about data characteristics, and updates the statistics and cost model accordingly. We stress that the quality of the plans generated and executed still depends on the accuracy of the statistics and the cost model, even though the second class of techniques attempts to recover from estimation errors. Moreover, they still suffer from the overhead of collecting and maintaining statistics. Finally, they cannot anticipate nor detect the existence of correlations. ROX goes beyond these approaches by continually intertwining optimization and execution, and effectively basing all decisions on cardinalities and costs accurately observed through sampling, removing any dependency on a priori collected statistics and a pre-built cost model, hence making it much less vulnerable to estimation

errors and undetected correlations. In short, the described techniques have a reactive behavior and cannot detect early enough selective correlations that 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.

2.2.4 Eddies

Unlike the previous plan-based techniques, this routing-based approach [14, 108] views query execution as a process of routing tuples through the most efficient sequence of operators based on properties observed in the data. Eddies can be seen as an approach in which the query evaluation plan is continuously reordered on a tuple-by-tuple basis. The introduced eddy operator acts as a router for the tuples. It monitors the execution and makes the routing decisions for each tuple. It also collects statistics about the query execution and uses these to make the appropriate routing decisions. Adapting to the changes in data characteristics consists of simply routing the tuples through a different order of operators. We note that Ingres, one of earliest adaptive query processors, had already incorporated a basic tuple-based routing strategy [121].

The common aspect between Eddies and ROX is that they completely interleave optimization and execution steps; however, contrary to ROX, no concrete execution plan is defined in eddies as different tuples might follow different routes of operators. Row-routing in Eddies presents a high opportunity for re-optimization but it contains four main drawbacks:

1. The continuous re-ordering of operators incurs a large optimization cost.
2. Eddies need to maintain query execution states, which can become expensive.
3. They rely on symmetric operators resulting in restrictiveness in the number of candidate plans considered for optimization.
4. Each tuple is routed along a greedy locally optimal path without considering the overall execution cost.

ROX, on the other hand, manages to completely interleave optimization and execution steps while avoiding the above disadvantages.

2.2.5 Database cracking

Another fundamentally different technique that is worth mentioning is database cracking [69, 70, 83]. The motivation behind this approach is to achieve a self-organizing database system by continuously learning from every executed query. Database cracking suggests to physically reorganize the database store into smaller pieces using the tuples touched

by the executed query such that the execution of subsequent queries touching the same, overlapping, or even disjoint pieces of data is speeded up. Another benefit of this approach is in its ability to adapt to different query loads. We find that database cracking and our research work share the same principle: rendering database systems and their components self-dependent. Database cracking aims at making the database system more self-organized while we aim at developing an autonomous optimizer which makes its decisions without depending on any knowledge provided by statistics and cost models. We also think that these two techniques are complementary: ROX can use the cracked tables as indexes and can adapt its optimization decisions based on whether the queried table is already cracked or not.

2.3 Conclusion

In this chapter, we began by giving a quick overview on query optimization in database systems. We then reviewed work related to the proposed ROX approach. We first focused on compile-time techniques for ordering join operators in the context of both relational and XML database systems. We then proceeded by describing run-time optimization approaches, some of which are proposed to counteract optimization errors resulting from inaccurate estimations and the inability to detect correlations.

3 Foundations and Formalization

ROX focuses on optimizing the execution order of the XPath steps and joins in an XQuery. This is achieved at run-time by interleaving optimization and execution steps. Therefore, *an order-independent representation of the join and step relationships* in the XQuery needs to be conveyed to the run-time environment of the database system, as part of the complete execution plan. We have chosen for an adapted form of a *Join Graph* as our order-independent representation. The optimal ordering of the step and join operators in the XQuery is decided based on the result size of the operators. Therefore, ROX needs a way to estimate the size of the operators in the input join graph. This is accomplished by means of *sampling-based techniques*, which are required to be supported by the underlying database system. In this chapter, we describe the foundations on which ROX builds: the join graph input to ROX, and the sampling techniques used.

3.1 Join Graphs

Join graphs have already been used in relational database systems to represent the collection of joins in a query whose execution order should be optimized. Since we have a similar purpose, we use an adapted form of join graphs to represent the XPath steps and relational joins in an XQuery. In this section, we define the notion of join graph, explain its semantics, and then describe the technique we adopt to find the join graphs of a given XQuery.

3.1.1 Join Graph Definition

We first give the definition of a join graph and then describe with more details its vertex and edge components.

In the relational context, a join graph, also referred to as query graph, is defined "to have a node for each relation mentioned in the query. And for each join operator in the query expression, for each predicate conjunct, there is an edge labeled with that conjunct" [112]. "The graph does not

impose a partial ordering on the operations" [112]. We adopt the same definition for our join graphs and tune it to fit the XQuery context.

Join Graph **Definition 3.1.1.** A Join Graph $G = (V, E)$ is defined as an edge-labeled graph where:

- a vertex $v \in V$ represents a database relation containing XML nodes or values which is input to join and path step operators in the query.
- an edge $e \in E$ represents a path step or join operator in the query.

Definition 3.1.1 is similar to the join graph definition used in the relational context. The only difference is that the vertices and edges in our new definition are made more XML specific.

Following is a more detailed definition of a vertex in a join graph.

Vertex **Definition 3.1.2.** Given a join graph $G = (V, E)$, a vertex $v \in V$ can correspond to:

1. the root of the XML document *doc.xml*. The vertex v is then annotated with the string $\underset{doc.xml}{root}$. The relation associated with v is a singleton containing the tuple in the database that corresponds to the document's root. Note that the vertex v may as well represent the roots of a document collection, and therefore the relation associated with v consists of the root nodes of the documents in the collection.
2. an XML element type with qualified name *qname*. The vertex v is then annotated with the qualified name *qname*. The relation associated with v contains all the XML element nodes in the queried document(s) that have the qualified name *qname*.
3. an XML text node. The vertex v is then annotated with the string *text*(). The relation associated with v contains all XML text nodes in the queried document(s)
4. an XML text node with an equality or range-selection predicate *pred*. The vertex v is then annotated with the string $\underset{pred}{text()}$. The relation associated with v contains those text nodes in the queried document(s) with a value satisfying the *pred* condition.
5. an XML attribute node with name *attr*. The vertex v is then annotated with the string *@attr*. The relation associated with v contains all XML attribute nodes in the queried document(s) with *attr* as name.
6. an XML attribute node with name *attr* and an equality or range-selection predicate *pred*. The vertex v is then annotated with the string $\underset{pred}{@attr}$. The relation associated with v contains all XML attribute nodes in the queried document(s) with a value satisfying the *pred* condition.

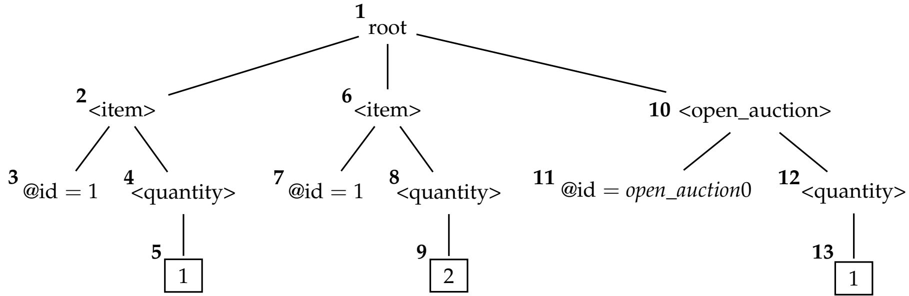


Figure 3.1 The XML tree of an XML document (xmark.xml) inspired by the XMark benchmark [4]. The nodes between tags (<>) correspond to XML elements. The attribute nodes are represented by the @ symbol followed by the name and value of the attribute. The nodes surrounded by a rectangle are XML text nodes. Each node is associated with a node identifier number (bold number left top of the node) that is generated by a pre-order traversal of the tree.

7. a pre-materialized table representing the result of a subexpression, and containing any type of XML node or value. We give an example and further explain this vertex type in Example 3.1.7.

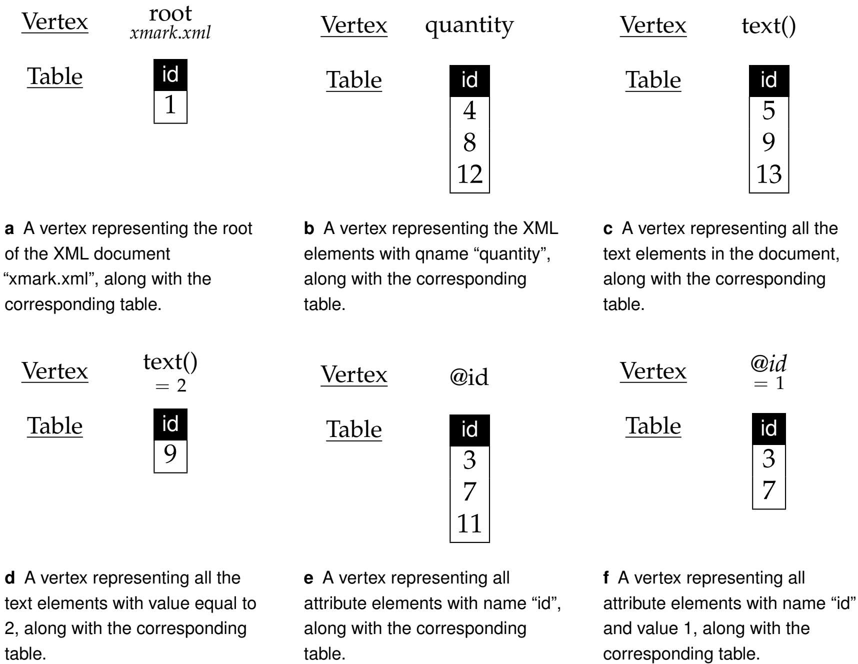
Example 3.1.3. In Figure 3.2, we illustrate the different types of vertices using the document *xmark.xml*, the corresponding XML tree of which is shown in Figure 3.1. The document is inspired by the XMark benchmark [4], and its content is represented in the XML tree. The nodes in the tree surrounded by tags (<>) correspond to XML elements. The attribute nodes are represented by the @ symbol followed by the name and value of the attribute. The nodes surrounded by a rectangle represent XML text nodes and contain the value of the node. The number associated to each node in the tree (bold number at the node's upper left corner) is the node identifier generated by a pre-order traversal of the tree.
Figure 3.2 illustrates the different types of vertices listed in Definition 3.1.2. Along with each vertex v is the table associated with v. The contents of the table consist of the node identifier of the nodes in the XML tree that match the vertex.

Now that we have a better insight in what a vertex represents, we give a more detailed explanation of an edge in the join graph.

Definition 3.1.4. Given a join graph $G = (V, E)$, an edge $e = (v, v') \in V$ with label lb might represent:

Edge

1. an XPath step that computes a step join between the tables associated with the two vertices v and v' along the axis lb. The value of



a A vertex representing the root of the XML document "xmark.xml", along with the corresponding table.

b A vertex representing the XML elements with qname "quantity", along with the corresponding table.

c A vertex representing all the text elements in the document, along with the corresponding table.

d A vertex representing all the text elements with value equal to 2, along with the corresponding table.

e A vertex representing all attribute elements with name "id", along with the corresponding table.

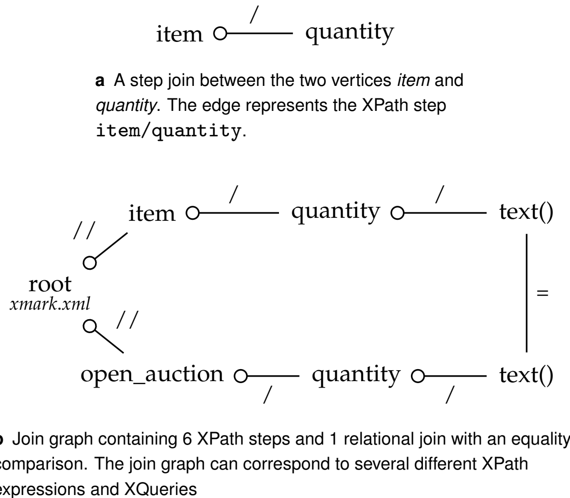
f A vertex representing all attribute elements with name "id" and value 1, along with the corresponding table.

Figure 3.2 The different types of vertices a join graph can contain.

the label *lb* can be any of the XPath axes: $lb \in$ {`child`, `descendant`, `descendant-or-self`, `parent`, `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, `preceding-sibling`, `self`}. In a join graph, an XPath step edge is depicted as $\circ\overset{lb}{\text{——}}$ where the label *lb* represents the axis along which the XPath step is interpreted, and the circle $\circ$ denotes the context set of the step. For instance, the edge `v` $\circ\overset{ax}{\text{——}}$ `v'` stands for the XPath step `v/`*ax*`::v'`.

2. a relational join which according to the XQuery semantics computes a join using a value-based comparison of the tables associated with both inputs v and v'. In this case, the label *lb* can be any of the following: $lb \in \{=, \neq, <, \leq, >, \geq\}$. Typically, the input vertices v and v' of such joins are text- or attribute-nodes. We stress that this join operator is a normal relational join, and does not perform the existential comparison defined in XQuery. A relational join is depicted as `v` $\overset{lb}{\text{——}}$ `v'`, and it stands for the comparison `v` *lb* `v'`.

Example 3.1.5. In Figure 3.3, we illustrate the different types of edges. Figure 3.3a depicts a step join between the *item* and *quantity* vertices. The



a A step join between the two vertices *item* and *quantity*. The edge represents the XPath step `item/quantity`.

b Join graph containing 6 XPath steps and 1 relational join with an equality comparison. The join graph can correspond to several different XPath expressions and XQueries

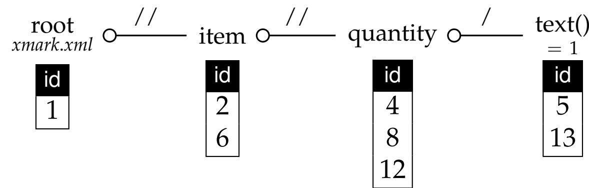
Figure 3.3 Edges in a join graph can represent XPath steps and relational joins.

edge `item` $\circ\overset{/}{—}$ `quantity` represents the XPath step `item/quantity`. Figure 3.3b shows a bigger join graph containing 6 step joins and 1 relational join with an equality comparison. This join graph can correspond to several different XPath expressions. One possible XPath expression is `doc(xmark.xml)//item/quantity[./text()=doc(xmark.xml) //open_auction/quantity/text()]`. The join graph can also represent several different XQueries. An example of another corresponding XQuery is the following:

```
for $i in doc("xmark.xml")//item,
    $q in doc("xmark.xml")//open_auction/quantity
where $i/quantity/text() = $q/text()
return $i
```

Note that although the above two queries are not equivalent (they do not return the same result), their corresponding join graph is the same. In fact, as we will see in Section 3.1.3, the join graph is only one part of the execution plan. Other operators in the plan process the output generated by the XPath steps and joins in the graph, and generate the right result for the given query.

Now that we have defined the notion of join graph, and explained in details the type of vertices and edges it can contain, we next describe the semantics of the join graph and the execution of its edges.



a A join graph consisting of 4 vertices. The content of the tables associated with the vertices is shown.

item	quantity	text()=1
2	4	5

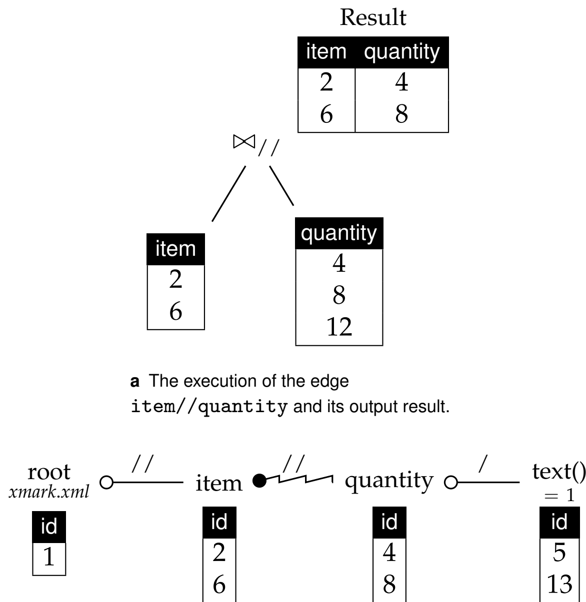
b The fully joined relation resulting from the execution of the edges in the join graph.

Figure 3.4 A join graph and its fully joined relation resulting from the execution of its edges.

3.1.2 Semantics of the Join Graph and Execution of Edges

The semantics ("result") of a join graph is a fully joined relation containing attributes from the tables associated with its vertices. The relation is the result of the subsequent executions of the edges (steps and joins) in the join graph. Figure 3.4 depicts an example join graph and the fully joined relation resulting from the execution of its edges. We next describe the execution of the edges, and the derivation of the fully joined relation.

Example 3.1.6. Figure 3.5a illustrates the execution of the edge `item//quantity` along with the result of the execution. The queried XML document is "xmark.xml" with the XML tree shown in Figure 3.1. The resulting join graph after the execution of the edge is shown in Figure 3.5b. The executed step is represented with a saw-shaped edge (●⌐⌐). As can be seen in the join graph, the table resulting from the execution of the operator is vertically partitioned, and each partition is associated with its corresponding vertex. Therefore, the content of the tables associated with the vertices `item` and `quantity` is updated with the result of the execution of the edge `item//quantity` . We stress that the partitioning of the result relation is only logical, and is only performed for representational reasons. The two partitioned tables are in fact positionally aligned in one single relation.
Figure 3.6a shows the execution and the output result of the second edge `quantity/text()[.=1]`. The left input to the step join is the table associated with the vertex `quantity` which, as a result of the previous execution and as visible in the figure, is positionally aligned with the table associated with the vertex `item`. Therefore, the result of the execution of the XPath



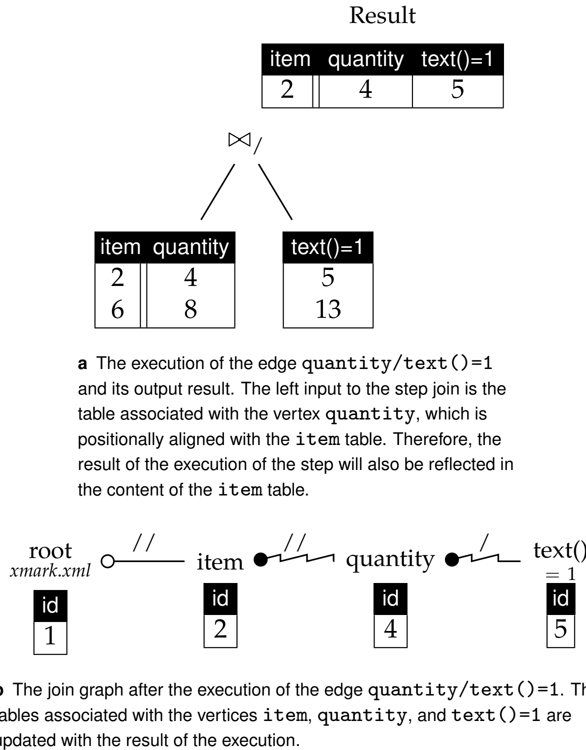
a The execution of the edge `item//quantity` and its output result.

b The join graph after the execution of the edge `item//quantity`. The saw-shaped edge represents the executed edge. The table resulting from the execution is vertically partitioned, and each partition is associated to its corresponding vertex. The partitioning is only logical, and performed only for representational reasons.

Figure 3.5 The execution of the edge `item//quantity` of the join graph of Figure 3.4a.

step will also be reflected in the table associated with the vertex `item`, as can be seen in the output relation. The join graph resulting from the step execution is depicted in Figure 3.6b. The content of the tables associated with the vertices `item`, `quantity` and `text()=1` are updated with the result of the execution. Note that the output relation depicted in Figure 3.6a matches the fully joined relation presented in Figure 3.4b. Also note that the edge `doc(xmark.xml)//item` is not executed. In fact, there is no need to process this edge since any `item` node in the document is certainly a descendant of the document's root. In case the edge was labeled with a child axis, it would have been necessary to execute the step to filter out all the `item` nodes that are not a direct child of the root node.

In summary, we have shown in the example that the execution of an edge in a given join graph results in combining the tables associated



a The execution of the edge `quantity/text()=1` and its output result. The left input to the step join is the table associated with the vertex `quantity`, which is positionally aligned with the `item` table. Therefore, the result of the execution of the step will also be reflected in the content of the `item` table.

b The join graph after the execution of the edge `quantity/text()=1`. The tables associated with the vertices `item`, `quantity`, and `text()=1` are updated with the result of the execution.

Figure 3.6 The execution of the edge `quantity/text()=1` of the join graph of Figure 3.4a.

with the vertices of the edge. The resulting relation is represented as vertically partitioned tables, each associated to its corresponding vertex. The partitioning is only logical, therefore any subsequent execution affects the content of all the partitions. The result of the join graph is the fully joined relation resulting from the execution of all its edges. We note that subsequent projections, as described in the next section, specify which part of the fully joined relation is of interest. We note that for optimization purposes, these projections can be pushed into the join graph.

In the next section, we explain the method we adopt for finding the join graphs of a given XQuery.

3.1.3 Join Graph Isolation

A join graph, input to the ROX algorithm, is an order-independent collection of XPath steps and relational join operators in a given XQuery. To

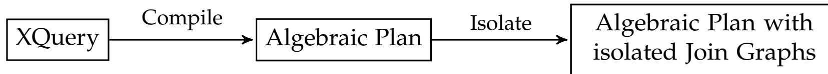


Figure 3.7 Process of finding the join graph corresponding to a given XQuery.

find the join graph of an XQuery, we use the existing technique named *Join Graph Isolation*, explained in [59, 60]. This section gives a brief description of the adopted technique.

The join graph isolation process is currently implemented in *Pathfinder*. Pathfinder is a full-fledged compiler for the complete XQuery language, targeting relational database back-ends. We use Pathfinder as a representative for XML database systems that support a relational algebra. Figure 3.7 depicts the process of finding the join graph corresponding to a given XQuery. It consists of two stages; a plan compilation step followed by a *join graph isolation* phase:

- **Plan Compilation:** In this phase, the input XQuery is compiled into a DAG-shaped plan of algebraic relational operators. A peep-hole driven optimization, described in [56], rewrites the DAG into an equivalent more efficient and simple form. This simplification process uses compositional rewriting rules, the applicability of which is decided based on the properties of operators in the plans. Therefore, the application of the peep-hole-style equivalence rules is preceded by a phase that traverses the DAG multiple times to infer the properties of the operators. Given the XQuery Q_1 shown in Figure 3.8a, the plan resulting from the compilation and peep-hole optimization of Q_1 in Pathfinder is depicted in Figure 3.8b (the reader is not expected to fully understand the plan and the used operators). The semantics of the operators used in the DAG are listed in Figure 3.9.
- **Join Graph Isolation:** The join graph isolation step rewrites the DAG-shaped plan generated by the previous compilation phase, such that step and join operators are grouped together into one cluster. In fact, as a result of the nesting of `for` loops, and of the conditional expressions in an XQuery, *joins*, *numbering* and *distinct* operators are usually scattered all over the DAG-shaped plan. The numerous occurrences of *numbering* operators between the step joins restrict the possibilities of join reordering, since the correct execution of the step joins depends on the columns introduced by these *numbering* operators. In the plan of Figure 3.8b, the step join $\rtimes_{item/age}$, the most selective join in the plan, cannot be pushed below the other two step joins, because of the presence of the attach operator $\#_{iter}$. The join graph isolation process takes care of pushing the blocking operators higher in the plan, and of moving the joins down in the

```
for $p in doc("xmark.xml")//person/profile
where $p/age
return $p
```

a XQuery Q_1.

b The DAG-shaped plan resulting from the compilation and peep-hole optimization of Q_1.

c The DAG resulting from the join graph isolation step.

Figure 3.8 The process of finding the join graph corresponding to the XQuery Q_1.

plan, such that a cluster, as large as possible, of selection, projection, and join operators is created. The boundaries of those sections are then detected and the clusters are replaced by corresponding Join Graphs. Doing so seems easy, but in reality requires a finely tuned set of optimization rules [59, 60]. The plan after join graph isolation

Operator	Semantics
$\bowtie_p$	relational join with predicate p
$\bowtie_{step}$	XPath step executing the step *step*, returns the new node in column *pre*
$\pi_{a_1:b_1,\ldots,a_n:b_n}$	project onto columns b_i, rename b_i into a_i
$\#_a$	attach unique row id in column a
δ	duplicate removal
$\text{♀}_{a_1,a_2}$	serialize column a_1 by order in a_2

Figure 3.9 Algebraic operators used by the *Pathfinder* system.

is shown in 3.8c. The operators within the boundaries of the oval define the join graph of the XQuery. The remaining operators in the plan form a tail to the join graph. Figure 3.10 shows the execution plan generated for query Q_1 including the corresponding join graph (within the oval boundaries) and tail. The execution plan with the embedded join graph 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 graph.

For most XQuery queries, a tail of operator is formed as a result of the join graph isolation process. The tail usually consists of *Project*, *Sort*, and *Distinct* operators. The functionality of the latter two is to ensure that the order and distinctness semantics specified in the XQuery query are preserved. The tail of the join graph shown in Figure 3.10 contains a *Project* operator, a *Distinct* operator, and a *Serialize* operator that performs the required result sorting. 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.

Occasionally for certain queries, some operator constructs, such as aggregation computation and element construction, separating two or more groups of steps and joins cannot be pushed by the join graph isolation process below or above the clusters. This results in an execution plan containing multiple isolated join graphs connected by the blocking operators. ROX will then optimize each of the different join graph sub-plans separately. Following is an example XQuery, the generated execution plan of which, contains two separate join graphs.

Example 3.1.7. Consider the following XQuery Q_2:

```
for $a in doc("xmark.xml")//open_auction
```

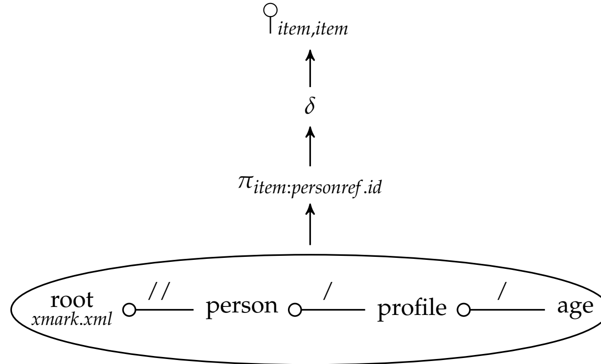


Figure 3.10 The generated execution plan of Q_1 including the join graph (inside the circle) and tail corresponding to the query. The join graph provides the optimizer with a description of the *step and join relationships* in the input query *without prescribing a particular execution order* of these operators.

```
where count($a//bidder) = 0 and
      $a//reserve
return $a/@id
```

Its corresponding execution plan generated by the *Pathfinder* compiler is shown in Figure 3.11. The execution plan contains two join graphs separated by *Project*, *Count* and *Select* operators. ROX will optimize each join graph separately, starting with the bottom one.

We recall the last type of vertex presented in Definition 3.1.2: a vertex can correspond to "a pre-materialized table containing any type of XML node, generated from a previous execution of a sequence of operators." The top join graph in Figure 3.11 contains an example of such a vertex. The vertex π_{item} corresponds to the table of XML nodes that results from the execution of the sub-plan of operators rooted at the project operator π_{item}.

3.1.4 Wrap-up

In this section, we have introduced the join graph concept. We wrap up our previous description by emphasizing the following points:

- **An order-free representation of step and relational joins:** The join graphs embedded in the generated execution plans provide the optimizer with a description of the *step and join relationships* in the input query *without prescribing a particular execution order* of these operators. This allows to effectively defer to run-time any decisions on the execution order of the joins and steps in the plan.

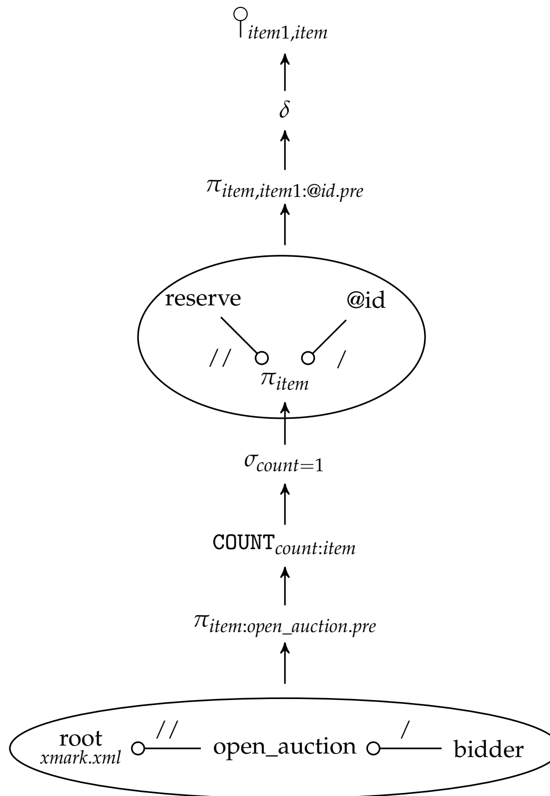


Figure 3.11 The generated execution plan of query Q_2. It includes two join graphs separated by *Project*, *Count* and *Select* operators. ROX will optimize each join graph separately, starting with the bottom one and then optimizing the top graph the last.

- **Support the entire XQuery language:** The possibility to handle multiple join graphs contained in one execution plan allows ROX to *support the entire XQuery language* while focusing on the optimization of the crucial order of relational joins and step operators. We also stress that the fragment of the XQuery language that can be mapped to an execution query with a single join graph is more expressive than the twig queries widely considered in other previous work [26, 35, 129, 33, 65].
- **Seamless handling of step and relational joins:** The fact that XPath steps and relational joins co-exist together in the same join graph gives ROX the possibility to *optimize seamlessly the execution order of these two different types of operators*. Based on our knowledge this is not supported by any previous work where the reordering of XPath

steps is performed separately without taking into consideration the relational joins in the query.

3.2 Sampling Techniques

Given an input join graph, the objective of the ROX algorithm is to find a good execution order of the XPath steps and relational join operators in the graph. Generally speaking, given a set of steps and relational joins, an optimizer needs to acquire knowledge about the selectivity of each of the operators to determine a good execution order for these operators. Required to have no dependency on any a priori collected statistics and cost model, ROX resorts to using sampling techniques to estimate the selectivity of the operators in the join graph. For the usage of the sampling techniques to be rewarding, sampling should be cheap, efficient, as well as result in accurate estimations.

In this section, we start by describing the adopted sampling approach and the technique for estimating the result size of joins. We then introduce and explain our cutoff-sampling technique, the purpose of which is to keep the cost of the sampling operations under control. We finally stress some important points about the sampling techniques used in the ROX prototype. We draw the attention of the reader to the fact that in the following the term join and the symbol $\bowtie$ are used to refer to both XPath steps and relational joins.

3.2.1 The Sampling Operation

ROX uses sampling techniques in two situations. The first is when a *table* is sampled to construct a set of tuples randomly chosen from the table. The second is when a *join* is sampled to estimate the result size of an XPath step or a relational join in the join graph. The sampling operation is denoted with the symbol $\triangleright$. It takes as input, among others, an integer value `LIMIT` that defines an upper bound on the number of tuples to be returned in the sampling output. In the following, we describe in more detail the sampling of tables and joins, and the technique used to estimate the result size of joins.

Sampling From Tables

In some situations, ROX will need to sample a table to construct a set of tuples randomly chosen from the given table. In this case, the sampling operation takes as input the table T from which the tuples are selected, and the integer value `LIMIT`. The sampling operation $\triangleright_{\texttt{LIMIT}}(T)$ will randomly

pick from T a number of tuples equal to LIMIT. Next, we give a formal definition of the sampling operation from tables.

Definition 3.2.1.

$$\begin{aligned} \rhd \quad &: \quad \mathit{Table} \times \mathbb{N} \rightarrow \mathit{Table} \\ \rhd_{\texttt{LIMIT}}(T) \quad &= \quad S \\ &\textbf{where} \begin{cases} S \text{ is a table containing tuples randomly selected from } T \\ |S| \leq \texttt{LIMIT} \end{cases} \end{aligned}$$

The inequality $|S| \leq \texttt{LIMIT}$ is introduced in the above definition to handle the case in which the sampling operation cannot return a LIMIT number of tuples. This can occur if the size of the table T is less than LIMIT.

Sampling Joins

ROX also samples joins to estimate the result size of XPath steps and relational joins in the join graph. For the estimations to be accurate, sampling should return a *representative* subset of the output of the sampled operator. In other words, sampling the join $R \bowtie T$ consists of generating the set $S = \rhd_{\texttt{LIMIT}}(R \bowtie T)$, such that S is a representative subset of the output of the join $R \bowtie T$.

Obviously, for efficiency reasons, the sampling result S should be generated without first computing the full join. Therefore, to sample the join operator, we need to execute the operator with a sample of its input, meaning that the sample operation $\rhd$ is applied to the input of the join instead of the output of the join itself.

A naive approach would apply the sample operation $\rhd$ to the two inputs of the join; however, it is known from literature [29] that a join of two random samples chosen from the operands of the join does not result in a representative sample of the output of the join, *i.e.* $\rhd_{\texttt{r}}(R) \bowtie \rhd_{\texttt{t}}(T)$ does not generate a sample that is as representative as the sample produced by the operation $\rhd_{\texttt{LIMIT}}(R \bowtie T)$.

Therefore, the sample operator $\rhd$ is usually applied to only one of the join's inputs, as depicted in Figure 3.12. The figure illustrates two ways to sample the join $R \bowtie T$: by evaluating one of the following two expressions: $(\rhd_{\texttt{LIMIT}}(R) \bowtie T)$ or $(\rhd_{\texttt{LIMIT}}(T) \bowtie R)$. The first one refers to the execution of the join using as input a sample chosen from the left input table R (sampling method 1), while the latter corresponds to the case in which the sample operation $\rhd$ is applied to the right operand of the join (sampling method 2).

Next, we give a definition of the sampling operation of a join.

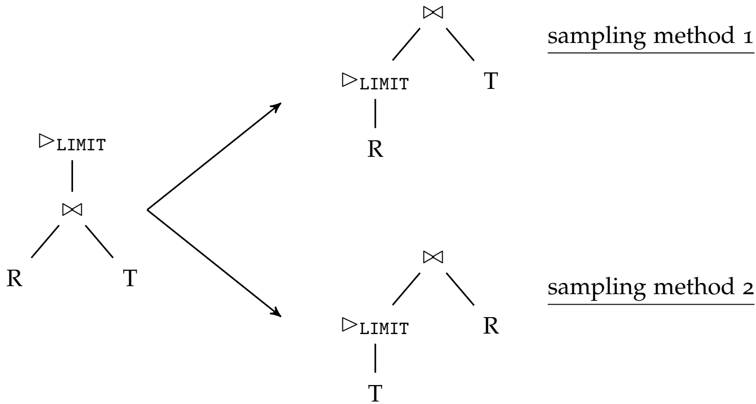


Figure 3.12 Two ways to sample the join $R \bowtie T$ without fully executing the join first: the sample operation $\rhd$ is applied to one of the join's input tables.

Definition 3.2.2. Given the join $R \bowtie T$, sampling the join can be performed using one of the following methods:

$$\rhd_{\texttt{LIMIT}}(R \bowtie T) = \rhd_{\texttt{LIMIT}}(R) \bowtie T \qquad \underline{\text{sampling method 1}}$$

$$\rhd_{\texttt{LIMIT}}(R \bowtie T) = \rhd_{\texttt{LIMIT}}(T) \bowtie R \qquad \underline{\text{sampling method 2}}$$

Obviously, as the result of a sampling operation depends on the tuples in its sampled input, the above two sampling methods will most probably produce two different result relations. The choice from which of the join's operands (R or T) to pick the sample can depend on several criteria. One possibility is to choose the sample from the smallest input, since in general picking a set of tuples from a smaller table results in a more representative input sample, which yields to a more representative output of the join's result, and consequently to a more accurate estimation of the join's result size. Another factor takes into consideration the cost of the join sampling operation, and tries to keep it as cheap as possible. For instance, if an index is built on only one of the two operands, say T, then it might be better to pick the sample from R, and use an index-based join to compute the join between the sample set and T.

Estimating the Result Size of a Join

The relation generated by the sampling operation of a join is used to estimate the result size of the join operator. In the following, we explain

the method adopted by ROX to estimate the cardinality of a join. For that purpose, we first make the following assumption and define the *hit ratio* of a join.

Assumption 3.2.3. Given two relations T and t where $t = \triangleright_{\texttt{LIMIT}}(T)$, we assume that the tuples in t are a representative sample of the tuples in the full table T.

Definition 3.2.4. We define the hit ratio of a join as the following function:

$$\begin{array}{lcl} HR & : & Join \rightarrow \mathbb{R}^+ \\ HR(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) & = & \dfrac{|S|}{|R_1|} \\ & \textbf{where} & S = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \end{array}$$

It can be observed from the above definition that the hit ratio of a join is defined with respect to its left operand. We therefore conclude that $HR(R \bowtie T) \neq HR(T \bowtie R)$.

We also note that given the join $R \bowtie T$, the size of its result S can be inferred using the above formula if its hit ratio hr is known: $|S| = |R| \times hr$. Therefore, to estimate the result size of a join through sampling, we first need to derive the join's hit ratio.

Suppose that the join $R \bowtie T$ is sampled using a sample from its left operand, that is $S = r \bowtie T$ where $r = \triangleright_{\texttt{LIMIT}}(R)$. The hit ratio hr of the sampled join is: $hr = HR(r \bowtie T) = \frac{|S|}{|r|}$. Using Assumption 3.2.3, the hit ratio of the full join $R \bowtie T$ can be linearly extrapolated from the hit ratio of the sampled join, and therefore we have $HR(R \bowtie T) = hr$.

Now that we have described the derivation of the hit ratio of a join through sampling, we define the cardinality estimation of the join's result. The following definition will be refined in Section 3.2.2 after introducing the cutoff-sampling technique.

Definition 3.2.5. Given the join $R \bowtie T$, and supposing that the join is sampled using the expression $\triangleright_{\texttt{LIMIT}}(R) \bowtie T$ (*i.e.* sampling method 1), the estimated cardinality of its result is computed as follows:

$$\begin{array}{lcl} card & : & Join \rightarrow \mathbb{R}^+ \\ card(R \bowtie T) & = & |R| \times hr \\ & \textbf{where} & \begin{cases} hr = HR(r \bowtie T) \\ r = \triangleright_{\texttt{LIMIT}}(R) \end{cases} \end{array}$$

The above definition estimates the result size of the join $R \bowtie T$ while sampling the join using the sampling method 1. Obviously, it is also

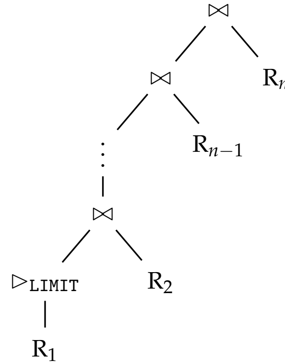


Figure 3.13 Sampling the sequence of joins $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. A set of tuples is randomly chosen from the relation R_1 and then joined with the other $(n-1)$ tables. Obviously, as explained earlier, the sample set can also be picked from any of the $(n-1)$ input relations $R_2, R_3, \ldots,$ or R_n.

possible to estimate the cardinality of the join with the sampling method 2, *i.e.* picking the input sample from the relation T ($\rhd_{\texttt{LIMIT}}(T) \bowtie R$). In that situation, the result size is computed as follows: $card(R \bowtie T) = |T| \times hr$ where $hr = HR(\rhd_{\texttt{LIMIT}}(T) \bowtie R)$.

Example 3.2.6. Suppose that the size of the join $R \bowtie T$ is estimated using the sampling operation $S = r \bowtie T$ where $r = \rhd_{100}(R)$. Let $|R| = 5000$ and $|S| = 400$. Using the formula in Definition 3.2.4, the hit ratio of the sampled join $r \bowtie T$ is estimated to be equal to $\frac{400}{100} = 4$. Using linear extrapolation (Assumption 3.2.3), we infer that $HR(R \bowtie T) = 4$. The result size of the join is then estimated to be $card(R \bowtie T) = 5000 \times 4 = 20000$.

Sampling a Sequence of Joins

To sample a sequence of joins, ROX adopts the same approach used to sample a single join. Figure 3.13 depicts the sampling operation of the sequence $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. As in the case of sampling a single join, a sample set chosen from one of the input tables is joined with the other $(n-1)$ tables. In the plan illustrated in the figure, the sample is picked from the relation R_1, but as we have explained earlier, the sample set may as well be chosen from any of the other $(n-1)$ input tables $R_2, R_3, \ldots,$ or R_n.

Definition 3.2.5 can be generalized to a tree of join operators, and then used to estimate the result size of a sequence of joins. The hit ratio hr of the sampled join sequence $S = \rhd_{\texttt{LIMIT}}(R_1) \bowtie R_2 \bowtie \ldots \bowtie R_n$ is equal to

$\frac{|S|}{|r_1|}$ where $r_1 = \rhd_{\texttt{LIMIT}}(R_1)$. The hit ratio hr is linearly extrapolated to the full join $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, and the estimated cardinality of the result of the join sequence is then computed as follows $card(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) = |R_1| \times hr$.

3.2.2 Cutoff-sampling

In the previous section, we stressed that sampling a join consists of joining a sample randomly chosen from one of the join's operands with the full table of the other operand. Although the size of the input sample is usually small, the result of the join sampling operation might be large in case the join hit ratio is high. In fact, the worst case complexity of join operators is quadratic: they can in principle return the Cartesian product of their two inputs. Though this certainly is not typically the case, our run-time optimizer should guarantee efficiency against the possible occurrence of high join hit ratios which blow up the size of the result of a sampling operation. To achieve this, ROX uses *cutoff-sampling* which ensures that the size of the output of a sampling operation does not exceed a pre-defined number of tuples.

Cutoff-sampling, denoted by $\trianglerighteq$, takes as input a join operator $\bowtie$, the two tables R, T to be joined, and an integer value `LIMIT`. The operation $\trianglerighteq_{\texttt{LIMIT}}(R \bowtie T)$ executes the join $(R \bowtie T)$, and returns from the execution result a number of tuples equal to at most `LIMIT`. Cutoff-sampling does this in one step: rather than producing the full result of $(R \bowtie T)$ and then reducing it to *LIMIT*, cutoff-sampling is integrated into the operator. That is the execution of $(R \bowtie T)$ is terminated when the number of output tuples reaches the `LIMIT` value, thus cutting-off early the generation of results.

By processing only a fraction of the tuples in its input sample and limiting the generation of results, the introduced cutoff-sampling technique succeeds in keeping the cost of sampling under control and the result size of these operations within usable range; however, it might suffer from generating less representative result sets. We will further explain this point in Section 5.2 and present possible solutions.

Example 3.2.7. Suppose that the join $R \bowtie T$ is sampled with the operation $\trianglerighteq_{100}(r \bowtie T)$, where r is a sample randomly chosen from R ($r = \rhd_{100}(R)$ and $|r| = 100$). Cutoff-sampling will partially execute the join $r \bowtie T$ until the size of the output reaches 100. Therefore, only a fraction of tuples in r necessary to produce the required 100 tuples is consumed. If we suppose that the join hit ratio is equal to 5, and that it is uniform across the tuples in r, then the total number of tuples consumed from r is around 20, which

means that only a fraction of 0.2 of the r tuples will be processed during the sampling operation.

We observe from the previous example, that to sample the join $R \bowtie T$ the following two steps should be applied:

1. The input relation R is first sampled to construct the sample r with size `sample-size`: $r = \triangleright_{\texttt{sample-size}}(R)$
2. The sample r is joined with the other input relation T using cutoff-sampling and the cutoff `LIMIT`: $\trianglerighteq_{\texttt{LIMIT}}(r \bowtie T)$

In the previous example, we had `sample-size` = `LIMIT` = 100. Obviously, it is also possible to sample the join $R \bowtie T$ by picking a sample from the table T, and cutoff-sampling the join as follows $\trianglerighteq_{\texttt{LIMIT}}(\triangleright_{\texttt{sample-size}}(T) \bowtie R)$. As explained in Section 3.2.1, the choice from which table to pick the sample depends on several criteria.

Although cutoff-sampling is explained in the context of joins, we note that the introduced technique might as well be used with other operators. Generally speaking, cutoff-sampling an operator *op* with cutoff value `LIMIT` consists of partially executing *op* until the size of the generated result reaches `LIMIT`.

Estimating the Result Size of a Join

We know from Definition 3.2.5 that to estimate the result size of the join $R \bowtie T$ in the previous example, the hit ratio of its sampled join $r \bowtie T$ should be first derived. However, the cutoff-sampling operation $\trianglerighteq_{100}(r \bowtie T)$ partially executed the join $r \bowtie T$, and generated the output table S, while processing a fraction of only 0.2 of the tuples in r. Since 80% of the r tuples has not been evaluated, it is not possible to use the size of the output S (which is equal to the pre-defined limit 100) to estimate the hit ratio of the sampled join $r \bowtie T$ as described in Definition 3.2.4.

As a result, the estimation of the hit ratio of the join $r \bowtie T$ is integrated into the cutoff-sampling operation, *i.e.* cutoff-sampling will compute and subsequently return the hit ratio of its input join. To derive the hit ratio of $r \bowtie T$, cutoff-sampling takes note of the number n of processed tuples in r, and then uses it to compute an estimate of the hit ratio hr as $hr = \frac{|S|}{n}$. Let A be the set of n processed tuples from r. Assuming that A is a representative sample of the tuples in r (Assumption 3.2.3), it is possible to linearly extrapolate the join hit ratio hr to the cutoff-sampled join $r \bowtie T$. Then assuming that the set r is a representative sample of the tuples in R (Assumption 3.2.3), it is possible to linearly extrapolate the join hit ratio hr to the full join $R \bowtie T$.

Two linear extrapolations have been applied here. The first one goes from the sample A to the set r, while the second applies from r to R. We

note that the method used by cutoff-sampling to pick the tuples to be in A and the sampling technique used to construct the sample r might very well be different. We also stress that because of the two consecutive linear extrapolations, and depending on the method used to choose the tuples in A, cutoff-sampling might generate less representative result sets. We will further explain this point in Section 5.2 and present possible solutions.

We now formally define the cutoff-sampling operator $\trianglerighteq$.

Definition 3.2.8.

$$
\begin{aligned}
\trianglerighteq \quad &: \quad \mathit{Join} \times \mathbb{N} \rightarrow \mathit{Table} \times \mathbb{R}^{+} \\
\trianglerighteq_{\texttt{LIMIT}}(r \bowtie T) \quad &= \quad \{S, hr\} \\
&\textbf{where} \begin{cases} S \text{ is the result of the cutoff execution of the} \\ \text{operator } (r \bowtie T) \\ S = \triangleright_{\texttt{LIMIT}}(r \bowtie T) \wedge |S| \leq \texttt{LIMIT} \\ hr = HR(r \bowtie T) = \frac{|S|}{n} \\ n \text{ is the number of processed tuples from } r \end{cases}
\end{aligned}
$$

The inequality $|S| \leq \texttt{LIMIT}$ is introduced in the above definition to handle the case in which the cutoff-sampling operation cannot return a LIMIT number of tuples. This can occur if the total number of tuples that originally match the join operation $r \bowtie T$ is less than LIMIT.

To make it more suitable to the introduced cutoff-sampling technique, we now refine Definition 3.2.5 which formulates the process of estimating the result size of a join.

Definition 3.2.9. Given the join $R \bowtie T$, and assuming that the join is cutoff-sampled using the expression $\trianglerighteq_{\texttt{LIMIT}}(\triangleright_{\texttt{sample-size}}(R) \bowtie T)$, the cardinality of its result is estimated using the following formula:

$$
\begin{aligned}
\mathit{card}(R \bowtie T) = &|R| \times hr \\
&\textbf{where} \begin{cases} (S, hr) = \trianglerighteq_{\texttt{LIMIT}}(r \bowtie T) \\ r = \triangleright_{\texttt{sample-size}}(R) \end{cases}
\end{aligned}
$$

We stress again that it is also possible to estimate the result size of the join $R \bowtie T$ by picking a sample from the table T, and cutoff-sampling the join as follows $\trianglerighteq_{\texttt{LIMIT}}(\triangleright_{\texttt{sample-size}}(T) \bowtie R)$.

Continuation of Example 3.2.7. In Example 3.2.7, we sampled the join $R \bowtie T$ using the operation $(S, hr) = \trianglerighteq_{100}(r \bowtie T)$, where r is a sample randomly chosen from R ($r = \triangleright_{100}(R)$ and $|r| = 100$). We know that

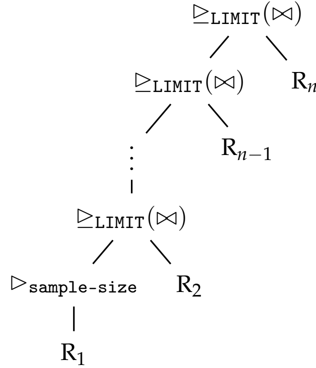


Figure 3.14 Cutoff-sampling the sequence of joins $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. A set of tuples is randomly chosen from the relation R_1 and then joined with the other $(n-1)$ tables $R_2, R_3, \ldots, R_n$. The joins are executed with the cutoff LIMIT. Obviously, as explained earlier, the sample set can also be chosen from any of the other $(n-1)$ input relations R_2, $R_3, \ldots$, or R_n.

$|S| = 100$ and that only 20 of the r tuples have been processed. Cutoff-sampling exploits the number n of processed tuples to estimate the hit ratio (hr) of the join $r \bowtie T$, by computing the formula shown in Definition 3.2.8: $hr = \frac{100}{20} = 5$. Let $|R| = 3000$, ROX can now compute the cardinality of the full join $R \bowtie T$ as shown in Definition 3.2.9: $card(R \bowtie T) = 3000 \times 5 = 15000$.

Cutoff-sampling for a Sequence of Joins

When sampling a sequence of joins, ROX also uses cutoff-sampling to limit the sampling cost. Figure 3.14 illustrates the cutoff-sampling of the join sequence $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$. A set of tuples is chosen from the relation R_1 and then joined with the other $(n-1)$ input tables $R_2, R_3, \ldots$ and R_n. The $(n-1)$ joins are cutoff-sampled with the operations shown below:

$$\begin{aligned}
(S_2, hr_1) &\leftarrow \trianglerighteq_{\texttt{LIMIT}}(\triangleright_{\texttt{sample-size}}(R_1) \bowtie R_2) \\
(S_3, hr_2) &\leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_2 \bowtie R_3) \\
&\vdots \\
(S_{n-1}, hr_{n-2}) &\leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_{n-2} \bowtie R_{n-1}) \\
(S_n, hr_{n-1}) &\leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_{n-1} \bowtie R_n)
\end{aligned}$$

As we have mentioned earlier, the sample set can as well be chosen from any of the other $(n-1)$ input relations $R_2, R_3, \ldots$, or R_n.

To estimate the result size of the join sequence $(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$,

we first need to derive the join hit ratio of the sequence. We claim the following:

Lemma 3.2.10. *Suppose the join sequence $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ is cutoff-sampled by executing the following operations:*

$$\begin{array}{c} (S_2, hr_1) \leftarrow \trianglerighteq_{LIMIT}(\triangleright_{sample\text{-}size}(R_1) \bowtie R_2) \\ (S_3, hr_2) \leftarrow \trianglerighteq_{LIMIT}(S_2 \bowtie R_3) \\ \vdots \\ (S_{n-1}, hr_{n-2}) \leftarrow \trianglerighteq_{LIMIT}(S_{n-2} \bowtie R_{n-1}) \\ (S_n, hr_{n-1}) \leftarrow \trianglerighteq_{LIMIT}(S_{n-1} \bowtie R_n) \end{array}$$

We claim that $HR(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) \approx \prod_{i=1}^{n-1}(hr_i)$

In the following, we prove the above lemma. The proof is given with $n = 3$, and can be generalized to all values of n ($n \in \mathbb{N}^+$).

Proof. Given the join sequence $R \bowtie T \bowtie Q$, it is cutoff-sampled by executing the following two operations:

$$(S_1, hr_1) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(r \bowtie T) \quad (3.1)$$

$$(S_2, hr_2) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_1 \bowtie Q) \quad (3.2)$$

Let

$$r = \triangleright_{\texttt{sample-size}}(R) \quad (3.3)$$

$$S = r \bowtie T \quad (3.4)$$

From the above, we can infer the following:

$$\begin{aligned} & HR(r \bowtie T) = hr_1 && \text{by (3.1) and Definition 3.2.8} \\ \Longrightarrow\ & card(r \bowtie T) = |r| \times hr_1 && \text{by Definition 3.2.4} \end{aligned} \quad (3.5)$$

$$\begin{aligned} S_1 &= \triangleright_{\texttt{LIMIT}}(r \bowtie T) && \text{by (3.1) and Definition 3.2.8} \\ &= \triangleright_{\texttt{LIMIT}}(S) && \text{by (3.4)} \end{aligned} \quad (3.6)$$

$$
\begin{array}{llll}
 & HR(S_1 \bowtie Q) & = hr_2 & \text{by (3.2) and Definition 3.2.8} \\
\Longrightarrow & HR(S \bowtie Q) & \approx hr_2 & \text{by (3.6) and Assumption 3.2.3} \\
\Longrightarrow & |S \bowtie Q| & = |S| \times hr_2 & \text{by Definition 3.2.4} \\
\Longrightarrow & |r \bowtie T \bowtie Q| & = |r \bowtie T| \times hr_2 & \text{by (3.4)} \\
 & & \approx card(r \bowtie T) \times hr_2 & \text{by } card(r \bowtie T) \approx |r \bowtie T| \\
 & & = |r| \times hr_1 \times hr_2 & \text{by (3.5)} \\
\Longrightarrow & HR(r \bowtie T \bowtie Q) & = hr_1 \times hr_2 & \text{by Definition 3.2.4} \\
\Longrightarrow & HR(R \bowtie T \bowtie Q) & \approx hr_1 \times hr_2 & \text{by (3.3) and Assumption 3.2.3}
\end{array}
$$

□

We have proven that the hit ratio of the join sequence $R \bowtie T \bowtie Q$ can be estimated to be the product of the hit ratios of the individual cutoff-sampled joins. The proof is generalizable to a sequence of n joins.

We now give a formal definition of the cutoff-sampling operation and the result size estimation technique of a sequence of joins.

Definition 3.2.11.

$$
\begin{array}{lcl}
\trianglerighteq & : & Join \times \mathbb{N} \rightarrow Table \times \mathbb{R}^+ \\
\trianglerighteq_{\texttt{LIMIT}}(r_1 \bowtie R_2 \ldots \bowtie R_n) & = & (S_n, hr) \\
 & \textbf{where} & \begin{cases} hr = HR(r_1 \bowtie R_2 \ldots \bowtie R_n) \approx \prod_{i=1}^{n-1}(hr_i) \\ (S_2, hr_1) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(r_1 \bowtie R_2) \\ (S_3, hr_2) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_2 \bowtie R_3) \\ \quad\vdots \\ (S_{n-1}, hr_{n-2}) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_{n-2} \bowtie R_{n-1}) \\ (S_n, hr_{n-1}) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_{n-1} \bowtie R_n) \end{cases}
\end{array}
$$

Definition 3.2.12. Given the join sequence $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, and assuming that it is cutoff-sampled by picking a sample from the input relation R_1, the cardinality of its result is estimated using the following formula:

$$
card(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) = |R_1| \times hr \quad \textbf{where} \begin{cases} (S, hr) = \trianglerighteq_{\texttt{LIMIT}}(r_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) \\ r_1 = \triangleright_{\texttt{sample-size}}(R_1) \end{cases}
$$

Cutoff-sampling	Join Hit Ratio
$\trianglerighteq_{100}(\triangleright_{100}(R) \bowtie T) = (S_1, \mathbf{5})$	$hr(\triangleright_{100}(R) \bowtie T) = \mathbf{5}$
$\trianglerighteq_{100}(S_1 \bowtie Q) = (S_2, \mathbf{2})$	$hr(S_1 \bowtie Q) = \mathbf{2}$
	$hr(R \bowtie T \bowtie Q) = \Pi(hr) = \mathbf{2} \times \mathbf{5} = \mathbf{10}$
Estimated Cardinality	$card(R \bowtie T \bowtie Q) = \lvert R\rvert \times \mathbf{10} = 30000$

Figure 3.15 The estimation process of the result size of the join sequence $R \bowtie T \bowtie Q$ by computing the hit ratio of the join sequence. The hit ratio of the join sequence is the product of the hit ratio of each of the individually cutoff-sampled join.

As noted before, it is also possible to estimate the result size of the join sequence $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ by picking a sample from any of the other $(n-1)$ input tables $R_2, R_3, \ldots,$ or R_n.

Example 3.2.13. We suppose that the join sequence $R \bowtie T \bowtie Q$ is cutoff-sampled by executing the two operations $(S_1, 5) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(r \bowtie T)$ and $(S_2, 2) \leftarrow \trianglerighteq_{\texttt{LIMIT}}(S_1 \bowtie Q)$ where $r = \triangleright_{\texttt{sample-size}}(R)$. Let $|R| = 3000$, $\texttt{sample-size} = 100$, and $\texttt{LIMIT} = 100$. Therefore, the two cutoff-sampled joins take as input a sample set of size 100 and limit their output to 100 tuples. The hit ratio of the first cutoff-sampled join is 5, and the hit ratio of the second cutoff-sampled join is 2. Therefore, the hit ratio hr of the join sequence $r \bowtie T \bowtie Q$ is equal to $5 \times 2 = 10$. Assuming r is a representative sample of the table R (Assumption 3.2.3), the hit ratio hr can be extrapolated to the join sequence $R \bowtie T \bowtie Q$. Therefore, the result size of the join $R \bowtie T \bowtie Q$ is equal to $|R| \times 10 = 30000$. Figure 3.15 illustrates the estimation method. The arrows linking elements in the table represent the definition-use chain of some of the values and relations produced during the cutoff-sampling and result size estimation process.

3.2.3 Notes on Sampling in the ROX Prototype

In this section, we describe some important points concerning the sampling techniques adopted in the ROX prototype.

Sampling from indexes: To build a sample of the tuples associated with a given vertex in a join graph, ROX samples from indexes built on base tables and from generated intermediate tables. In fact, in ROX we prefer to

sample indexes instead of base tables since it is more efficient. We therefore assume the existence of indexes on XML elements and values. We stress that the availability of element and value indexes is not a requirement for ROX, tuples may as well be sampled from base tables if the appropriate index does not exist. Efficient and reliable sampling from indexes, using techniques like *partial sum trees* is well-known [100]. An index on relation R is represented by the symbol $\overset{R}{\bigtriangledown}$. Similar to sampling from a table, an element index is sampled with the expression: $\rhd_{\texttt{LIMIT}}(\overset{R}{\bigtriangledown})$. Due to its more complex structure, a value index is not sampled but cutoff-sampled with the expression $\unrhd_{\texttt{LIMIT}}(\overset{R}{\bigtriangledown})$. With some extra engineering effort, the normal sampling technique could be used. While sampling from an index, we can also estimate the number of nodes associated with a given vertex in a join graph. We denote the cardinality estimation process from an index with the symbol $EstCard(\overset{R}{\bigtriangledown})$.

Sampling joins: To ensure efficiency when sampling a join, it is better to use physical join operators that satisfy the *zero-investment* property with respect to their sampled input. A zero-investment operator is defined to be an operator, the complexity of which only depends on the cardinality of its sampled input. That is the operation $OP(r, T)$ between the sampled input r and the relation T complies to the zero-investment property if its cost is linear in the size of the sample r. This rules out any algorithm that, prior to producing results, makes an investment that is linear (or worse) with respect to any input relation other than the sampled one. Equi-join algorithms that satisfy the zero-investment property are nested-loop index lookup, merge join (only applicable if the inner input is ordered), and hash-join (only applicable if a hash table is already built on the inner relation). The zero-investment condition is a generalization of the "index-available" condition [114] that is long known to simplify the issue of efficiently obtaining reliable join samples.

Sampling of relational joins in a database system has been the subject of research, and several sampling approaches have been suggested already [29, 64, 100]. One sampling methodology, proposed in *index based join selectivity estimation* [114], takes a random sample of input tuples from the outer operand, and looks-up efficiently, using an index, *all* matching tuples in the inner operand. This sampling technique complies to the introduced zero-investment property, and typically applies in our join graph when there is an equality edge touching two attribute- or two text-nodes. In that case the XML value indexes are used.

We note that the XPath step joins used in the ROX prototype also conform to the zero-investment property with regards to their sampled

input. In fact, we observed that our join graphs can be fully sampled in ROX with zero-investment operators. The implementation of the sampling operation of XPath steps and relational joins will be explained in greater detail in Section 5.2 when describing the ROX prototype implementation.

Representativeness of samples (Assumption 3.2.3): The cardinality estimation techniques we have described in this section are all built upon Assumption 3.2.3. It is known from literature that the representativeness of a sample affects the accuracy of the estimated result size. We will see in the conducted experiments in Chapter 5 and Chapter 6 to which extent this assumption holds, and the consequences on the quality of the decisions made by the optimizer when it does not.

3.3 Notation

This section presents some notations that will be used in the subsequent chapters of this thesis. The definitions are grouped into four sections; the first corresponds to a summary of the possible edges in any given join graph, the second enumerates some notations corresponding to vertices in the join graph, the third presents notations for edges, and the fourth lists some notations for operators.

We first start by defining the notations to be used in the following sections:

$$\begin{array}{rcl} \mathit{Edge} & = & \mathit{Vertex} \times \mathit{Vertex} \\ \mathit{Path} & = & \mathit{Edge} \mid (\mathit{Edge} \times \mathit{Path}) \\ \mathit{Table} & = & \text{list of XML nodes and/or values} \\ \mathit{Join} & = & \text{a relational join or XPath step} \end{array}$$

We note that some of the functions defined in the following sections are total and some are partial.

3.3.1 Notations for Join Graph

Figure 3.16 shows a table summarizing the different possible edges, and their semantics, in any given join graph.

3.3.2 Notations for Vertices

In this section, we introduce some notations and terms corresponding to vertices in a join graph.
Given a join graph $G = (V, E)$ and a vertex $v \in V$, we define the following:

$$\mathit{TBL} \quad : \quad \mathit{Vertex} \rightarrow \mathit{Table}$$

Notation	Semantic
$v_1 \overset{ax}{\circ\!\!-\!\!-} v_2$	the XPath step $v_1/ax::v_2$
$v_1 \overset{pred}{-\!\!-} v_2$	the relational join $v_1 \bowtie_{pred} v_2$
$v_1 -\!\!- v_2$	an arbitrary join between between v_1 and v_2, abstracting from a particular XPath step or relational join.
$v_1 \overset{ax}{\bullet\!\!\rightsquigarrow} v_2$	the executed XPath step $v_1/ax::v_2$
$v_1 \overset{pred}{\rightsquigarrow} v_2$	the executed relational join $v_1 \bowtie_{pred} v_2$
$v_1 \rightsquigarrow v_2$	an arbitrary executed join between v_1 and v_2, abstracting from a particular XPath step or relational join.

Figure 3.16 The different possible edges in a join graph.

$TBL(v)$ is the full table associated with v. It consists of a relation containing all the XML nodes corresponding to the vertex v

$SMPL : Vertex \rightarrow Table$
$SMPL(v)$ is the sample table associated with v. It consists of a relation containing a random sample of the XML nodes corresponding to the vertex v

$card : Vertex \rightarrow \mathbb{N}$
$card(v)$ is the cardinality of the vertex v. It represents an estimation of the number of XML nodes corresponding to v

$edges : Vertex \rightarrow \mathcal{P}(Edge)$
$edges(v)$ is a set containing all the outgoing edges of v

$edges^+ : Vertex \rightarrow \mathcal{P}(Edge)$
$edges^+(v)$ is a set containing all the outgoing *executed* edges of v

$edges^{-}$: $Vertex \rightarrow \mathcal{P}(Edge)$
$edges^{-}(v)$ is a set containing all the outgoing *unexecuted* edges of v

$graph^{+}$: $Vertex \rightarrow Graph$
$graph^{+}(v)$ is a rooted graph consisting of the join graph G rooted at vertex v and containing only the chains of *executed* edges

$graph^{-}$: $Vertex \rightarrow Graph$
$graph^{-}(v)$ is a rooted graph consisting of the join graph G rooted at vertex v and containing only the chains of *unexecuted* edges

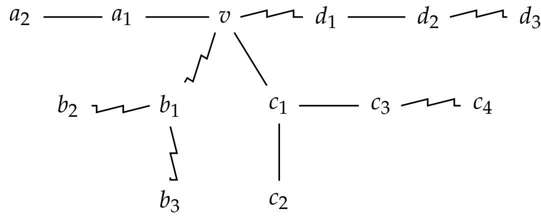
$edges^{*}$: $Vertex \rightarrow \mathcal{P}(Edge)$
$edges^{*}(v)$ is a set containing all the edges in the rooted graph $graph^{+}(v)$

$paths^{-}$: $Vertex \rightarrow \mathcal{P}(Path)$
$paths^{-}(v)$ is a set containing all the possible paths in the rooted graph $graph^{-}(v)$ starting from the root v and ending at a leaf in the graph. In the case of a cyclic graph, each path cannot contain the same edge more than once.

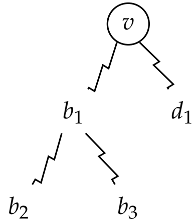
Definition 3.3.1. Given a join graph $G = (V, E)$, a vertex $v \in V$ is said to be an *executed vertex* if and only if $|edges^{+}(v)| > 0$.

We now illustrate some of the above definitions using the join graph depicted in Figure 3.17:

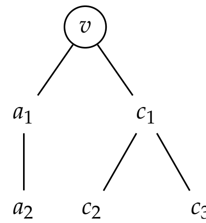
$$
\begin{aligned}
edges(v) &= \{(v,a_1),(v,b_1),(v,c_1),(v,d_1)\} \\
edges^{+}(v) &= \{(v,b_1),(v,d_1)\} \\
edges^{-}(v) &= \{(v,a_1),(v,c_1)\}
\end{aligned}
$$



a The join graph used to illustrate some of the defined notations corresponding to vertices.



b The rooted graph $graph^+(v)$ with the encircled root v. The graph consists of all the chains of executed edges contained in the join graph in Figure 3.17a.



c The rooted graph $graph^-(v)$ with the encircled root v. The graph consists of all the chains of unexecuted edges contained in the join graph in Figure 3.17a.

Figure 3.17 The join graph used to illustrate some of the defined notations corresponding to vertices. The two rooted graphs $graph^+(v)$ $graph^-(v)$ are also depicted.

$$
\begin{aligned}
graph^+(v) &= \text{Shown in Figure 3.17b} \\
graph^-(v) &= \text{Shown in Figure 3.17c} \\
edges^*(v) &= \{(v, b_1), (b_1, b_2), (b_1, b_3), (v, d_1)\} \\
paths^-(v) &= \Big\{\{(v, a_1), (a_1, a_2)\},\ \{(v, c_1), (c_1, c_2)\}, \\
&\quad \{(v, c_1), (c_1, c_3)\}\Big\}
\end{aligned}
$$

3.3.3 Notations for Edges

In the following, we introduce some notations corresponding to edges in a join graph.
Given a join graph $G = (V, E)$ and an edge $e = (v_1, v_2) \in E$, we define the following:

$$op \quad : \quad Edge \times Table \times Table \rightarrow Join$$

$op(e, T_1, T_2)$ is the join operator associated to the edge e taking as input the tables T_1 and T_2

$$w : Edge \rightarrow \mathbb{N}$$

$w(e)$ is the weight of the edge e which represents the estimated cardinality of the join operator $op(e, TBL(v_1), TBL(v_2))$

$$exec : Edge \rightarrow Table$$

$exec(e)$ represents the execution of the join operator $op(e, TBL(v_1), TBL(v_2))$. It returns a table containing the output result of the execution.

$$exec : Edge \times Table \times Table \rightarrow Table$$

$exec(e, T_1, T_2)$ represents the execution of the join operator $op(e, T_1, T_2)$. It returns a table containing the output result of the execution.

3.3.4 Notations for Operators

We define the following operators:

$$\triangleright : (Table \cup Index) \times \mathbb{N} \rightarrow Table$$

$\triangleright_{\texttt{LIMIT}}(T)$ Samples a set of tuples of size LIMIT from the table or index T

$$\trianglerighteq : Operator \times \mathbb{N} \rightarrow Table \times \mathbb{R}^+$$

$\trianglerighteq_{\texttt{LIMIT}}(r_1 \bowtie R_2 \ldots)$ Cutoff-sample the input join(s)

$\trianglerighteq_{\texttt{LIMIT}}(\overset{R}{\bigtriangledown})$ Cutoff-sample the input index

$$\overset{D_{elt}}{\bigtriangledown} : String \rightarrow Table$$

$\overset{D_{elt}}{\bigtriangledown}(q)$ returns the list of all element nodes in document D with qname q

$\overset{D_{text}}{\bigtriangledown}$	:	$\mathbb{R} \rightarrow \mathit{Table}$
$\overset{D_{text}}{\bigtriangledown}(v)$		returns the list of all candidate text nodes in document D with value v
EstCard	:	$\mathit{Index} \rightarrow \mathbb{N}^{+}$
$\mathit{EstCard}(\overset{D_x}{\bigtriangledown}(y))$		returns an estimation of the size of the list generated by the specified index lookup $\overset{D_x}{\bigtriangledown}(y)$

3.4 Conclusion

In this chapter, we have introduced the foundations on which ROX is built. We first described the join graph structure which is the input to the ROX algorithm. The join graph is an order-free representation of the XPath steps and relational joins in an XQuery, providing ROX with the opportunity to seamlessly order the two different types of operators. ROX can also optimize several join graphs embedded in a single plan allowing it to handle the full XQuery language.

We have also described the sampling technique used by ROX to sample joins and the method employed to estimate the result size of the joins. We have introduced the cutoff-sampling operation which guarantees that the cost of sampling is kept under control.

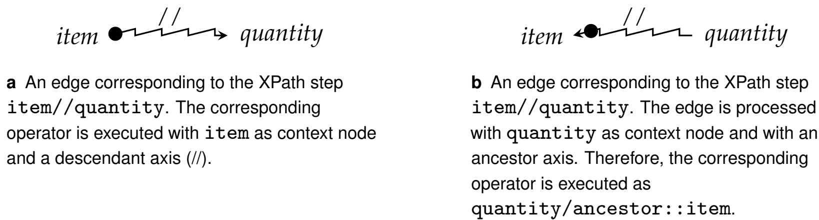
Last but not least, some notations which will be used in the subsequent chapters were presented.

ROX: Run-time Optimization of XQueries 4

Traditional relational database systems are currently the most used corporate database systems in companies and organizations. As we have stressed in Chapter 1 of this thesis, the optimizers of these systems fail, is some situations, to deliver the expected optimization quality due to several existing challenges. The reasons behind these challenges originate mostly from the design of the optimizer. In the context of XQuery, these challenges and their impact are aggravated. As a solution, we propose a new XQuery optimizer which adopts a design different than the one existing in traditional optimizers, allowing it to overcome the current challenges.

We opt for an optimizer that postpones the optimization of crucial sections of the query plan to run-time. These sections consist of join operators (XPath steps and relational joins), the most expensive but heavily used operators in database systems. Contrary to traditional relational optimizers, the new run-time XQuery optimizer is autonomous: it does not depend on any a priori collected statistics and cost model, and uses sampling techniques to learn about the characteristics of the queried data and the correlations among them. The choices we have made stem from our belief that for an optimizer to be good and robust, it has to be adaptive, basing its decisions on up-to-date observations about the content of the database, the characteristics of the queried data, as well as the availability of computing resources.

This chapter describes **ROX**, our **R**un-time **O**ptimizer for **X**Queries. In this chapter, ROX is explained in the context of database systems optimized to fully materialize their intermediate results. We first give a general description of ROX and follow it up with a detailed presentation of the algorithm. We then explain in detail the chain sampling process, and stress the differences between the theoretical and implemented chain sampling variants. We end this chapter with an example that illustrates the ability of ROX to detect and exploit the existing correlation in the queried data.



a An edge corresponding to the XPath step `item//quantity`. The corresponding operator is executed with `item` as context node and a descendant axis (//).

b An edge corresponding to the XPath step `item//quantity`. The edge is processed with `quantity` as context node and with an ancestor axis. Therefore, the corresponding operator is executed as `quantity/ancestor::item`.

Figure 4.1 An edge corresponding to the XPath step `item//quantity`. The edge can be executed with either a forward or a backward axis.

4.1 Introducing ROX

Before we give a general description of ROX, we briefly restate the optimization problem that ROX aims to solve.

4.1.1 Problem Statement

ROX focuses on optimizing at run-time the execution order of XPath steps and relational joins in an XQuery query. As we have seen in Section 3.1, the to-be-ordered operators are passed to ROX in a join graph structure as part of the complete execution plan. Given a join graph G, the ROX algorithm will assign an ordering to the edges in the graph, defining the *execution order* of the corresponding join operators.

ROX not only designates an order to the edges, but also an *execution direction*. The execution direction of an edge, indicated with an arrow, specifies the left and right operands of the execution of the join operator corresponding to the edge. For instance, the edge ($a \rightsquigarrow b$) is executed using the tables associated with the vertices a and b as respectively the left and right input operands ($TBL(a) \bowtie TBL(b)$). If the edge corresponds to an XPath step, the execution direction not only marks the vertex used as context node in the execution, but also defines the type of XPath axis to be used. An example of an XPath step execution is illustrated in Figure 4.1. The step can be executed in two ways: either with a forward axis (`item//quantity`) or with a backward axis (`quantity/ancestor::item`). We note that in XML query processing, depending on the document structure, there may be significant differences in executing an XPath step with e.g. a `child` or `parent` axis.

Therefore, given a join graph, ROX needs to determine robustly and efficiently the execution order and direction of the edges in the graph, such that the generated plan is (near-)optimal.

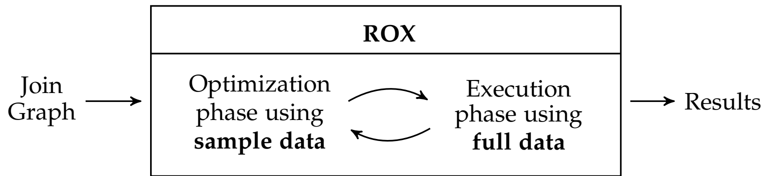


Figure 4.2 An illustration of the steps of ROX: optimization and execution steps are alternated until all operators in the join graph are executed. Every optimization consists of sampling path segments in the join graph until a "superior" path is found. The execution phase executes the join operators in the chosen path and materializes the results. Therefore, ROX defines the execution plan step-by-step, with each optimization phase shaping one section of the plan, and each execution phase directly evaluating the operators in the newly defined section. When all operators in the graph are executed, ROX returns a relation containing all the tuples satisfying the join operations in the graph.

4.1.2 General Description of ROX

In ROX, 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. To accomplish this, ROX adopts a strategy of interleaving *optimization* and *execution* steps. We stress again that this chapter describes ROX in the context of database systems optimized to fully materialize their intermediate results.

Every optimization phase explores the search space looking for a "superior" path segment (sequence of join operators). The search is performed by the *Chain Sampling* process which samples efficiently and iteratively different path segments in the join graph. As soon as a path segment is found to be superior to others, the sampling stops. The execution phase then executes the associated XPath steps and relational join operators in the chosen path segment, and materializes the results. Then a new optimization phase initiates the search for the next superior path segment, benefiting from the newly materialized intermediates which are analyzed to obtain up-to-date, more accurate information about the joins in the graph. ROX stops the intertwining of optimization and execution steps when all operators in the join graph are executed. Figure 4.2 presents an illustration of the steps of ROX. Every iteration consists of an optimization phase which is based on join sampling operations, and an execution phase which executes these joins using full tables. Therefore, the execution plan is defined in an iterative manner as the ROX algorithm proceeds. Every path segment chosen during an optimization phase determines one section of the plan which is then executed directly during the subsequent execution step.

We note that it is the alternation of optimization and execution steps that gives ROX part of its robustness. Since every execution phase is followed by a full materialization of the generated intermediate result, it opens the opportunity for the next optimization step to use the new data to update its knowledge about the selectivities of the joins in the graph. This, as a result, allows the detection of existing data correlations among the joined vertices in the graph. Next, we enumerate some of the characteristics of ROX, then we give a more detailed description of the optimization phase in ROX.

Characteristics of ROX

We stress the following important aspects of ROX:

1. ROX is an autonomous, robust and efficient optimizer. It is the first run-time optimizer for XQueries and one of the very few techniques in the relational context that completely interleaves query optimization with query execution.
2. ROX improves the state-of-the art in XQuery optimizers both in plan quality as well as running time.
3. ROX does not depend on any a priori collected statistics and a pre-built cost model, and therefore it does not suffer from the deficiencies of the current state-of-the-art in cost estimation.
4. The chain sampling technique implemented in ROX provides the first generic and robust method to detect any type of correlated data.
5. ROX is the first optimizer which can seamlessly optimize the execution order and direction of both relational joins and XPath steps.
6. The ROX approach is generic enough to be exported to other query languages, like SQL and SPARQL.[1]
7. ROX is not "query-specific": unlike traditional optimizers, the quality of the optimization decisions in ROX does not degrade when handling unexpected query loads.

Optimization in ROX

In ROX, the execution plan is generated step-by-step: every optimization phase shapes one section of the plan by determining the best sequence of operators to be executed. To produce cheap execution plans, a strategy followed by most database systems is to reduce as much as possible the number of tuples generated and flowing through the operators in the plan. This basically translates into first executing the operators that keep the output small. Using sampling techniques, ROX can estimate the weight

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

(*i.e.* result size) of each edge in the join graph and then pick out the one with the smallest output cardinality for evaluation. However, we realize that executing the operator corresponding to the spotted edge directly is a greedy decision, and can lead to an execution plan that is far from optimal.

Determining the edge with the smallest weight is in fact equivalent to finding a *local minimum* in the search space of the join graph. However, there might exist a sequence of edges in the graph which forms a *global minimum*, and thus generates a smaller number of intermediary tuples. To detect the existence of such paths (sequences of joins), the optimization phases of ROX adopt a *chain sampling* technique which uses the local minimum as starting point and then invests a small amount of time to climb the hill searching for a global minimum.

Before we explain the chain sampling technique in more detail, we give the following two definitions:

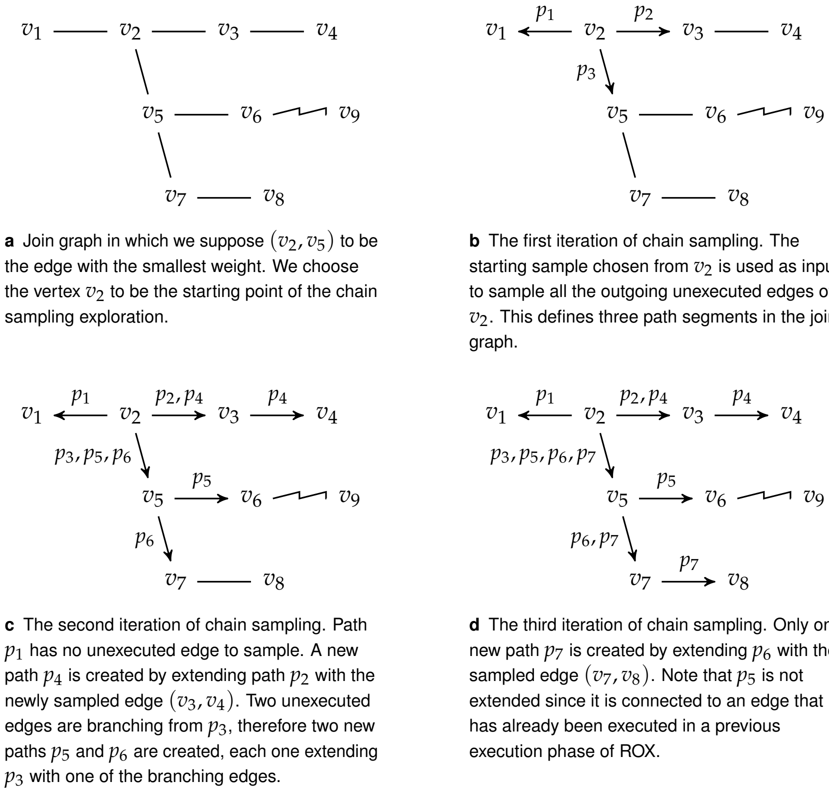
Definition 4.1.1. Superiority of a path: Given a set S of paths, a path p is said to be *superior* to all other paths in S if the execution of p followed by the execution of any path $p' \in S$ is cheaper than executing first p' and then p.

Definition 4.1.2. Absolute superiority of a path: Given a set S of paths, a path p is said to be *absolutely superior* to all other paths in S if the execution of p followed by the execution of any path $p' \in S$ is cheaper than executing p' alone.

We note that absolute superiority implies superiority, that is a path p that is absolutely superior to all other paths in S is also superior to all paths in S.

We also stress that the set S does not include all possible paths in the search space of the join graph. Moreover, the comparisons between pairs of paths can be implemented efficiently such that little time is spent on the checking for the superiority and absolute superiority of a path.

Given the edge e with the smallest weight, chain sampling is a process exploring the path segments (sequences of joins) around e, in search for a path segment that is ***superior*** *to all the explored paths in the join graph including the edge e.* When such a path is found, the path is returned for execution. The starting point of the exploration process is the edge e: given a sample chosen from one of the vertices v of e, chain sampling samples iteratively in a breadth-first manner the sequences of unexecuted edges branching from v, until detecting the path p. During every sampling iteration, new path segments are defined by extending previously explored paths with an additional newly sampled edge. The selectivity of an explored path segment is estimated by consecutively sampling the edges along the path, using the output of the sampling operation of one operator as input to the sampling



a Join graph in which we suppose (v_2, v_5) to be the edge with the smallest weight. We choose the vertex v_2 to be the starting point of the chain sampling exploration.

b The first iteration of chain sampling. The starting sample chosen from v_2 is used as input to sample all the outgoing unexecuted edges of v_2. This defines three path segments in the join graph.

c The second iteration of chain sampling. Path p_1 has no unexecuted edge to sample. A new path p_4 is created by extending path p_2 with the newly sampled edge (v_3, v_4). Two unexecuted edges are branching from p_3, therefore two new paths p_5 and p_6 are created, each one extending p_3 with one of the branching edges.

d The third iteration of chain sampling. Only one new path p_7 is created by extending p_6 with the sampled edge (v_7, v_8). Note that p_5 is not extended since it is connected to an edge that has already been executed in a previous execution phase of ROX.

Figure 4.3 Illustration of chain sampling. The starting point of exploration is the edge with the smallest weight which we suppose to be (v_2, v_5). A sample table chosen from the vertex v_2 is used to sample the surrounding path segments of unexecuted edges. The sampling is performed iteratively in a breadth-first manner. The labels on the edges denote the path id(s) to which the edges belong, and the arrows indicate the sampling direction (*i.e.* the left and right operands of the sampling operation).

operation of the subsequent operator. By sampling ahead in the branches, ROX may discover that a path, due to correlations, produces a result of much lower cardinality than the initially predicted estimations, and hence proving to be superior to others. We illustrate the chain sampling process with the following example.

Example 4.1.3. Consider the join graph in Figure 4.3a, we suppose that the edge in the join graph with the smallest weight is (v_2, v_5), and we choose the vertex v_2 to be the starting point of the chain sampling exploration. Therefore, chain sampling will explore iteratively, in a breadth-first-manner

Iteration 1	Sampled Edges	(v_2, v_1) - (v_2, v_3) - (v_2, v_5)
	Defined Paths	$p_1 = \{(v_2, v_1)\}$ $p_2 = \{(v_2, v_3)\}$ $p_3 = \{(v_2, v_5)\}$
Iteration 2	Sampled Edges	(v_3, v_4) - (v_5, v_6) - (v_5, v_7)
	Defined Paths	$p_4 = \{(v_2, v_3), (v_3, v_4)\}$ $p_5 = \{(v_2, v_5), (v_5, v_6)\}$ $p_6 = \{(v_2, v_5), (v_5, v_7)\}$
Iteration 3	Sampled Edges	(v_7, v_8)
	Defined Paths	$p_7 = \{(v_2, v_5), (v_5, v_7), (v_7, v_8)\}$

Figure 4.4 The sampled edges and definition of paths at every iteration of the chain sampling process illustrated in Figure 4.3. At every iteration, new paths are created by extending previously explored paths with newly sampled edges.

the path segments branching from vertex v_2. The edges in Figure 4.3b, Figure 4.3c, and Figure 4.3d are labeled with the *path id* to which they belong, and the arrows denote the direction of sampling (*i.e.* the left and right operands of the sampling operation). Figure 4.4 enumerates the edges sampled at each iteration, and illustrates the creation of paths.

Iteration 1 **(Figure 4.3b):** During the first iteration, all the unexecuted edges branching from v_2 are sampled using as input a sample set chosen from v_2. Since the number of outgoing unexecuted edges is three, three new paths p_1, p_2 and p_3 are created. Paths p_1, p_2, and p_3 contain respectively the sampled edges (v_2, v_1), (v_2, v_3), and (v_2, v_5).

Iteration 2 **(Figure 4.3c):** The second iteration samples the next unexecuted edges branching from each of the three defined paths. From each unexecuted edge e branching from the path p, a new path p' is created. Path p' contains all the edges in p in addition to the newly sampled edge e. For the previously explored paths p_1, p_2, p_3, we have the following:

1. Path p_1: No unexecuted edge is branching from the end vertex v_1 of p_1, and therefore no new path is created.
2. Path p_2: Path p_2 has one unexecuted edge (v_3, v_4) branching from its end vertex v_3. A new path p_4 is created: $p_4 = \{(v_2, v_3), (v_3, v_4)\}$.
3. Path p_3: Two unexecuted edges (v_5, v_6) and (v_5, v_7) branch from the end vertex v_5 of path p_3. Therefore, two paths are created: $p_5 = \{(v_2, v_5), (v_5, v_6)\}$ and $p_6 = \{(v_2, v_5), (v_5, v_7)\}$.

Iteration 3 **(Figure 4.3d):** In the third iteration, the only remaining edge to sample is (v_7, v_8). The edge is branching from path p_6, and therefore a new path p_7 containing the edges in p_6 and the newly sampled edge (v_7, v_8) is created. Note that the edge (v_6, v_9) branching from path p_5 is not sampled since it has already been executed in a previous execution phase of ROX.

Chain sampling does not necessarily sample all the unexecuted edges in the join graph before deciding about the superior path to return for execution. In fact, at every iteration, the chain sampling process notes down the characteristics of the newly created paths (e.g. cost and selectivity of the operators in the paths), and compares the observed properties of all paths sampled so far to decide if one path is so selective that the exploration can be safely halted. The detected selective path is then returned for execution. To assist chain sampling in comparing the paths and deciding whether to proceed with or stop the exploration process, a *stopping condition* has been formulated. *The stopping condition identifies the existence of a sampled path segment p that is* ***absolutely superior*** *to every other explored path in the join graph. The stopping condition guarantees that even if any of the explored paths other than p is extended with highly selective edges, p remains absolutely superior to the newly extended paths, making it safe to stop the chain sampling exploration*. If the stopping condition succeeds in finding a path p that is absolutely superior to all the other explored paths, then the chain sampling process is halted and p is returned for execution. As we have noted earlier, p is also superior to all the explored paths. The chain sampling process and the proposed stopping condition are explained in more detail in Section 4.3.

In this section, we have given a general description of ROX in which optimization and execution steps are alternated. We have also briefly explained the optimization phase of ROX introducing the adopted chain sampling technique. In the next section, we give an elaborate description of the ROX algorithm.

4.2 The ROX Algorithm

Before we start the presentation of the ROX algorithm, we make the following note.

Note 4.2.1. Given the edge e with its two vertices v_1 and v_2, the sampling with cutoff limit τ and the execution of e is represented in the ROX algorithm with the following two respective operations:

- $\trianglerighteq_\tau(op(e, SMPL(v_1), TBL(v_2)))$
- $exec(e, TBL(v_1), TBL(v_2))$

Although the above operations are represented as a join between the sample or full table associated with the vertex v_1 and the full table associated with v_2, we stress that they correspond to only the logical representations of the sampling and execution operations, and should not be interpreted as the physical implementations used to actually carry out these two operations. In fact, the sampling and execution operations might be processed using different strategies, *e.g.* using an index-based join. Therefore, we note that the above operations do not require the pre-materialization of the full table $TBL(v_2)$ to carry out the sampling or execution of the join.

To make the description of the ROX algorithm easy to follow, we adopt the above logical representation for the sampling and execution operation of edges, and, for the time being, abstract away the details of the implementation of these two operations. We will describe in detail the actual physical implementation of the operations in Section 5.2 when presenting the ROX prototype.

The ROX algorithm is given in Algorithm 1. It takes as input the join graph G containing the to-be-ordered join operators, the limit for cutoff-sampling τ, and the size τ' of the sample tables associated with the vertices in the graph. The algorithm consists of two phases. The first phase initializes the join graph. The second phase is the core of the algorithm, in which search space exploration using chain sampling and path segment execution are alternated until all edges in the graph are executed. We now proceed with describing in detail the two phases.

4.2.1 Phase 1 (Algorithm 1: lines 1-9)

This first phase initializes the join graph by acquiring knowledge about its vertices and edges. As we will see, for some of the vertices and edges in the graph, no information can be learned.

- **Learning about vertices (Algorithm 1: lines 1-7):** Indexes are exploited to acquire knowledge about the vertices in the graph. For a vertex v, the corresponding index is sampled to build a subset $SMPL(v)$ of size τ' of the XML nodes corresponding to v **(line 3, line 6)**. The index is also used to estimate the cardinality $card(v)$ of the vertex v, *i.e.* to estimate the total number of XML nodes corresponding to v, **(line 4, line 7)**. In principle, the sampling and cardinality estimation operations are performed for all kinds of vertices as long as a technique that can execute the two operations while respecting the zero-investment property is available. In the ROX prototype, the sampling and cardinality estimation operations are restricted to vertices representing either an XML element type with

Algorithm 1: ROX: RUN-TIME OPTIMIZER FOR XQUERIES

```
   INPUT : Join Graph G = (V, E), Int τ, Int τ'
   // τ = limit for cutoff-sampling, τ' = size of the sample
      tables associated with the vertices in the graph

   // Initialization phase
1  FOREACH v ∈ V DO
2  |  IF v is an element type with qualified name q THEN
3  |  |  SMPL(v) ← ▷_τ'( ▽^{D_elt}(q)) ;
4  |  |  card(v) ← EstCard( ▽^{D_elt}(q)) ;
5  |  ELSE IF v is a text node with predicate "= x" THEN
6  |  |  SMPL(v) ← ⊵_τ'( ▽^{D_text}(x)) ;
7  |  |  card(v) ← EstCard( ▽^{D_text}(x)) ;

8  FOREACH e = (v1, v2) ∈ E | SMPL(v1) ≠ NULL ∨ SMPL(v2) ≠ NULL DO
9  |  w(e) ← WEIGHT(e);

   // Core phase: alternation of optimization and execution
10 WHILE ∃ more edges to execute DO
11 |  Path p ;      // p = sequence of operators to be executed
12 |  Edge e = (v1, v2) | e ∈ E ∧ w(e) = min_{e_i ∈ E} w(e_i);
13 |  IF |edges⁻(v1)| > 1 ∨ |edges⁻(v2)| > 1 THEN
14 |  |  p ← CHAINSAMPLE(e);
15 |  ELSE
16 |  |  p ← {e};
17 |  EXECPATH&UPDATEJG(p);
```

qualified name q or an XML text node with an equality predicate condition $= x$ **(line 2, line 5)**. We note that the type of indexes supported in the ROX implementation cannot be used to lookup nor to estimate the total number of tuples corresponding to vertices representing either an XML text node without a predicate condition or with a predicate condition different than equality, or an XML attribute node with or without a predicate condition. A detailed description of the indexes implemented in the ROX prototype is given in Section 5.1.3.

- **Learning about edges (Algorithm 1: lines 8-9):** For an edge e in the join graph, the weight of e, *i.e.* the result size of the operator associated with e, is computed. The weight $w(e)$ is estimated with the *Weight* function which uses the cutoff-sampling technique described in Section 3.2. Since the computation of the weight of an edge consists of sampling the edge using as input a sample chosen from one of its vertices, an edge $e = (v_1, v_2)$ with two vertices that do not have a materialized sample ($SMPL(v_1) = SMPL(v_2) = \texttt{NULL}$) will stay initially unweighted.

The WEIGHT Function

We now describe the *Weight* function presented in Algorithm 2. The function takes as input an edge $e = (v, v')$ and returns its weight which is estimated by sampling the operator associated with e. First the left and right operands of the sampling operation are determined, in other words the table from which to pick the input sample is specified **(lines 2-10)**. There are two criteria that define our choice: the availability of materialized samples associated with the edge's vertices, and the estimated cardinality of each vertex. If the edge's two vertices v and v' have a materialized sample table, then the input sample is selected from the vertex with the smaller estimated cardinality **(lines 2-6)**. The reasoning behind our choice is that picking a sample from a smaller table results in a more representative sample set which in turn leads to a more accurate cardinality estimation. If only one of the vertices has a materialized sample, then the only possible choice is to choose the input sample set from that vertex **(lines 7-10)**.

Once the left and right operands *lopd* and *ropd* of the sampling operation are determined, the operator associated to the edge e is cutoff-sampled with the limit τ using as input the sample table $SMPL(lopd)$ and the full table $TBL(ropd)$ **(line 11)**. The weight of the join is then estimated using the hit ratio value returned by the sampling operation **(line 12)**.

Note about the sampling of edges: The above sampling operation is represented as a join between the sample table $SMPL(lopd)$ and the full table $TBL(ropd)$. As mentioned in Note 4.2.1, we stress that the sampling operation can be executed differently and therefore the table $TBL(ropd)$ does not need to be pre-materialized. We elaborate on the implementation details of the sampling operation in Section 5.2

Now that we have explained the initialization phase of the ROX algorithm in which knowledge about the cardinality of the vertices and the weight of the edges in the join graph is acquired, we proceed with describing the second and core phase of ROX.

Algorithm 2: WEIGHT

```
   INPUT  : Edge e = (v, v')
   OUTPUT: Int weight
1  Vertex List (lopd, ropd) = (NULL, NULL);
   // Determine sampling direction of e
2  IF SMPL(v) ≠ NULL ∧ SMPL(v') ≠ NULL THEN
3  |  IF card(v) ≤ card(v') THEN
4  |  |  (lopd, ropd) = (v, v');
5  |  ELSE
6  |  |  (lopd, ropd) = (v', v);
7  ELSE  IF SMPL(v) ≠ NULL THEN
8  |  (lopd, ropd) ← (v, v');
9  ELSE
10 |  (lopd, ropd) ← (v', v);
   // Cutoff-sampling of e
11 (S, hr) ← ⊵τ(op(e, SMPL(lopd), TBL(ropd)));
   // Compute weight of e
12 Int weight ← card(lopd) × hr;
13 RETURN weight;
```

4.2.2 Phase 2 (Algorithm 1: lines 10-17)

The second phase of the algorithm represents the core of ROX. It consists of the alternation of optimization and execution steps until all edges in the join graph are executed. Optimization starts by picking the unexecuted edge e in the graph with the smallest weight **(line 12)**. This edge represents the operator that is estimated to return the smallest number of tuples. However, as we have stressed earlier, the edge e might be a local minimum, and its execution would be a greedy decision. Therefore, chain sampling is used to climb the hill and find a global minimum **(line 14)**. Chain sampling initiates an exploration process in search for a path segment p (sequence of joins) that is ***superior*** to all other explored paths in the join graph including the chosen edge e. The selected path segment p is then returned for execution. The search for p starts from the edge e and explores the branches of unexecuted edges around e. Therefore, if each of the vertices of e have at most one unexecuted outgoing edge, chain sampling is not applied. In that case, the path segment p to be executed consists of the singleton edge e **(line 16)**. The chain sampling process is

given in Algorithm 5, and will be explained in detail in Section 4.3. Once p is defined, the execution step of ROX starts **(line 17)**. The execution phase executes the operators associated with the edges in the path segment p and, using the newly materialized intermediate results, updates the knowledge about the vertices and edges in the join graph. The execution phase is given in Algorithm 3 and will be described next. Optimization phases based on chain sampling and execution phases are alternated in ROX until all operators in the join graph are executed.

The EXECPATH&UPDATEJG Function

Given a path segment p, the EXECPATH&UPDATEJG function presented in Algorithm 3 executes every edge $e = (v_1, v_2) \in p$ **(lines 1-11)**. Before the execution, the XML nodes corresponding to each of the edge's vertices v_1 and v_2 are first retrieved, in case the full tables $TBL(v_1)$ and $TBL(v_2)$ have not been materialized yet **(lines 2-9)**. Similarly to the first phase of ROX, available indexes are used to look-up the corresponding nodes. Therefore, the full table of only those vertices that represent either an element type with a certain qualified name or a text node with an equality predicate condition are materialized. For the other types of vertices and given the available indexes, it is not possible to efficiently retrieve their corresponding XML nodes. The full table associated with the vertex representing the document's root is a singleton relation containing the value 1 of the root node's id. In this situation we are supposing a single document is queried; however, this is easily generalizable to a collection of documents. In the latter case, the full table will contain the id value of the root of each individual document in the collection. Next the execution direction of the edge (*i.e.* the left and right operands $(lopd, ropd)$ of the join) is determined using the function EXECUTIONDIRECTION **(line 10)** which is presented in Algorithm 4, and explained in the subsequent section. The edge is then executed as a join between the tables $TBL(lopd)$ and $TBL(ropd)$ **(line 11)**.

Note about the execution of edges: We stress again that it might not be possible to efficiently materialize the full table $TBL(ropd)$, right input of the execution operation presented above. If that is the case, the join will be executed differently as we have observed in Note 4.2.1. More details about the implementation of the execution of edges will be presented in Section 5.2.

After each execution of an edge $e = (v_1, v_2)$ in the path p, the knowledge in the join graph is updated **(lines 12-16)**. In fact, we have already seen in Section 3.1.2 that, as a result of the execution of an edge in the join graph, the content of the full tables associated with the edge's vertices is updated

Algorithm 3: ExecPath&UpdateJG

```
INPUT  : Path p
   // Materialize full tables of vertices of e
1  FOREACH edge e = (v1, v2) ∈ p DO
2      FOREACH v ∈ {v1, v2} DO
3          IF TBL(v) = NULL THEN
4              IF v is a root node THEN
5                  TBL(v) ← [id | 1];
6              ELSE IF v is an element type with qualified name q THEN
7                  TBL(v) ← ▽^{D_elt}(q);
8              ELSE IF v is a text node with predicate "= x" THEN
9                  TBL(v) ← ▽^{D_text}(x);

       // Determine execution direction of e
10     Vertex List (lopd, ropd) = ExecutionDirection(e);
       // Execute e
11     exec(e, TBL(lopd), TBL(ropd));
       // Update knowledge in join graph
12     FOREACH v ∈ {v1, v2} DO
13         SMPL(v) ← ▷_τ'(TBL(v));
14         card(v) ← |TBL(v)|;
15         FOREACH e ∈ edges⁻(v) DO
16             w(e) ← Weight(e);
```

with the result of the execution (refer back to Example 3.1.6). These up-to-date, newly materialized intermediates are used to gain better and more accurate knowledge about the vertices and the joins in the graph. For each executed edge $e = (v_1, v_2)$, the sample table of each of the vertices v_1 and v_2 is updated with a new sample randomly chosen from the full tables of respectively v_1 and v_2 **(line 13)**. The cardinality of the vertices is updated as well with the size of the corresponding full tables **(line 14)**. Finally the weight of the outgoing unexecuted edges of v_1 and v_2 is re-estimated using

the newly created samples and materialized tables **(lines 15-16)**. Note that previously computed weights are also updated: the edges are re-sampled using as input the newly materialized sample tables. This is a crucial feature of ROX: simply adjusting the already computed weights by, for instance, multiplying with the join hit ratio of the executed path would not suffice as this implies an independence assumption. By re-sampling, ROX is able to detect and naturally adapt its optimization decisions to the arbitrary correlations existing between the vertices in the join graph.

We now describe the ExecutionDirection function.

The ExecutionDirection Function

Given an edge $e = (v_1, v_2)$, the ExecutionDirection function, presented in Algorithm 4, determines the edge's execution direction. If the full tables of the two vertices of e are materialized, then the execution can be accomplished in two ways: either with $TBL(v_1)$ and $TBL(v_2)$ as respectively the left and right operands (*i.e.* $TBL(v_1) \bowtie TBL(v_2)$) or the other way around (*i.e.* $TBL(v_2) \bowtie TBL(v_1)$). The best execution direction to choose is the one with the smallest execution time. To estimate the execution time of the two strategies, the ExecutionDirection function samples the edge in both directions, takes note of the execution time of each sampling operation, and extrapolates each of the measured times to estimate the processing time of an execution using the full tables **(lines 2-10)**. Therefore, the edge e is sampled using as input a sample chosen first from v_1 and then from v_2 **(lines 3, 5)**. The two sampling operations are timed and the noted times are linearly extrapolated to take into account all the tuples in the full table of the left operand **(lines 4, 6)**. The time of the two sampling operations is compared, and the execution direction is chosen to be the same as the direction of the sampling operation that has the smaller estimated execution time **(lines 7-10)**. We stress that one of the two sampling operations executed in this algorithm might already have been performed during the previous optimization step. Therefore, with some additional engineering, the redundant sampling operation can be optimized away. We chose to explain the ExecutionDirection function with the avoidable sampling operation for readability reasons.

If the full table of only one of the two vertices is materialized, then the vertex with the materialized full table is used as the left operand for the execution **(lines 11-14)**. The join can then be executed using for instance an index built on the right operand.

Example 4.2.2. We explain the linear extrapolation of the measured times performed by the algorithm using the following example. Let the edge (v_1, v_2) be sampled with the following operation: $t_1 = \texttt{TIME}\Big((S, hr) \leftarrow$

Algorithm 4: ExecutionDirection

INPUT : Edge $e = (v_1, v_2)$
OUTPUT: Vertex List $(lopd, ropd)$

1 Vertex List $(lopd, ropd) = (\texttt{NULL}, \texttt{NULL})$;
2 IF $TBL(v_1) \neq \texttt{NULL} \wedge TBL(v_2) \neq \texttt{NULL}$ THEN
3 Time $t_1 = \texttt{TIME}\Big((S, hr) \leftarrow \trianglerighteq_\tau(op(e, SMPL(v_1), TBL(v_2)))\Big)$;
4 $t_1 = t_1 \times \frac{hr \times card(v_1)}{|S|}$;
5 Time $t_2 = \texttt{TIME}\Big((S, hr) \leftarrow \trianglerighteq_\tau(op(e, SMPL(v_2), TBL(v_1)))\Big)$;
6 $t_2 = t_2 \times \frac{hr \times card(v_2)}{|S|}$;
7 IF $t_1 \leq t_2$ THEN
8 $(lopd, ropd) = (v_1, v_2)$;
9 ELSE
10 $(lopd, ropd) = (v_2, v_1)$;
11 ELSE IF $TBL(v_1) \neq \texttt{NULL}$ THEN
12 $(lopd, ropd) = (v_1, v_2)$;
13 ELSE
14 $(lopd, ropd) = (v_2, v_1)$;
15 RETURN $(lopd, ropd)$;

$\trianglerighteq_{200}(op(e, SMPL(v_1), TBL(v_2)))\Big)$. We suppose the following: $|SMPL(v_1)| = 100$, $card(v_1) = 20000$, $|S| = 200$, $hr = 4$, and $t_1 = 1.5msec$. Since the estimated join hit ratio hr is equal to 4, then the result size of the join associated with (v_1, v_2) is estimated to be equal to $hr \times card(v_1) = 80000$. It took $1.5msec$ to generate the portion of 200 tuples (the set S) from the full result, therefore generating all the 80000 tuples in the full result will require $1.5 \times \frac{hr \times card(v_1)}{|S|} = 60msec$.

We have presented the ROX algorithm, explaining the alternation of its optimization and execution steps and describing the way it acquires and updates its knowledge about the vertices and edges in the join graph. We have explained in detail the execution phases of ROX but we have not elaborated yet on the chain sampling technique which is the core of the optimization phases. The next section gives a detailed description of chain sampling and its algorithm.

4.3 Chain Sampling

The exploration of the join graph search space is performed through chain sampling. Chain sampling takes as input the edge with the smallest weight $e = (v_1, v_2)$, and analyzes the path segments branching from one of the vertices of e in search for a path p that is ***superior*** to all explored paths. The superiority of p implies that the execution of p followed by any of the other explored paths p' is cheaper than executing p' and then p. In fact, the input edge e is suspected to be only a *local minimum* due to potentially existing correlations between the joined vertices. The function ChainSample invests a small amount of time to climb the hill in search for a *global minimum*, "making sure" that the existing correlations are detected and that the chain of operators that might produce an intermediary result with smaller cardinality is spotted and returned for execution.

Given a vertex v of the edge e, the path segments branching from v are sampled iteratively in a breadth-first manner. In every iteration, new path segments are created by sampling one additional unexecuted edge in every possible direction. Chain sampling assigns to every path a set of properties which are used to keep track of the sampling process and to compare the paths. At the end of every sampling iteration, a stopping condition compares the properties of the paths to check if one path is absolutely superior to all the other explored paths. If such a path is detected, then the exploration can be safely halted and the chosen path is returned for execution. If at the end of every iteration no ***absolutely superior*** path is found, chain sampling progresses until all the edges in the join graph are sampled. Then the SuperiorPath function chooses the path p that is superior to all the explored paths so far, and returns p for execution.

Before we give a detailed description of the chain sampling algorithm, we define the sampling operations executed during chain sampling and the properties assigned to the explored paths.

4.3.1 Sampling Operations Executed during Chain Sampling

Given a join graph G, we suppose that during one of the optimization steps of ROX, the path segment $p = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$ is explored through chain sampling. The edges in p are sampled consecutively, with each chain sampling iteration sampling exactly one edge at a time. The sequence of edges are sampled by executing the following sampling operations (for more information on cutoff-sampling a sequence of joins, refer back to Section 3.2.2):

$$
\begin{aligned}
(S_2, hr_1) &\leftarrow \trianglerighteq_\tau(SMPL(v_1) \bowtie TBL(v_2)) \\
(S_3, hr_2) &\leftarrow \trianglerighteq_\tau(S_2 \bowtie TBL(v_3))
\end{aligned}
$$

Iteration	Path	Sampled Edge	Sampling Operation
Iteration 1	p_1	(v_2, v_1)	$(S_{11}, hr_{11}) \leftarrow \trianglerighteq_\tau(SMPL(v_2) \bowtie TBL(v_1))$
	p_2	(v_2, v_3)	$(\mathbf{S_{21}}, hr_{21}) \leftarrow \trianglerighteq_\tau(SMPL(v_2) \bowtie TBL(v_3))$
	p_3	(v_2, v_5)	$(\mathbf{S_{31}}, hr_{31}) \leftarrow \trianglerighteq_\tau(SMPL(v_2) \bowtie TBL(v_5))$
Iteration 2	p_4	(v_3, v_4)	$(S_{22}, hr_{22}) \leftarrow \trianglerighteq_\tau(\mathbf{S_{21}} \bowtie TBL(v_4))$
	p_5	(v_5, v_6)	$(S_{32}, hr_{32}) \leftarrow \trianglerighteq_\tau(\mathbf{S_{31}} \bowtie TBL(v_6))$
	p_6	(v_5, v_7)	$(\mathbf{S_{42}}, hr_{42}) \leftarrow \trianglerighteq_\tau(\mathbf{S_{31}} \bowtie TBL(v_7))$
Iteration 3	p_7	(v_7, v_8)	$(S_{43}, hr_{43}) \leftarrow \trianglerighteq_\tau(\mathbf{S_{42}} \bowtie TBL(v_8))$

Figure 4.5 The sampling operations executed for every path segment at every iteration of the chain sampling process illustrated in Figure 4.3. The input sample to the sampling operation performed during the first iteration is the sample table of the start vertex v_2. The subsequent operations use as input sample the output generated by the sampling operation executed during the previous iteration. The generated output and their usage as input samples are indicated in bold.

$$\begin{array}{rcl} & \vdots & \\ (S_{n-1}, hr_{n-2}) & \leftarrow & \trianglerighteq_\tau(S_{n-2} \bowtie TBL(v_{n-1})) \\ (S_n, hr_{n-1}) & \leftarrow & \trianglerighteq_\tau(S_{n-1} \bowtie TBL(v_n)) \end{array}$$

The first chain sampling iteration uses as input the sample table associated with the starting vertex v_1. Every subsequent iteration uses as input sample the result generated by the sampling operation executed during the previous iteration.

Continuation of Example 4.1.3. We reconsider the join graph and chain sampling process presented in Example 4.1.3 and Figure 4.3. The edge with the minimum weight was identified to be (v_2, v_5) and the vertex from which to start the exploration was chosen to be v_2. Figure 4.5 illustrates the sampling operations executed at every iteration and for every path segment. The sampling operations executed during the first iteration use as input the sample table associated with the start vertex v_2 of the edge with the smallest weight $e = (v_2, v_5)$. During the subsequent iterations, the input sample of the sampling operations corresponds to the output generated by the sampling operation executed during the previous iteration (the generated output and their usage as input samples are indicated in bold). This is an important feature of chain sampling, as it allows the detection of arbitrary correlations between the joined vertices.

In the following, we define the properties that chain sampling assigns to every explored path p.

4.3.2 Properties Associated with Path Segments

Given a path segment p, we define the following properties associated with p:

hr : $Path \rightarrow \mathbb{R}^+$

$hr(p)$ is the hit ratio of path p. It represents the hit ratio of the sequence of join operators in p.

$cost$: $Path \rightarrow \mathbb{R}^+$

$cost(p)$ is the cost of the path p. It represents the cumulative estimated cardinality of the intermediate result generated by the consecutive execution of the join operators in p.

I : $Path \rightarrow Table$

$I(p)$ is the input sample of p. It represents the sample table to be used as input to the sampling operation that will be executed during the next iteration of chain sampling.

$StartVertex$: $Path \rightarrow Vertex$

$StartVertex(p)$: is the start vertex of p. It represents the vertex from which the next chain sampling iteration resumes the search space exploration.

The hit ratio of a path segment is equal to the hit ratio of the sequence of joins it contains. We have explained in Section 3.2.2 that the hit ratio of a sequence of joins can be derived by multiplying the hit ratio of each individual join when the joins are sampled consecutively. Therefore, the hit ratio of a path segment is equal to the multiplication of the hit ratio of each of its individual join operators.

We now explain the cost of a path. Given a join graph G, we suppose that the following path segment $p = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$ is

sampled during the chain sampling process initiated by one of the optimization steps of ROX. To ease the explanation, we represent p in terms of relations and join operators instead of edges. The path p is then equal to the sequence of joins $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$ where $R_i = TBL(v_i)$ for $i \in [1, n]$. The path p is then sampled with the following operations:

$$\begin{aligned}
(S_2, hr_1) &\leftarrow \trianglerighteq_\tau (SMPL(v_1) \bowtie R_2) \\
(S_3, hr_2) &\leftarrow \trianglerighteq_\tau (S_2 \bowtie R_3) \\
&\vdots \\
(S_{n-1}, hr_{n-2}) &\leftarrow \trianglerighteq_\tau (S_{n-2} \bowtie R_{n-1}) \\
(S_n, hr_{n-1}) &\leftarrow \trianglerighteq_\tau (S_{n-1} \bowtie R_n)
\end{aligned}$$

We have defined the cost of a path to be equal to the cumulative estimated cardinality of the result generated by the consecutive execution of its joins. We therefore have the following:

$$\begin{aligned}
cost(p) &= card(R_1 \bowtie R_2) + card(R_1 \bowtie R_2 \bowtie R_3) + \cdots + \\
&\quad card(R_1 \bowtie R_2 \ldots \bowtie R_n) \\
&= |R_1| \times hr(R_1 \bowtie R_2) + |R_1| \times hr(R_1 \bowtie R_2 \bowtie R_3) + \cdots + \\
&\quad |R_1| \times hr(R_1 \bowtie R_2 \ldots \bowtie R_n) \\
&\qquad \text{by Definition 3.2.12} \\
&\approx |R_1| \times hr_1 + |R_1| \times \prod_{i=1}^{2} |Big(hr_i) + \cdots + |R_1| \times \prod_{i=1}^{n-1} (hr_i) \\
&\qquad \text{by Definition 3.2.11} \\
&= |R_1| \times \left(hr_1 + \prod_{i=1}^{2} (hr_i) + \cdots + \prod_{i=1}^{n-1} (hr_i) \right) \\
&= |R_1| \times \sum_{j=1}^{n-1} \left(\prod_{i=1}^{j} (hr_i) \right) \\
&\approx card(v_1) \times \sum_{j=1}^{n-1} \left(\prod_{i=1}^{j} (hr_i) \right)
\end{aligned}$$

4.3.2.1 Formal Definition of the Path Properties

We now give a formal definition of the introduced path properties. Given a join graph G, we suppose that the following path segment

$p = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$ is sampled during the chain sampling process initiated by one of the optimization steps of ROX. The edges in p are sampled by consecutively executing the following sampling operations:

$$\begin{aligned} (S_2, hr_1) &\leftarrow \trianglerighteq_\tau(SMPL(v_1) \bowtie TBL(v_2)) \\ (S_3, hr_2) &\leftarrow \trianglerighteq_\tau(S_2 \bowtie TBL(v_3)) \\ &\vdots \\ (S_{n-1}, hr_{n-2}) &\leftarrow \trianglerighteq_\tau(S_{n-2} \bowtie TBL(v_{n-1})) \\ (S_n, hr_{n-1}) &\leftarrow \trianglerighteq_\tau(S_{n-1} \bowtie TBL(v_n)) \end{aligned}$$

We then have the following:

$$\begin{aligned} hr(p) &= \prod_{i=1}^{n-1}(hr_i) \\ cost(p) &= card(v_1) \times \sum_{j=1}^{n-1}\left(\prod_{i=1}^{j}(hr_i)\right) \\ I(p) &= S_n \\ StartVertex(p) &= v_n \end{aligned}$$

In the subsequent sampling iteration, one of the outgoing edges of the vertex $StartVertex(p)$ is sampled and a new path p' is created. We suppose the sampled edge to be (v_n, v_{n+1}). Path p' contains the edges of p and the newly sample edge (v_n, v_{n+1}). The sampling operation of (v_n, v_{n+1}) uses as input the sample table $I(p)$ associated with p:

$$(S_{n+1}, hr_n) \leftarrow \trianglerighteq_\tau(S_n \bowtie TBL(v_{n+1}))$$

The properties of the path p' are assigned the following values:

$$\begin{aligned} hr(p') &= \prod_{i=1}^{n}(hr_i) \\ cost(p') &= card(v_1) \times \sum_{j=1}^{n}\left(\prod_{i=1}^{j}(hr_i)\right) \\ I(p') &= S_{n+1} \\ StartVertex(p') &= v_{n+1} \end{aligned}$$

We note that the hit ratio and cost of the path p' can also be computed iteratively using the properties associated with the path p as shown below:

$$\begin{aligned} hr(p') &= \prod_{i=1}^{n}(hr_i) \\ &= \prod_{i=1}^{n-1}(hr_i) \times hr_n \\ &= hr(p) \times hr_n \\ cost(p') &= card(v_1) \times \sum_{j=1}^{n}\left(\prod_{i=1}^{j}(hr_i)\right) \\ &= card(v_1) \times \sum_{j=1}^{n-1}\left(\prod_{i=1}^{j}(hr_i)\right) + card(v_1) \times \prod_{i=1}^{n}(hr_i) \\ &= cost(p) + card(v_1) \times hr(p') \end{aligned}$$

Therefore, given a path p' newly created by extending path p with the edge e, we note that the hit ratio of p' is equal to the hit ratio of p multiplied with the hit ratio of the sampled edge e, and that the cost of p' is the sum of the cost of p and the estimated result size of the edges in the p'.

Note 4.3.1. We note that

$$cost(p_i) = 0 \quad \Longrightarrow \quad hr(p_i) = 0$$

In fact, if $cost(p_i) = 0$ then no intermediate tuples have been generated by any join operator in p_i. A join operator which generates an empty result has a hit ratio equal to 0. Therefore, $hr(p_i)$, the multiplication of the hit ratio of all the joins in p_i, is equal to 0.

Continuation of Example 4.1.3. We reconsider the join graph and chain sampling process presented in Example 4.1.3 and Figure 4.3. The edge with the minimum weight was identified to be (v_2, v_5) and the vertex from which to start the exploration was chosen to be v_2. Figure 4.6 and Figure 4.7 illustrate the value of the properties associated with the created paths at every iteration. We refer the reader to Figure 4.5 for the sampling operations executed at every iteration.

Iteration 0**:** Before chain sampling starts (iteration 0), an empty path p is initialized with the properties $StartVertex(p)$ and $I(p)$ respectively equal to the starting vertex v_2 and the sample table $SMPL(v_2)$ associated with v_2.

Iteration	Path	hr	I	$StartVertex$
Iteration 0	p	-	$SMPL(v_2)$	v_2
Iteration 1	p_1	hr_{11}	S_{11}	v_1
	p_2	hr_{21}	S_{21}	v_3
	p_3	hr_{31}	S_{31}	v_5
Iteration 2	p_4	$hr(p_2) \times hr_{22}$	S_{22}	v_4
	p_5	$hr(p_3) \times hr_{32}$	S_{32}	v_6
	p_6	$hr(p_3) \times hr_{42}$	S_{42}	v_7
Iteration 3	p_7	$hr(p_6) \times hr_{43}$	S_{43}	v_8

Figure 4.6 The hit ratio, input sample, and start vertex properties of every path segment explored during the chain sampling process illustrated in Figure 4.3. The executed sampling operations are listed in Figure 4.5.

Iteration	Path	$cost$
Iteration 0	p	-
Iteration 1	p_1	$card(v_2) \times hr(p_1)$
	p_2	$card(v_2) \times hr(p_2)$
	p_3	$card(v_2) \times hr(p_3)$
Iteration 2	p_4	$cost(p_2) + card(v_2) \times hr(p_4)$
	p_5	$cost(p_3) + card(v_2) \times hr(p_5)$
	p_6	$cost(p_3) + card(v_2) \times hr(p_6)$
Iteration 3	p_7	$cost(p_6) + card(v_2) \times hr(p_7)$

Figure 4.7 The cost property of every path segment explored during the chain sampling process illustrated in Figure 4.3. The executed sampling operations are listed in Figure 4.5.

Iteration 1: During the first iteration, all the unexecuted edges of v_2 are sampled using as input the sample table $I(p)$. Since the number of edges is three, three new paths p_1, p_2, and p_3 are created. Path p_1, p_2, and p_3 contain respectively the sampled edges (v_2, v_1), (v_2, v_3), (v_2, v_5). The hit ratio and cost of each path are equal to respectively the hit ratio and the estimated result size of the sampled edge. The input sample $I(p_1)$, $I(p_2)$, $I(p_3)$ is equal to the output table generated by the corresponding sampling operation. The vertex from which to start the search exploration in the next iteration for each of p_1, p_2, and p_3 is equal to the end vertex of the sampled edge: respectively v_1, v_3, and v_5.

Iteration 2: The second iteration attempts to sample the unexecuted edges branching from the *StartVertex* of each of the three defined paths. In case of p_1, its start vertex v_1 has no outgoing unexecuted edges. The start vertex of p_2 has one outgoing edge (v_3, v_4), therefore a new path p_4 is created ($p_4 = p_2 \cup \{(v_3, v_4)\}$). The start vertex of p_3 has two outgoing unexecuted edges (v_5, v_6), and (v_5, v_7), therefore two new paths are created: $p_5 = p_3 \cup \{(v_5, v_6)\}$ and $p_6 = p_3 \cup \{(v_5, v_7)\}$. The properties associated with p_4, p_5, and p_6 are updated correspondingly. The input sample is equal to the output generated by the sampling operation of the edge, and the start vertex of p_4, p_5, and p_6 is equal to the end vertex of the sampled edge: respectively v_4, v_5, and v_6.

Iteration 3: In the third iteration, the start vertex of only path p_6 has an outgoing unexecuted edge (v_7, v_8), therefore one additional path p_7 is created and its properties are defined. Chain sampling stops since there are no more unexecuted edges to explore.

We conclude the following: given an explored path p and an unexecuted edge (v, v') branching from the start vertex of path p, chain sampling samples the edge (v, v') using as input the sample table $I(p)$. A new path p' is created: $p' = p \cup \{(v, v')\}$. At the end of the sampling iteration, the properties of path p' are defined. The input sample $I(p)$ is updated to the output generated by the sampling operation. The start vertex $StartVertex(p)$ is equal to v'. The hit ratio $hr(p)$ is equal to the hit ratio of p multiplied by the hit ratio of the newly sampled edge (v, v'). The cost $cost(p)$ is equal to the sum of the cost of p and the estimated result size of the sequence of join operators in p'.

As we will see in the next section, in which we present the ChainSample algorithm, chain sampling does not necessarily sample all the unexecuted edges in the graph before determining the path to return for execution. A stopping condition is used to assist chain sampling in deciding whether to initiate a new sampling iteration or to halt the exploration process and return the ***absolutely superior*** path just found.

4.3.3 Chain Sampling Algorithm

The ChainSample function (Algorithm 5) takes as input the edge $e = (v_1, v_2)$ with the smallest weight. It explores the collection of edges around e and returns for execution the path p that is found to be superior to all the explored paths.

First the vertex of e used as a starting point of the chain sampling exploration is determined **(line 1)** using the function StartingVertex, explained shortly. When the *start_vertex* is chosen, a first path p is initialized with

Algorithm 5: ChainSample

```
   INPUT  : Edge e = (v1, v2)
   OUTPUT: Path p

 1 Vertex start_vertex ← StartingVertex(e) ; // Determine the
   vertex from which to start the chain sampling exploration

   // Initialization of path
 2 Path p ← {};
 3 StartVertex(p) ← start_vertex;
 4 I(p) ← SMPL(start_vertex);
 5 Path List all_paths = {};
 6 Path List new_paths = {p};

   // Increase the cutoff-sampling limit by incr.  In the
      prototype incr = τ, but another value can be used
 7 int incr ← τ;
 8 int LIMIT ← τ + incr;

   // Iterative exploration of path segments
 9 while ∃ more edges to sample do
10 |   Path List current_paths ← new_paths;
11 |   new_paths = {};
12 |   foreach Path p ∈ current_paths do
13 |   |   Vertex v ← StartVertex(p);
14 |   |   foreach Edge e′ = (v, v′) ∈ (edges⁻(v) \ p) do
15 |   |   |   Path p′ ← p ∪ {e′};
16 |   |   |   (S, hr) ← ⊵LIMIT(op(e′, I(p), TBL(v′)));
17 |   |   |   hr(p′) ← hr(p) × hr;
18 |   |   |   cost(p′) ← cost(p) + hr(p′) × card(start_vertex);
19 |   |   |   I(p′) ← S;
20 |   |   |   StartVertex(p′) ← v′;
21 |   |   |   new_paths.Insert(p′);
22 |   all_paths = all_paths ∪ new_paths;
23 |   Path p ← StoppingCondition(new_paths, all_paths);
24 |   if p ≠ NULL then
25 |   |   return p ; // Chain sampling safely halted since an
   |   |   absolutely superior path is found
26 |   LIMIT ← LIMIT + incr ; // Increase cutoff-sampling limit

27 Path p ← SuperiorPath(all_paths) ; // No absolutely superior
   path was found, so we return the superior path
28 return p;
```

$StartVertex(p) = start_vertex$ and $I(p) = SMPL(start_vertex)$. Path p is added to the list *new_paths* **(lines 2-6)**. Additionally, the LIMIT used in the cutoff-sampling operations of chain sampling is initialized. We choose to increase the initial cutoff limit (τ) by *incr* to allow for a bigger and therefore more representative generated result **(lines 7-8)**. This is performed to counteract the propagation of possible estimation errors during chain sampling. In the prototype, we choose *incr* to be τ, but other values might also be used. In fact, we have not explored what the best value of *incr* is.

Next, chain sampling starts its iterative exploration of path segments **(lines 9-26)**. For each path p in the list *current_paths* ($current_paths = new_paths$), all the outgoing unexecuted edges of the start vertex of p that are not yet contained in the path segment p are explored. For each such unexecuted edge $e' = (v, v')$, the following steps are performed:

1. A new path p' is created, and initialized to all the edges in p and the edge e' **(line 15)**.
2. The edge e' is cutoff-sampled using as left operand the input sample $I(p)$ associated with path p **(line 16)**.
3. The hit ratio of p' is assigned the value of the multiplication of $hr(p)$ with the hit ratio returned by the sampling operation of e' **(line 17)**.
4. The cost of p' is initialized to the sum of $cost(p)$ and the estimated result size of the sequence of joins in p' **(line 18)**.
5. The input sample of p' is equal to the output generated by the sampling operation of e' **(line 19)**.
6. The start vertex of p' is equal to the end vertex v' of the sampled edge e' **(line 20)**.
7. The path p' is added to the list *new_paths* **(line 21)**.

The above process is repeated for every path $p \in current_paths$.

After sampling the next unexecuted edges of every path p in the list *current_paths*, the StoppingCondition function **(line 23)** compares the hit ratio and cost properties of the newly created paths to all the paths sampled so far to determine if it is safe to stop the chain sampling process or if a new exploration iteration should be initiated. The function searches for a path p that is ***absolutely superior*** to all other paths: the execution of p followed by the execution of any other path p' is cheaper than executing p' alone. If such a path p exists then chain sampling can be safely stopped and p is returned for execution **(lines 24-25)**. We explain the stopping condition in Section 4.3.5.

If none of the paths in the list *new_paths* is absolutely superior to the others, chain sampling initiates a new sampling iteration. It also increases by *incr* the value LIMIT to be used in the cutoff-sampling operations which will

be performed during the new exploration iteration **(line 26)**. This results in a larger and more representative output generated by the sampling operations, therefore counteracting the possible creation and propagation of estimation errors possibly introduced during the previous sampling iterations.

If no absolutely superior path is found at the end of every chain sampling iteration and all path segments in the join graph are sampled, the SuperiorPath function compares all the explored paths to choose the ***superior*** path among all **(line 27)**. The chosen path is returned for execution **(line 28)**. The SuperiorPath function is explained in Section 4.3.4.

The StartingVertex Function

The StartingVertex function is presented in Algorithm 6. The function takes as input an edge $e = (v_1, v_2)$ and returns the vertex of e from which to start the chain sampling exploration. Three criteria determine the choice of the start vertex (*start_vertex*): the availability of materialized samples associated with the edge's vertices, the number of outgoing unexecuted edges of each vertex, and the estimated cardinality of each vertex. If the edge's two vertices v_1 and v_2 both have a materialized sample table, and more than one outgoing unexecuted edge, then the start vertex is chosen to be the vertex with the smaller estimated cardinality **(lines 2-7)**. The reasoning behind our selection is that picking the sample from a smaller table results in a more representative sample set to use as input in the first iteration of chain sampling. This in turn leads to the generation of a more representative result which will be used as input in the next iterations. If only one of the vertices has more than one outgoing unexecuted edge, then that vertex is used to start the exploration **(lines 8-11)**. Finally, if only one of the vertices has a materialized sample table, then the only possible choice is to pick the sample to be used as input in chain sampling from that vertex **(lines 12-16)**.

Given the edge e with the smallest weight, the chain sampling process is initiated by ROX to explore the join graph in search for a ***superior*** path using the edge e as starting point of exploration. When all the edges in the graph are sampled, the SuperiorPath function determines the path that is superior to all the explored paths and returns it for execution. In order to make the search more efficient, the StoppingCondition function is used after each sampling iteration to detect the existence of an ***absolutely superior*** path p to return for execution. The stopping condition guarantees that if chain sampling would progress, no path ***superior*** to p would be found, and the SuperiorPath function will at the end of chain sampling still choose p for execution. Next, we explain the SuperiorPath function and then describe the proposed stopping condition.

Algorithm 6: STARTINGVERTEX

```
INPUT  : Edge e = (v1, v2)
OUTPUT: Vertex start_vertex

1  Vertex start_vertex ← null;
2  IF SMPL(v1) ≠ NULL ∧ SMPL(v2) ≠ NULL THEN
3  |  IF |edges⁻(v1)| > 1 ∧ |edges⁻(v2)| > 1 THEN
4  |  |  IF card(v1) ≤ card(v2) THEN
5  |  |  |  start_vertex ← v1;
6  |  |  ELSE
7  |  |  |  start_vertex ← v2;
8  |  ELSE IF |edges⁻(v1)| > 1 THEN
9  |  |  start_vertex ← v1;
10 |  ELSE
11 |  |  start_vertex ← v2;
12 ELSE
13 |  IF SMPL(v1) ≠ NULL THEN
14 |  |  start_vertex = v1;
15 |  ELSE
16 |  |  start_vertex = v2;
17 RETURN start_vertex;
```

4.3.4 The SUPERIORPATH Function

Before we start explaining the SUPERIORPATH function, we introduce the following notation.

Definition 4.3.2. Given two paths p_i and p_j explored during chain sampling, we define the notation $cost(p_j|p_i)$ as follows:

$$cost(p_j|p_i) \quad \text{is the cost of path } p_j \text{ after having executed the path } p_i$$

We estimate the value of $cost(p_j|p_i)$ to be:

$$cost(p_j|p_i) \quad = \quad cost(p_j) \times hr(p_i)$$

The above formula estimates the cost of executing the sequence of joins in path p_j using the data returned by the execution of p_i. The intuition behind the formula is as follows. Suppose that the vertex v in the join graph is the starting point of the chain sampling exploration and that $card(v) = X$.

Algorithm 7: SuperiorPath

INPUT : Path List *all_paths*
// *all_paths* = `list of all path segments that have been explored during chain sampling`
OUTPUT: Path p

1 FOREACH Path $p_i \in all_paths$ DO
2   IF $cost(p_i) + cost(p_j|p_i) \leq cost(p_j) + cost(p_i|p_j)\ \forall\ p_j \in all_paths \mid i \neq j$ THEN
3   RETURN p_i ; // `Path` p_i `is superior to all the other paths in the list` *all_paths*`, therefore` p_i `is returned for execution.`

Knowing that the hit ratio of p_i is $hr(p_i)$, we estimate that the execution of the operators in p_i using as input the full table $TBL(v)$ will generate a result of size $X \times hr(p_i)$. We also know that the cost of executing p_j using as input the X tuples in table $TBL(v)$ is $cost(p_j)$. Therefore, the execution cost of path p_j after having executed p_i, *i.e.* using as input the output of size $X \times hr(p_i)$ generated by p_i, is estimated to be the value $cost(p_j) \times hr(p_i)$. For example, if $cost(p_j)$ was estimated to be equal to 1000 and the execution of p_i is estimated to reduce the intermediate result by half (*i.e.* $hr(p_i) = 0.5$), then the cost of executing p_j after p_i is estimated to be equal to 500.

The SuperiorPath function is given in Algorithm 7. It takes as input the list *all_paths* which contains all the path segments that have been explored during chain sampling. The algorithm compares in a pairwise fashion the paths in *all_paths* to find the path that is ***superior*** to the others. Given a path $p_i \in all_paths$, p_i is compared to every other path $p_j \in all_paths$ using the following inequality **(line 2)**:

$$cost(p_i) + cost(p_j|p_i) \quad \leq \quad cost(p_j) + cost(p_i|p_j) \tag{4.1}$$

The inequality checks if the execution of path p_i succeeded by the execution of any other path p_j ($p_j \in all_paths$) is cheaper than first executing p_j followed by p_i. If the path p_i satisfies the above inequality for all other paths $p_j \in all_paths$, then p_i is a superior path and is returned for execution **(line 3)**.

It may seem like a large number of path comparisons is performed, but a path p_i which fails to be superior to any other path is directly disregarded by the algorithm, and the next path in the list *all_paths* is then checked for

superiority. Moreover, the comparisons can be implemented in an efficient manner (*e.g.* by first sorting the paths), hence reducing the amount of time spent on the superiority checking.

Definition 4.3.3. Given two paths p_i and p_j, we define the superiority relation $\preceq$ as follows:

$$p_i \preceq p_j \quad \equiv \quad cost(p_i) + cost(p_j|p_i) \leq cost(p_j) + cost(p_i|p_j)$$

Therefore, given a set of paths P, we say that the path p_i is superior in P if and only if $p_i \preceq p_j \qquad \forall\, p_j \in P \mid i \neq j$.

Definition 4.3.4. Given two paths p_i and p_j, we say that p_i and p_j are equivalent if and only if the following three statements hold:

1. $p_i \neq p_j$
2. $p_i \preceq p_j$
3. $p_j \preceq p_i$

In other words, p_i and p_j are equivalent if and only if:

$$p_i \neq p_j \text{ and}$$
$$cost(p_i) + cost(p_j|p_i) = cost(p_j) + cost(p_i|p_j)$$

Lemma 4.3.5. *Given a set P of paths explored during chain sampling, we claim that there exists at least one superior path $p_i \in P$.*
In other words, $\exists p_i \in P\ \forall\, p_j \in P \mid i \neq j \implies p_i \preceq p_j$

Proof. Given a set P of paths explored during chain sampling, we prove the claim in Lemma 4.3.5 by proving that the set of paths P is linearly ordered modulo path equivalences under the relation $\preceq$. We therefore need to prove that the relation $\preceq$ satisfies the following three properties for all paths in P:

1. **Reflexivity:**

$$p_i \preceq p_i \qquad \forall\, p_i \in P$$

2. **Totality:**

$$p_i \preceq p_j \ \vee\ p_j \preceq p_i \qquad \forall\, p_i, p_j \in P$$

3. **Transitivity:**

$$p_i \preceq p_j \ \wedge\ p_j \preceq p_k \implies p_i \preceq p_k \qquad \forall\, p_i, p_j, p_k \in P$$

In the following, we prove each of the above properties.

Reflexivity

Given a path $p_i \in P$, we have:

$$\begin{aligned} & cost(p_i) + cost(p_i) \times hr(p_i) &=& \quad cost(p_i) + cost(p_i) \times hr(p_i) \\ \Longrightarrow\ & cost(p_i) + cost(p_i|p_i) &=& \quad cost(p_i) + cost(p_i|p_i) \\ \Longrightarrow\ & cost(p_i) + cost(p_i|p_i) &\leq& \quad cost(p_i) + cost(p_i|p_i) \\ \Longrightarrow\ & p_i &\preceq& \quad p_i \end{aligned}$$

Totality

Given any two paths $p_i, p_j \in P$, we know that $cost(p_i) \in \mathbb{R}^+$, $cost(p_j) \in \mathbb{R}^+$, $hr(p_i) \in \mathbb{R}^+$, $hr(p_j) \in \mathbb{R}^+$. Therefore, we have:

$$\begin{aligned} & cost(p_i) + cost(p_j) \times hr(p_i) &\in& \quad \mathbb{R}^+ \\ \Longrightarrow\ & cost(p_i) + cost(p_j|p_i) &\in& \quad \mathbb{R}^+ \end{aligned} \tag{4.2}$$

$$\begin{aligned} & cost(p_j) + cost(p_i) \times hr(p_j) &\in& \quad \mathbb{R}^+ \\ \Longrightarrow\ & cost(p_j) + cost(p_i|p_j) &\in& \quad \mathbb{R}^+ \end{aligned} \tag{4.3}$$

$$\begin{aligned} (4.2) \wedge (4.3) \quad & \Longrightarrow \begin{cases} cost(p_i) + cost(p_j|p_i) \ \leq \ cost(p_j) + cost(p_i|p_j) & \text{or} \\ cost(p_j) + cost(p_i|p_j) \ \leq \ cost(p_i) + cost(p_j|p_i) \end{cases} \\ & \Longrightarrow \begin{cases} p_i \preceq p_j & \text{or} \\ p_j \preceq p_i \end{cases} \end{aligned}$$

Transitivity

Given three paths $p_i, p_j, p_k \in P$, we suppose that: $p_i \preceq p_j \ \wedge \ p_j \preceq p_k$, and we need to prove that $p_i \preceq p_k$.
To make the proof easy to follow we note the following:

$$\begin{aligned} cost(p_i) &= a & \quad hr(p_i) &= x \\ cost(p_j) &= b & \quad hr(p_j) &= y \\ cost(p_k) &= c & \quad hr(p_k) &= z \\ & \textbf{where} & \quad & a, b, c, x, y, z \in \mathbb{R}^+ \end{aligned}$$

We therefore have the following:

$$
\begin{aligned}
& p_i \preceq p_j \\
\Longrightarrow\ & cost(p_i) + cost(p_j|p_i) \ \leq\ cost(p_j) + cost(p_i|p_j) \\
\Longrightarrow\ & cost(p_i) + cost(p_j) \times hr(p_i) \ \leq\ cost(p_j) + cost(p_i) \times hr(p_j) \\
\Longrightarrow\ & a + bx \ \leq\ b + ay \\
\Longrightarrow\ & b - bx + ay - a \ \geq\ 0 \\
\Longrightarrow\ & b(1-x) + a(y-1) \ \geq\ 0
\end{aligned} \tag{4.4}
$$

$$
\begin{aligned}
& p_j \preceq p_k \\
\Longrightarrow\ & cost(p_j) + cost(p_k|p_j) \ \leq\ cost(p_k) + cost(p_j|p_k) \\
\Longrightarrow\ & cost(p_j) + cost(p_k) \times hr(p_j) \ \leq\ cost(p_k) + cost(p_j) \times hr(p_k) \\
\Longrightarrow\ & b + cy \ \leq\ c + bz \\
\Longrightarrow\ & c - cy + bz - b \ \geq\ 0 \\
\Longrightarrow\ & c(1-y) + b(z-1) \ \geq\ 0
\end{aligned} \tag{4.5}
$$

Given (4.4) and (4.5), we need to prove that

$$
\begin{aligned}
& p_i \preceq p_k \\
\Longrightarrow\ & cost(p_i) + cost(p_k|p_i) \ \leq\ cost(p_k) + cost(p_i|p_k) \\
\Longrightarrow\ & cost(p_i) + cost(p_k) \times hr(p_i) \ \leq\ cost(p_k) + cost(p_i) \times hr(p_k) \\
\Longrightarrow\ & a + cx \ \leq\ c + az
\end{aligned} \tag{4.6}
$$

Before we give the proof, we derive the following.

$$
\begin{aligned}
y \ \leq\ 1 \quad & \equiv \quad a(y-1) \ \leq\ 0 & \\
& \Longrightarrow \quad b(1-x) \ \geq\ 0 & \text{by (4.4)} \\
& \equiv \quad 1-x \ \geq\ 0 & \text{for } b \neq 0 \\
& \equiv \quad x \ \leq\ 1 &
\end{aligned}
$$

$$
\begin{aligned}
x \ \geq\ 1 \quad & \equiv \quad b(1-x) \ \leq\ 0 & \\
& \Longrightarrow \quad a(y-1) \ \geq\ 0 & \text{by (4.4)} \\
& \equiv \quad y-1 \ \geq\ 0 & \text{for } a \neq 0 \\
& \equiv \quad y \ \geq\ 1 &
\end{aligned}
$$

Using a similar proof, we can conclude the same about y and z. We therefore have derived the following:

$$y \leq 1 \implies x \leq 1 \quad \text{for } b \neq 0 \tag{4.7}$$
$$x \geq 1 \implies y \geq 1 \quad \text{for } a \neq 0 \tag{4.8}$$

$$z \leq 1 \implies y \leq 1 \quad \text{for } c \neq 0 \tag{4.9}$$
$$y \geq 1 \implies z \geq 1 \quad \text{for } b \neq 0 \tag{4.10}$$

We now prove that (4.6) holds if we have $x \leq 1 \leq z$. That is:

$$x \leq 1 \leq z \quad \implies \quad (4.6) \tag{4.11}$$

We have

$$\begin{aligned} x \leq 1 \quad &\equiv \quad x - 1 \leq 0 \\ &\equiv \quad c(x-1) \leq 0 \end{aligned}$$

$$\begin{aligned} z \geq 1 \quad &\equiv \quad z - 1 \geq 0 \\ &\equiv \quad a(z-1) \geq 0 \end{aligned}$$

$$\begin{aligned} x \leq 1 \leq z \quad &\implies \quad c(x-1) \leq 0 \leq a(z-1) \\ &\equiv \quad c(x-1) \leq a(z-1) \\ &\equiv \quad cx - c \leq az - a \\ &\equiv \quad a + cx \leq c + az \\ &\equiv \quad (4.6) \end{aligned}$$

Following is the proof of transitivity. The proof is derived by considering several cases.

Case 1: We first examine the case in which the fractions $\frac{z-1}{y-1}$ and $\frac{1-x}{1-y}$ are well-defined and non-negative, *i.e.* $y \neq 1$ and $\Big((x < 1 \wedge y < 1) \vee (x > 1 \wedge y > 1)\Big)$ and $\Big((y < 1 \wedge z < 1) \vee (y > 1 \wedge z > 1)\Big)$. We can therefore multiply the left-hand-side and right-hand side of the inequalities (4.4) and

(4.5) with respectively $\frac{z-1}{y-1}$ and $\frac{1-x}{1-y}$ and add the resulting inequalities. We then get:

$$\begin{aligned}
& \frac{z-1}{y-1}\Big(b(1-x)+a(y-1)\Big) + \frac{1-x}{1-y}\Big(c(1-y)+b(z-1)\Big) \ \geq \ 0 \\
\equiv \quad & a(z-1)+b(1-x)(z-1)\left(\frac{1}{y-1}+\frac{1}{1-y}\right)+c(1-x) \ \geq \ 0 \\
\equiv \quad & a(z-1)+c(1-x) \ \geq \ 0 \\
\equiv \quad & az-a+c-cx \ \geq \ 0 \\
\equiv \quad & a+cx \ \leq \ c+az \\
\equiv \quad & (4.6)
\end{aligned}$$

We have proved that transitivity holds when $\frac{z-1}{y-1}$ and $\frac{1-x}{1-y}$ are well-defined and non negative. It remains to consider the situations in which either or both of $\frac{z-1}{y-1}$ and $\frac{1-x}{1-y}$ are ill-defined or negative, which consist of the following cases:

- **Case 2**: $y = 1$
- **Case 3**: $x \leq 1 \ \wedge \ y \geq 1$
- **Case 4**: $x \geq 1 \ \wedge \ y \leq 1$
- **Case 5**: $y \leq 1 \ \wedge \ z \geq 1$
- **Case 6**: $y \geq 1 \ \wedge \ z \leq 1$

We prove each of the above cases separately. We assume that $a \neq 0$, $b \neq 0$, $c \neq 0$.

Case 2: $y = 1$

$$\begin{aligned}
y = 1 \ \Longrightarrow \ & x \leq 1 \leq z && \text{by (4.7), (4.10)} \\
\Longrightarrow \ & (4.6) && \text{by (4.11)}
\end{aligned}$$

Case 3: $x \leq 1 \ \wedge \ y \geq 1$

$$\begin{aligned}
x \leq 1 \leq y \ \Longrightarrow \ & x \ \leq \ 1 \ \leq \ z && \text{by (4.10)} \\
\Longrightarrow \ & (4.6) && \text{by (4.11)}
\end{aligned}$$

Case 4: $x \geq 1 \wedge y \leq 1$

$$y \leq 1 \leq x \qquad \text{cannot be true} \qquad \text{by (4.8)}$$

Case 5: $y \leq 1 \ \wedge \ z \geq 1$

$$\begin{aligned}
y \leq 1 \leq z \ \Longrightarrow \ & x \leq 1 \leq z && \text{by (4.7)} \\
\Longrightarrow \ & (4.6) && \text{by (4.11)}
\end{aligned}$$

Case 6: $y \geq 1 \ \wedge \ z \leq 1$

$$z \leq 1 \leq y \qquad \text{cannot be true} \qquad \text{by (4.9)}$$

Assuming that $a \neq 0$, $b \neq 0$, $c \neq 0$, we have proven that transitivity also holds when $\frac{z-1}{y-1}$ and $\frac{1-x}{1-y}$ are ill-defined or negative. Therefore, we still have to prove it holds for the following cases:

- **Case 7**: $a = 0$
- **Case 8**: $b = 0$
- **Case 9**: $c = 0$

Case 7: $a = 0$

$$\begin{aligned}
a = 0 \quad &\Longrightarrow \quad x = 0 && \text{by Note 4.3.1} \\
&\Longrightarrow \quad a + cx = 0 && \\
&\Longrightarrow \quad a + cx \leq c && \text{since } c \in \mathbb{R}^+ \\
&\Longrightarrow \quad a + cx \leq c + az && \text{since } az = 0 \\
&\Longrightarrow \quad (4.6) &&
\end{aligned}$$

Case 8: $b = 0$

$$\begin{aligned}
b = 0 \quad &\Longrightarrow \quad y = 0 && \text{by Note 4.3.1} \\
&\Longrightarrow \quad b + ay = 0 && \\
&\Longrightarrow \quad a + bx \leq 0 && \text{by (4.4)} \\
&\Longrightarrow \quad a \leq 0 && \text{since } bx = 0 \\
&\Longrightarrow \quad a = 0 && \text{since } a \in \mathbb{R}^+ \\
&\Longrightarrow \quad (4.6) && \text{Same as } \textbf{Case 7}
\end{aligned}$$

Case 9: $c = 0$

$$\begin{aligned}
c = 0 \quad &\Longrightarrow \quad z = 0 && \text{by (4.1)} \\
&\Longrightarrow \quad c + bz = 0 && \\
&\Longrightarrow \quad b + cy \leq 0 && \text{by (4.5)} \\
&\Longrightarrow \quad b \leq 0 && \text{since } cy = 0 \\
&\Longrightarrow \quad b = 0 && \text{since } b \in \mathbb{R}^+ \\
&\Longrightarrow \quad (4.6) && \text{Same as } \textbf{Case 8}
\end{aligned}$$

By this, we have covered all possible cases and therefore have proven that transitivity holds for the relation $\preceq$.
We have proven that $\preceq$ satisfies reflexivity, totality, and transitivity. Therefore, the claim in Lemma 4.3.5 holds. □

Note about Comparing a Path with its Subpath

Given a set S of paths explored during chain sampling, the inequality condition used in Algorithm 7 **(line 2)** compares pairs of paths in S to identify the superior path in the set. We restate the used inequality condition here:

$$cost(p_i) + cost(p_j|p_i) \leq cost(p_j) + cost(p_i|p_j)$$

The above comparison checks if the execution of path p_i followed by the execution of path p_j is cheaper than executing p_j and then p_i.

Given two paths p and p' where p' is an extension of p (*i.e.* $p' = p \cup \{e_1, e_2, \ldots, e_n\}$), we note that applying the above comparison to the pair (p, p') does not make much sense, since executing p' consists of first executing p itself. Therefore, to check which of the two paths is superior to the other, we compare them with all other non-subpath paths. Given the set S of explored paths, the to be used inequality condition is the following:

$$p \preceq p' \;\equiv cost(p) + cost(p_i|p) \leq cost(p') + cost(p_i|p') \qquad (4.12)$$

$$\forall\, p_i \in S \mid p_i \not\subset p \;\wedge\; p_i \not\subset p'$$

$$\textbf{where: } \begin{cases} p \in S \;\wedge\; p' \in S \\ p \subset p' \;\vee\; p' \subset p \end{cases}$$

The above formula translates into the following: given two paths p and p' such that one is a subpath of the other, path p is superior to p' if for every explored path p_i in S, the execution of p followed by the execution of p_i is cheaper than executing p' and then p_i.

We modify Algorithm 7 to take into account the above note. The new SuperiorPath function is given in Algorithm 8. Given a path p_i, the algorithm creates two lists of paths: the list *subpaths* which includes all paths that are subpaths of p_i or that contain p_i **(line 4)**, and the list *not_subpaths* which includes all paths that are not subpaths of p_i and that do not contain p_i **(line 5)**. In other words,

$$\begin{aligned} p_j \in subpaths \quad &\equiv \quad p_j \subset p_i \;\vee\; p_i \subset p_j \\ p_j \in not_subpaths \quad &\equiv \quad p_j \not\subset p_i \;\wedge\; p_i \not\subset p_j \end{aligned}$$

The algorithm first checks if p_i is superior to all paths in the list *not_subpaths* using the same inequality condition presented in Algorithm 7 **(line 6)**. If p_i proves to be superior, the algorithm proceeds by comparing p_i to all paths in the list *subpaths* using the newly introduced formula (4.12) **(line 7)**. The path p_i is returned for execution if it proves to be also superior to all paths in the list *subpaths*.

To prove that there exists at least one superior path in the list *subpaths*, we need to extend Lemma 4.3.5 to this special case. This proof has not been conducted yet and is left for future investigation.

Algorithm 8: SuperiorPath

```
INPUT  : Path List all_paths
// all_paths = list of all path segments that have been
   explored during chain sampling
OUTPUT: Path p

1 Path List subpaths ; // Given a path p_i, the list includes all
  paths that are subpaths of p_i or that contain p_i

2 Path List not_subpaths ; // Given a path p_i, the list includes
  all paths that are not subpaths of p_i and that do not
  contain p_i

3 FOREACH Path p_i ∈ all_paths DO
4 |  subpaths = {p_l ∈ all_paths | p_l ⊂ p_i ∨ p_i ⊂ p_l};
5 |  not_subpaths = all_paths \ subpaths;
6 |  IF cost(p_i) + cost(p_j|p_i) ≤ cost(p_j) + cost(p_i|p_j) ∀ p_j ∈ not_subpaths
  |  THEN
7 |  |  IF cost(p_i) + cost(p_k|p_i) ≤ cost(p'_i) + cost(p_k|p'_i) ∀ p_k ∈
  |  |  not_subpaths ∧ ∀ p'_i ∈ subpaths THEN
8 |  |  |  RETURN p_i ; // Path p_i is superior to all the other
  |  |  |  paths in the list all_paths, therefore p_i is
  |  |  |  returned for execution.
```

4.3.5 The Stopping Condition

Instead of exploring all the path segments in the join graph to decide upon the superior path, chain sampling uses a stopping condition that guarantees an early detection of the ***superior*** path and a safe halt of the chain sampling process. The stopping condition, initiated after each chain sampling iteration, detects the existence of an ***absolutely superior*** path which is safely returned for execution and which is guaranteed to be ***superior*** to any other path that might be explored if chain sampling is not stopped and allowed to progress.

The StoppingCondition function is given in Algorithm 9. It takes as input two lists of paths *new_paths* and *all_paths*. The list *new_paths* contains all path segments that have been created during the current chain sampling iteration, while the list *all_paths* contain all path segments that have been explored so far. The stopping condition compares in a pairwise fashion every path in *new_paths* to all the paths in *all_paths* to check if one path in

Algorithm 9: StoppingCondition

```
INPUT  : Path List new_paths, Path List all_paths
// new_paths = list of path segments that have been
   created during the current chain sampling iteration,
   all_paths = list of all path segments that have been
   explored so far (including the ones in new_paths)
OUTPUT: Path p

1 FOREACH Path p_i ∈ new_paths DO
2 |  IF cost(p_i) + cost(p_j|p_i) ≤ cost(p_j) ∀ p_j ∈ all_paths | i ≠ j THEN
3 |  |  RETURN p_i ; // Path p_i is absolutely superior to all
  |  |  the other paths in the list all_paths, therefore p_i is
  |  |  returned for execution.
  |
4 RETURN NULL;
```

new_paths is absolutely superior to all other paths and should therefore be returned for execution.

Given a path p_i in the input list *new_paths*, the StoppingCondition checks if p_i is absolutely superior to all the other explored paths by comparing p_i to every other path p_j in *all_paths* using the following inequality:

$$\underbrace{cost(p_i)}_{①} + \underbrace{cost(p_j|p_i)}_{②} \leq \underbrace{cost(p_j)}_{③} \tag{4.13}$$

$$\textbf{where:} \begin{cases} ① : \text{execution cost of } p_i \\ ② : \text{execution cost of } p_j \text{ once } p_i \text{ is executed} \\ ③ : \text{execution cost of } p_j \end{cases}$$

If there exists a path p_i which satisfies the above inequality for all other paths $p_j \in$ *all_paths*, then path p_i is absolutely superior and is returned for execution **(lines 1-3)**. If none of the paths in *new_paths* satisfy the inequality, no path is returned for execution **(line 4)**, and therefore chain sampling will initiate a new exploration iteration.

The idea behind inequality (4.13) is the following: given a path $p_i \in$ *new_paths*, if the execution of p_i followed by the execution of any other path p_j in *all_paths* is cheaper than executing p_j alone, then p_i is absolutely superior and chain sampling can be safely terminated returning p_i for execution. For example, if $cost(p_i)$, $cost(p_j)$, and $hr(p_i)$ are estimated to be

equal to respectively 500, 1200 and 0.2, then we have the following:

$$\begin{aligned} cost(p_j|p_i) &= cost(p_j) \times hr(p_i) \\ &= 1200 \times 0.2 \\ &= 240 \end{aligned}$$

$$\begin{aligned} cost(p_i) + cost(p_j|p_i) &= 500 + 240 \\ &= 740 \\ &< cost(p_j) \end{aligned}$$

It may seem like a large number of path comparisons is performed, but a path p_i which fails to be absolutely superior to any other path is directly disregarded by the algorithm, and the next path in the list *new_paths* is then checked for absolute superiority. Moreover, the comparisons can be implemented in an efficient manner (*e.g.* by sorting the paths), hence reducing the amount of time spent by the StoppingCondition function.

Definition 4.3.6. Given two paths p_i and p_j, we define the absolute superiority relation $\preceq\preceq$ as follows:

$$p_i \preceq\preceq p_j \quad \equiv \quad cost(p_i) + cost(p_j|p_i) \leq cost(p_j)$$

Therefore, given a set of paths P, we say that the path p_i is absolutely superior in P if and only $p_i \preceq\preceq p_j \ \forall \ p_j \in P \mid i \neq j$.

Lemma 4.3.7. *We claim the following:*

$$p_i \preceq\preceq p_j \quad \Longrightarrow \quad p_i \preceq p_j$$

Proof. Given two paths p_i and p_j, we have the following:

$$\begin{aligned} p_i \preceq\preceq p_j &\equiv cost(p_i) + cost(p_j|p_i) \leq cost(p_j) \\ &\Longrightarrow cost(p_i) + cost(p_j|p_i) \leq cost(p_j) + cost(p_i|p_j) \\ &\qquad\qquad \text{since } cost(p_i|p_j) \geq 0 \\ &\equiv p_i \preceq p_j \end{aligned}$$

□

We now prove that if a path p_i is found by the stopping condition to be absolutely superior to all other paths explored so far by chain sampling, then it is safe to halt chain sampling and return p_i for execution.

We first prove that if chain sampling is not stopped, and all paths p_j ($p_j \neq p_i$) are explored further, then no path superior to p_i would be found, and p_i will still be the best path to execute among all the newly explored paths.

Lemma 4.3.8. *Given a set of paths P, and a path $p_i \in P$ that is absolutely superior in P, we claim that if the chain sampling process would not be halted and would further explore every path p_j ($p_j \in P \mid p_j \neq p_i$), none of the newly explored paths will be superior to p_i.*

Proof. Given a set of paths P, and a path $p_i \in P$ that is absolutely superior in P, we have the following:

$$\begin{aligned} & p_i \preceq\preceq p_j \;\; \forall \;\; p_j \in P \mid i \neq j \\ \Longrightarrow\; & p_i \preceq p_j \;\; \forall \;\; p_j \in P \mid i \neq j && \text{by Lemma 4.3.7} \end{aligned}$$

To prove the above lemma, we suppose that the path p_i is not returned for execution and that the next chain sampling iteration is initiated, during which a path p_j from the set of paths P ($p_j \neq p_i$) is extended with a newly sampled edge e, creating a new path p' ($p' = p_j \cup \{e\}$). We consider the best possible case in which the cost of p' has the lowest possible value. The lowest cost that p' can have is when the selectivity of e in combination with the edges in p_j is the highest possible, which is the case when the hit ratio hr returned by the sampling operation of e is equal to 0. Then, we have the following:

$$\begin{aligned} hr(p') &= hr(p_j) \times hr \\ &= 0 \end{aligned}$$

$$\begin{aligned} cost(p') &= cost(p_j) + hr(p') \times card(v) \\ &= cost(p_j) \qquad (4.14) \end{aligned}$$

where v is the starting vertex of the chain sampling exploration.

We now need to prove that, even in such an optimal situation, the path p_i is still superior to the newly created path p', that is $p_i \preceq p'$.

We have the following:

$$\begin{aligned}
cost(p_i) + cost(p'|p_i) &= cost(p_i) + cost(p') \times hr(p_i) \\
&\qquad\qquad \text{by Definition 4.3.2} \\
&= cost(p_i) + cost(p_j) \times hr(p_i) \qquad \text{by (4.14)} \\
&= cost(p_i) + cost(p_j|p_i) \\
&\qquad\qquad \text{by Definition 4.3.2} \\
&\leq cost(p_j) \qquad \text{since } p_i \preceq\preceq p_j \\
&\leq cost(p') \qquad \text{by (4.14)} \\
\Longrightarrow p_i &\preceq\preceq p' \\
\Longrightarrow p_i &\preceq p' \qquad \text{by Lemma 4.3.7}
\end{aligned}$$

We have proven that in case chain sampling is not halted after an absolutely superior path p_i is found, and a path p_j ($p_j \neq p_i$) is explored further creating a new path p', then p' cannot be superior to p_i even in the best possible situation in which the newly sampled edge of p' has the highest selectivity ($hr = 0$). ☐

In the above lemma and proof, we have considered the case in which the path p_j extended during the chain sampling iteration is different than the absolutely superior path p_i. In the following, we investigate whether extending path p_i itself with a highly selective edge would make the termination of the chain sampling process unsafe.

Lemma 4.3.9. *Given a set of paths P, and a path $p_i \in P$ that is absolutely superior in P, we claim that it is safe to stop the chain sampling process even if, in the next chain sampling iteration, path p_i would have been extended with a highly selective edge.*

Proof. Given a set of paths P, and a path $p_i \in P$ that is absolutely superior in P, we suppose that chain sampling is not halted, and that p_i is extended with a newly sampled, highly selective edge e, creating a new path p' ($p' = p_i \cup \{e\}$).
Using a similar proof as in (4.14), we can derive the following:

$$\begin{aligned}
hr(p') &= 0 \\
cost(p') &= cost(p_i)
\end{aligned} \qquad (4.15)$$

We now prove that p' is in fact superior to p_i, that is $p' \preceq p_i$:

$$\begin{aligned}
cost(p') + cost(p_i|p') &= cost(p') + cost(p_i) \times hr(p') && \text{by Definition 4.3.2} \\
&= cost(p') && \text{since } hr(p') = 0 \\
&= cost(p_i) && \text{by (4.15)} \\
&\leq cost(p_i) + cost(p'|p_i) && \text{since } cost(p'|p_i) \in \mathbb{R}^+ \\
\Longrightarrow p' &\preceq p_i
\end{aligned}$$

We have proven that in case the absolutely superior path p_i is itself extended with a newly sampled, highly selective edge e ($hr = 0$), then the newly created path p' is superior to p_i. This means that the path p' should be returned for execution instead of p_i. The next question to ask is whether stopping chain sampling and returning p_i for execution without detecting the existence of the better path p', would result in a final plan P different than the final plan P' that would have been generated if p' would have been returned for execution. If it can be proven that the two plans P and P' are the same, then we can conclude that it is safe to stop chain sampling and to return p_i for execution.

We know that $p' = p_i \cup \{e\}$, which means that executing p' implies the execution of first the path p_i followed by the execution of the edge e. Therefore, the two plans P and P' are the same if after returning p_i for execution, and executing all edges in p_i, the subsequent optimization step of ROX would detect that the edge e is superior to all other paths in the join graph and would return e for execution.

Let us suppose that chain sampling is halted and that the path p_i is returned for execution. After executing all the joins in p_i, the knowledge in the join graph is updated, therefore the sample table of the vertices of every edge in p_i is updated. Using as input the new sample tables, the outgoing unexecuted edges of each vertex in p_i are re-sampled to re-estimate their weights. Among all the re-sampled edges is the edge e, the weight of which will be estimated to be 0, due to the highly selective correlation that exists between the joined vertices in p_i and the vertices of the edge e. When the subsequent optimization phase starts, the edge e is recognized to be the edge with the smallest weight, and a new chain sampling process is initiated to explore all the edges around e. We suppose that the path $p_k = \{e\}$ is created during the first chain sampling iteration. We know that $hr(e) = 0$, therefore $hr(p_k) = 0$ and $cost(p_k) = 0$. Subsequently, the stopping condition inequality (4.13) will hold when comparing p_k with any other explored path segment, proving that p_k is absolutely superior to all the other explored paths, and therefore p_k is returned for execution.

We have proven that the execution of p_i will definitely be followed by

the execution of e, which is similar to detecting the superior path p' and returning it for execution. We therefore conclude that it is safe to stop chain sampling and to return the path p_i for execution. $\square$

We have proven that if the stopping condition has found an absolutely superior path p_i, then chain sampling can be safely terminated returning p_i for execution. In fact, if a subsequent chain sampling iteration would detect a path segment that is highly selective generating an output of size 0, then p_i will still be superior to the detected path, except for the situation where the newly detected path is an extension of p_i. If the latter is the case, then it is still safe to stop chain sampling, since after the execution of p_i a new optimization phase will start using more accurate up-to-date samples with which the highly selective edge will be discovered and then returned for execution.

Note about Comparing a Path with its Subpath

Given two sets of paths *new_paths* and *all_paths* explored during chain sampling, the inequality condition used in Algorithm 9 **(line 2)** compares a path p_i in *new_paths* to every other path in *all_paths* to determine if p_i is absolutely superior such that chain sampling can be safely terminated, and p_i can be returned for execution. We restate the used inequality condition here:

$$cost(p_i) + cost(p_j|p_i) \leq cost(p_j)$$

The above comparison checks if the execution of path p_i followed by the execution of path p_j is cheaper than executing p_j alone.

Given two paths p and p' where p' is an extension of p (*i.e.* $p' = p \cup \{e_1, e_2, \ldots, e_n\}$), we note that applying the above comparison to check if p' is absolutely superior to p does not make much sense, since executing p' consists of first executing p itself. Therefore, the stopping condition should not compare a path with its subpath using the above inequality condition. In the following, we present the new StoppingCondition function and then explain the reasoning behind it.

We modify Algorithm 9 to take into account the above note. The new StoppingCondition function is given in Algorithm 10. Given a path p_i, the algorithm creates two lists of paths: the list *subpaths* which includes all paths that are subpaths of p_i or that contain p_i **(line 4)**, and the list *not_subpaths* which includes all paths that are not subpaths of p_i and that do not contain p_i **(line 5)**. In other words,

$$\begin{aligned} p_j \in subpaths &\equiv p_j \subset p_i \ \vee \ p_i \subset p_j \\ p_j \in not_subpaths &\equiv p_j \not\subset p_i \ \wedge \ p_i \not\subset p_j \end{aligned}$$

Algorithm 10: StoppingCondition

```
INPUT  : Path List new_paths, Path List all_paths
// new_paths = list of path segments that have been
   created during the current chain sampling iteration,
   all_paths = list of all path segments that have been
   explored so far (including the ones in new_paths)
OUTPUT: Path p

1 Path List subpaths ; // Given a path p_i, the list includes all
  paths in all_paths that are subpaths of p_i or that contain p_i
2 Path List not_subpaths ; // Given a path p_i, the list includes
  all paths in all_paths that are not subpaths of p_i and that
  do not contain p_i
3 FOREACH Path p_i ∈ new_paths DO
4   subpaths = {p_l ∈ all_paths | p_l ⊂ p_i ∨ p_i ⊂ p_l};
5   not_subpaths = all_paths \ subpaths;
6   IF cost(p_i) + cost(p_j|p_i) ≤ cost(p_j) ∀ p_j ∈ not_subpaths THEN
7     IF cost(p_i) + cost(p_k|p_i) ≤ cost(p'_i) + cost(p_k|p'_i) ∀ p_k ∈
      not_subpaths ∧ ∀ p'_i ∈ subpaths THEN
8       RETURN p_i ; // Path p_i is superior to all the other
        paths in the list all_paths, therefore p_i is
        returned for execution.
```

The algorithm first checks if p_i is absolutely superior to all paths in the list *not_subpaths* using the same inequality condition presented in Algorithm 9 **(line 6)**. If p_i proves to be absolutely superior, the new algorithm proceeds by comparing p_i to all paths in the list *subpaths* using inequality (4.12) described in section 4.3.4 **(line 7)**. The path p_i is returned for execution if it proves to be superior to all paths in *subpaths*.

We now explain the reasoning behind the new StoppingCondition function. In line 6 of Algorithm 9, the function checks if p_i is absolutely superior to all paths in *not_subpaths*. If it is the case, then p_i will remain the superior path even if the next chain sampling iteration extends any path $p_j \in$ *not_subpaths* with a highly selective edge generating an output of size 0. To safely stop chain sampling and return p_i for execution, we still need to make sure that p_i is also superior to all paths in *subpaths*. This is done with the comparison at line 7. We note that there is not need to

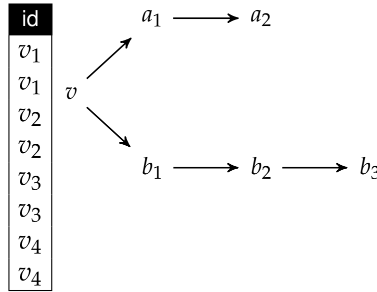


Figure 4.8 Join graph in which two path segments p_i and p_j are sampled during the chain sampling process using the vertex v as the starting point of exploration. We have $p_i = \{(v, a_1), (a_1, a_2)\}$ and $p_j = \{(v, b_1), (b_1, b_2), (b_2, b_3)\}$. The shown table corresponds to the sample table $SMPL(v)$ used as input to the sampling operations of the joins in p_i and p_j.

check the absolute superiority of p_i to all paths in *subpaths* before safely returning it for execution. It is, in fact, sufficient to prove it is superior. This is due to the fact that all paths in the list *subpaths* will not be extended during any subsequent chain sampling iteration (refer back to Algorithm 5) guaranteeing that p_i will remain the superior path in the next iterations.

4.3.6 Note about the Estimation Method of $cost(p_j|p_i)$

We consider the join graph shown in Figure 4.8, in which two paths p_i and p_j are sampled during a chain sampling process using v as the starting vertex of exploration. We have $p_i = \{(v, a_1), (a_1, a_2)\}$ and $p_j = \{(v, b_1), (b_1, b_2), (b_2, b_3)\}$. The figure shows $SMPL(v)$ the sample table of the vertex v used as input to the sampling operations of the paths p_i and p_j. If $hr(p_i) = 0.5$, then half of the tuples in the sample table of v do not match the joins in p_i and are filtered out. If half of the tuples in $SMPL(v)$ that have been filtered out by p_i match the joins in p_j, then the intermediate result generated by p_j is also reduced by half and therefore $cost(p_j|p_i) = cost(p_j) \times 0.5$. This estimation, presented in Definition 4.3.2, is accurate if the hit ratio $hr(p_i)$ is uniform among all the tuples in v. We give an example in which the above assumption does not hold.

Example 4.3.10. We reconsider the join graph of Figure 4.8. Figure 4.9 shows the sets S_1 and S_2, where the first contains the v tuples that match path p_i, while the latter consists of the v tuples that match the edge (v, b_1). We conclude from the figure that $hr(p_i) = 0.5$, and hence $cost(p_j|p_i) = cost(p_j) \times 0.5$. However, none of the v nodes in S_1 will match the edge (v, b_1), and therefore the cost of executing p_j using as input the output

S_1: tuples of v that match the path p_i
id
v_1
v_1
v_3
v_3

S_2: Tuples of v that match the edge (v, b_1)
id
v_2
v_2
v_4
v_4

Figure 4.9 Given the join graph and the chain sampled paths p_i and p_j of Figure 4.8, the set S_1 contains the v tuples that match path p_i, and the set S_2 consists of the v tuples that match the edge (v, b_1).

returned by p_i is 0 ($cost(p_j|p_i) = 0$), which is different from the derived estimation.

We can derive a better estimation of $cost(p_j|p_i)$ by intersecting the two sets S_1 and S_2 and noting the fraction f of the tuples in S_2 that are in the intersection results. The value of $cost(p_j|p_i)$ can then be estimated to be equal to $cost(p_j) \times f$.

It is also possible to get the actual value of $cost(p_j|p_i)$. One method is to re-sample the edges in p_j using S_1 as input, and summing up the size of generated intermediate results. Another more efficient method that requires no join operations is to intersect the set S_1 with each of the sets S_2, S_3, and S_4 and noting down the fraction of tuples that end up in the result. The sets S_3 and S_4 are the sets of v tuples that match respectively the two paths $\{(v, b_1), (b_1, b_2)\}$ and $\{(v, b_1), (b_1, b_2), (b_1, b_3)\}$. These intersection operations take as input two sample sets, the size of which is kept small. Subsequently, they can be executed at a cheap cost, especially if the input sample sets are already sorted. We therefore think that it is possible to use the above method to derive the actual cost $cost(p_j|p_i)$ while keeping the cost of chain sampling limited.

In the prototype of ROX, we estimate the value of $cost(p_j|p_i)$ as shown in Definition 4.3.2. To derive a better estimation, one of the above two methods could be used. Although the method we use introduces a risk of computing wrong estimations, the experiments show that ROX achieves a robust performance while making good optimization decisions.

4.4 Chain Sampling Implementation

The chain sampling process implemented in the ROX prototype differs subtly from the one described in Section 4.3. The chain sampling that has

been presented earlier in this chapter guarantees that the superior path in the explored region of the graph is detected and returned for execution. The chain sampling variant implemented in the ROX prototype does not give the same guarantee. The reason this difference exists arises from the fact that the variant explained in this chapter was detected at an advanced stage, only after the prototype has been implemented and the experiments completed. We therefore stress that the experimental results presented in the next chapter, which already prove the robustness and superiority of ROX, can be improved if the chain sampling shown in Algorithm 5 is adopted. In this section, we examine the differences between the theoretical and the implemented chain sampling process. We then show than the implemented variant might, in some situations, miss picking the best path to execute.

4.4.1 Chain Sampling in the Prototype

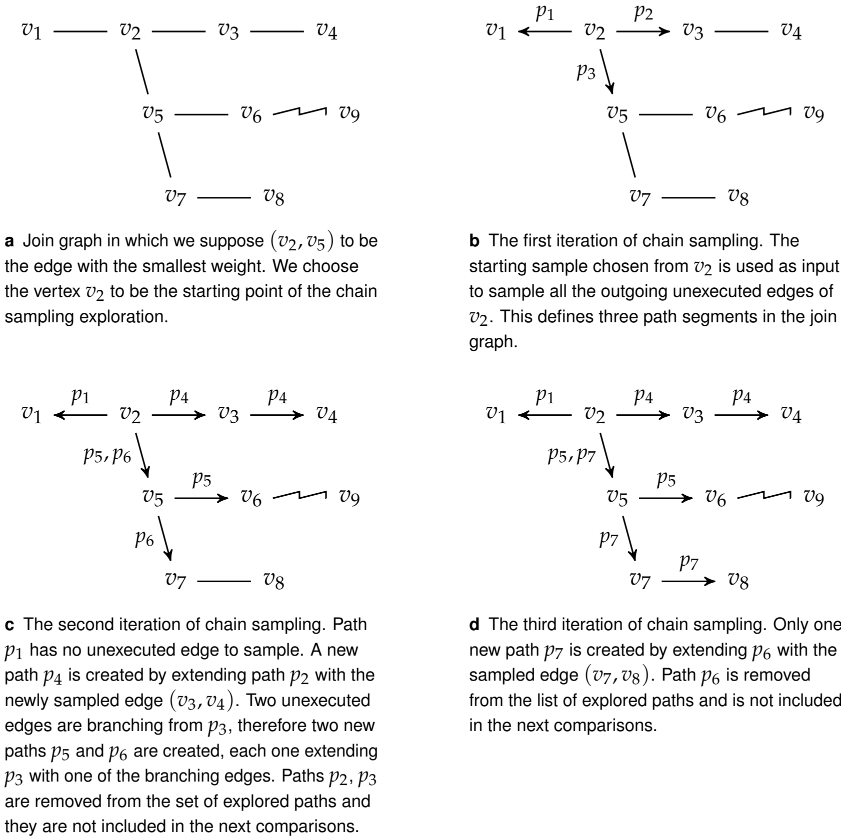
The implemented chain sampling variant differs from the theoretical one in the following point: *When a path p' is created by extending a path p with a new edge e ($p' = p \cup \{e\}$), then p is removed from the set of explored paths and is therefore not considered in the comparisons performed by the stopping condition or the* SuperiorPath *function.* In other words, a subpath p which is already included in a longer path p' is disregarded by the chain sampling process, and only p' is included in the path comparison performed by the stopping condition and possibly the SuperiorPath function.

To explain the chain sampling process implemented in the ROX prototype and to put in focus the differences with the theoretical chain sampling, we reuse the join graph of Example 4.1.3.

Example 4.4.1. Figure 4.10 illustrates the three iterations of the implemented chain sampling. The edge with the smallest weight is (v_2, v_5) and the starting point of exploration is chosen to be the vertex v_2. The edges in Figure 4.10b, Figure 4.10c, and Figure 4.10d are labeled with the path id to which they belong, and the arrows denote the direction of sampling (*i.e.* the left and right operands of the sampling operation). Figure 4.11 enumerates the edges sampled at each iteration, and illustrates the created and disregarded paths.

Iteration 1 **(Figure 4.10b):** This iteration is identical to that of the first iteration of the theoretical chain sampling, and results in the creation of three paths p_1, p_2 and p_3.

Iteration 2 **(Figure 4.10c):** The second iteration samples the next unexecuted edges branching from the start vertex of each of the three defined paths. Similar to the theoretical chain sampling process, three new paths



a Join graph in which we suppose (v_2, v_5) to be the edge with the smallest weight. We choose the vertex v_2 to be the starting point of the chain sampling exploration.

b The first iteration of chain sampling. The starting sample chosen from v_2 is used as input to sample all the outgoing unexecuted edges of v_2. This defines three path segments in the join graph.

c The second iteration of chain sampling. Path p_1 has no unexecuted edge to sample. A new path p_4 is created by extending path p_2 with the newly sampled edge (v_3, v_4). Two unexecuted edges are branching from p_3, therefore two new paths p_5 and p_6 are created, each one extending p_3 with one of the branching edges. Paths p_2, p_3 are removed from the set of explored paths and they are not included in the next comparisons.

d The third iteration of chain sampling. Only one new path p_7 is created by extending p_6 with the sampled edge (v_7, v_8). Path p_6 is removed from the list of explored paths and is not included in the next comparisons.

Figure 4.10 Illustration of the implemented chain sampling. The starting point of exploration is the edge with the smallest weight which we assume to be (v_2, v_5). The labels on the edges denote the path id(s) to which the edges belong, and the arrows indicate the sampling direction (*i.e.* the left and right operands of the sampling operation). Any explored path p included in a longer path p' is disregarded by the chain sampling process, and will not be included in the path comparison performed by the stopping condition and possibly the SuperiorPath function.

are created; however, in this case the two extended paths p_2 and p_3 are disregarded and removed from the list of explored paths.

Iteration 3 (Figure 4.10d): In the third iteration, only one new path p_7 is created by extending the path p_6 with the newly sampled edge (v_7, v_8). Path p_6 will no more be considered in the path comparisons performed by the stopping condition or the SuperiorPath function.

Iteration 1	Sampled Edges	(v_2, v_1) - (v_2, v_3) - (v_2, v_5)
	Defined Paths	$p_1 = \{(v_2, v_1)\}$ $p_2 = \{(v_2, v_3)\}$ $p_3 = \{(v_2, v_5)\}$
	Disregarded Paths	–
Iteration 2	Sampled Edges	(v_3, v_4) - (v_5, v_6) - (v_5, v_7)
	Defined Paths	$p_4 = \{(v_2, v_3), (v_3, v_4)\}$ $p_5 = \{(v_2, v_5), (v_5, v_6)\}$ $p_6 = \{(v_2, v_5), (v_5, v_7)\}$
	Disregarded Paths	p_2, p_3
Iteration 3	Sampled Edges	(v_7, v_8)
	Defined Paths	$p_7 = \{(v_2, v_5), (v_5, v_7), (v_7, v_8)\}$
	Disregarded Paths	p_6

Figure 4.11 The sampled edges and definition of paths at every iteration of the chain sampling process illustrated in Figure 4.10. At every iteration, new paths are created and some paths are disregarded.

	Theoretical		Implemented
	new_paths	*all_paths*	*all_paths*
Iteration 1	$\{p_1, p_2, p_3\}$	$\{p_1, p_2, p_3\}$	$\{p_1, p_2, p_3\}$
Iteration 2	$\{p_4, p_5, p_6\}$	$\{p_1, p_2, p_3, p_4, p_5, p_6\}$	$\{p_1, p_4, p_5, p_6\}$
Iteration 3	$\{p_7\}$	$\{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$	$\{p_1, p_4, p_5, p_7\}$

Figure 4.12 The paths that are checked for superiority by the stopping condition and possibly the SuperiorPath function in the two variants of chain sampling. In the theoretical version, newly created paths are compared against all the paths explored so far. In the implemented version, only the newly created and non-disregarded paths are checked against each other.

Figure 4.12 compares the theoretical and implemented chain sampling processes by listing the paths that are checked for superiority by the stopping condition and possibly the SuperiorPath function in the two chain sampling variants. In the theoretical chain sampling, new paths are compared against all paths, while in the implemented chain sampling newly created paths and non disregarded paths are checked against each other.

There are two differences between the algorithm of the theoretical chain sampling presented in Algorithm 5 and that of the implemented chain

sampling. We present the differences here:

1. **Difference 1:** The following two lines are added to Algorithm 5 before **line 14**:

   IF $|edges^{-}(v) \setminus p| > 0$ THEN

   $all_paths.\text{Remove}(p)$

   Every path that has at least one unexecuted to-be-explored edge branching from its start vertex v will be removed from the list *all_paths* before being extended with the newly sampled edges.

2. **Difference 2: Line 23** in Algorithm 5 is replaced by the following line:

   $p \leftarrow \text{StoppingCondition}(all_paths, all_paths)$

   In the theoretical chain sampling algorithm, the superiority of only the newly created paths is checked, by comparing them against all the paths explored so far. The reason only new paths are checked for superiority is that old paths have already been checked in previous iterations and they failed to prove superior. In the implemented chain sampling algorithm, the stopping condition compares all the paths, including the old paths, against each other. Since at every iteration some paths are removed from the list *all_paths*, then a chance exists that an old path, which was inferior to one of the disregarded paths, proves now to be absolutely superior to the paths in the list *all_paths*.

The fact that some paths are not included in the stopping condition comparison of the implemented chain sampling might result in suboptimal paths picked for execution. In the next section, we investigate the cases in which the implemented variant of chain sampling might miss picking the best plan for execution.

4.4.2 Suboptimality of the Implemented Chain Sampling

In this section, we show that the implemented chain sampling is suboptimal, and also identify the situation in which suboptimality might occur. For this purpose, we categorize all the paths explored during chain sampling in two lists:

1. *removed_paths*: all paths disregarded during the chain sampling process.

2. *all_paths*: all paths included in the checks for superiority and absolute superiority performed by respectively the SuperiorPath and StoppingCondition functions.

We need to identify the situation in which one of the paths disregarded during the chain sampling process and hence not included in the comparisons performed by the StoppingCondition and SuperiorPath functions

is better to execute than the path picked for execution by one of the two functions.

Let path $p \in removed_paths$ be superior to all paths in the list *removed_paths*: $p \preceq p_i \; \forall \; p_i \in removed_paths$. We suppose that the stopping condition or SuperiorPath function has detected that the path p' is absolutely superior or superior to all paths in *all_paths*. We therefore have: $p' \preceq p_i \; \forall \; p_i \in all_paths$.

If $p \preceq p'$ holds, then $p \preceq p_i \; \forall \; p_i \in all_paths$ is also true by transitivity. This means that p is superior to all paths in the list $removed_paths \cup all_paths$, and therefore p should be returned for execution instead of p'.

We have shown that the implemented chain sampling is suboptimal and might miss choosing the best path for execution. The suboptimality occurs when the disregarded path superior to all other disregarded paths is also superior to the path returned for execution by either the stopping condition or the SuperiorPath function. Although this puts the ROX prototype at a disadvantage when conducting the experiments, we observed that the obtained experimental results were already satisfying, proving the robustness and superiority of ROX.

4.5 The Power of the Run-time Optimizer

In this section we show the power of our proposed run-time optimizer using an example XQuery Q. We proceed as follows. We first present the decisions made by ROX when optimizing the query Q, showing the attitude of ROX towards the correlation existing between the queried data. Then, by changing the value of a predicate condition in Q, a different range of XML nodes is selected, and subsequently the impact of the correlation existing among the queried nodes on the cardinality of intermediates is modified. We then show how the optimization of the new variant of Q is addressed by ROX. The reaction of ROX to the change in the correlation illustrates the robustness of ROX and its ability to detect the existing correlations in a query, and to exploit them to generate a corresponding good execution plan.

The query Q corresponds to the following XQuery which queries the document `xmark.xml` generated from the XMark benchmark [4]:

```
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 $o
```

The query Q searches for all open auctions in the document for which the current price of the item in the bid is greater than 145, the quantity of the item in the bid is equal to 1, and information about the province of at least one of the bidders participating in the bid is available.

The new variant of Q is named Q'. The query Q' is identical to Q except for the predicate condition expressed on the `current/text()` node. The query Q' is presented below:

```
let $d := doc(“xmark.xml”)
for $o in $d//open_auction[.//current/text() < 125],
    $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 $o
```

In query Q, we select all the `open_auction` nodes with a current price > 145. In Q', we select those having a current price < 125. We realize that the higher the price of the item presented in the opened auction, the larger the number of bidders that have bid for the item, and vice versa. In Q', the selected items have a lower price than those in Q, hence the numbers of selected `bidder` nodes is smaller than those selected by the query Q. Therefore, there exists a correlation between the value of the `current` node and the number of `bidder` nodes participating in the bid. In fact this correlation spans over four different nodes: `text()` with a predicate condition, `current`, `open_auction`, and `bidder`. Therefore, our question is the following: would ROX be able to detect the correlation between the current price of an item and the number of participating bidders even though the correlation crosses over four different XML nodes? In the following, we examine the decisions made by ROX while optimizing the two queries Q and Q' to find out if ROX is capable of detecting the correlation between the four nodes, and of exploiting it to generate good execution plans.

Optimization of Q

Figure 4.13 depicts the join graph G corresponding to the query Q. Next, we illustrate some of the steps made by ROX while optimizing the input graph G.

The first phase of the ROX algorithm initializes the join graph G, *i.e.* knowledge about the vertices and edges in G is collected. First, using the available indexes, the cardinality of the XML nodes corresponding to each vertex is estimated, and a table containing a sample of these nodes is

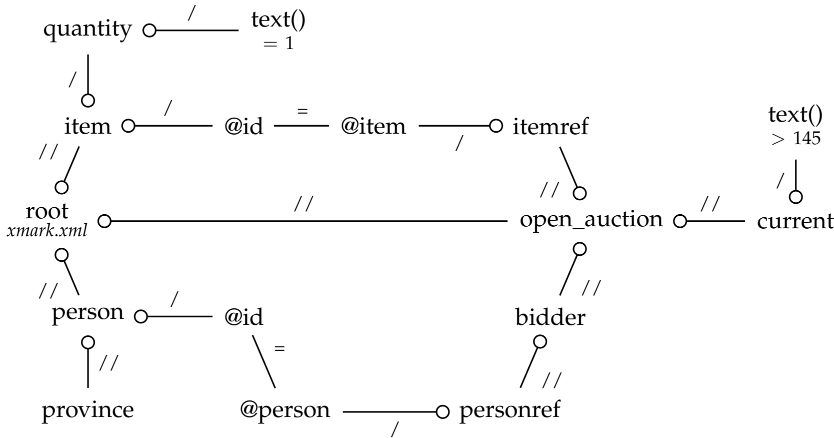


Figure 4.13 Join graph of the XQuery Q

materialized. Figure 4.14 lists the vertices and their estimated cardinality. Given the available indexes implemented in the ROX prototype, knowledge about some vertices cannot be efficiently acquired. Next the weights of the edges in the graph are computed. It is possible to compute the weight of only those edges that have at least one vertex, the sample table of which has been materialized during the previous step. Figure 4.15 shows the join graph updated with the edges' weight represented as a number label on some of the edges. Note that the edges connected to the vertex representing the root of the document are unweighted. In fact, the XPath axis of these edges is a descendant step, therefore the edges do not need to be executed and can be safely ignored during the optimization process without any risk of generating an erroneous result.

The second phase of ROX consists of alternating optimization based on chain sampling and execution steps. The following steps are iterated:

1. Pick the edge e in G with the smallest weight.
2. If at least one of the vertices of e has more than one outgoing unexecuted edge, initiate the chain sampling process with e as the starting point of exploration.
3. The path p found to be superior to the other paths is returned.
4. Execute p and update the knowledge in the join graph.

First Optimization and Execution Steps: The edge in G with the smallest weight is `(current, text() > 145)`. Since `current` has more than one unexecuted edge, chain sampling is initiated using `current` as the start

Vertex v	$card(v)$
item	21750
quantity	43500
text() = 1	42250
person	25500
province	6285
open_auction	12000
current	12000
bidder	59486
personref	59486
itemref	21750
text() > 145	–
@person	–
@item	–
@id (item)	–
@id (person)	–

Figure 4.14 The initialization phase of ROX. The estimated cardinality of XML nodes corresponding to the vertices in the join graph G. Given the indexes available in the ROX prototype, the cardinality of some of the vertices cannot be efficiently estimated.

vertex of the exploration. Figure 4.16 illustrates the chain sampling process. The edges in the graph are labeled with the paths they belong to and the arrows indicate the direction of the sampling operations. Note that the chain sampling illustrated in this example corresponds to the implemented variant described in Section 4.4 and not the theoretical one. Figure 4.16b gives a more detailed description of the chain sampling process. It lists the created paths with their cost and hit ratio. It also enumerates the paths checked for superiority by the stopping condition, and presents the outcome of the comparison. During the first two chain sampling iterations, no path satisfies the stopping condition inequality, while at the end of the third iteration, p_1 proves to be superior and is returned for execution. Figure 4.17 illustrates the resulting join graph after the first execution phase. Note that the weight of the edge (`current`, `open_auction`) is recomputed using the up-to-date newly materialized results: the weight of the edge is updated from the value 12000 to 6061. This means that it is estimated that 6061 of the 12000 `open_auction` nodes in the document have a current price greater than 145.

Second Optimization and Execution Steps: A second optimization phase is started. The edge in G with the smallest weight is (`open_auction`,

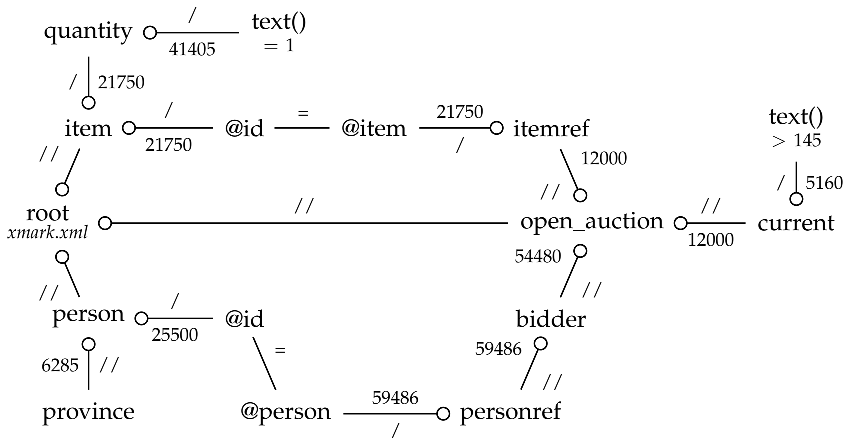
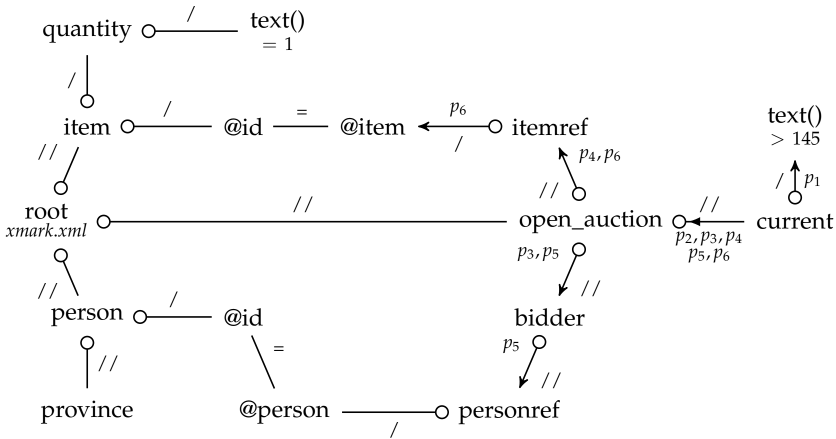


Figure 4.15 The join graph G showing the estimated weight of the edges (number labels on the edges). An edge with two vertices, the sample tables of which are not materialized, will initially stay unweighted. The edges connected to the vertex representing the root of the document are also unweighted. In fact, the XPath axis of these edges is a descendant step, therefore the edges do not need to be executed and can be safely ignored during the optimization process without any risk of generating an erroneous result.

`current`). Since `open_auction` has more than one unexecuted edge, chain sampling is initiated using `open_auction` as the start vertex of the exploration. After few iterations, the chain sampling process finds the edge (`open_auction, current`) to be superior. Therefore, the edge is executed and the weight of the two edges (`open_auction, itemref`) and (`open_auction, bidder`) is updated. The new weight of each of the two edges is respectively 6061 and 41897. We notice the following: although the number of selected `open_auction` nodes is half the total number of `open_auction` nodes in the document (6061 out of 12000), the estimated cardinality of matching `bidder` nodes is more than half the total number of `bidder` nodes in the XMark document (41897 out of 54480). This indicates that the distribution of bidders among the different opened auctions is not uniform, which is to be expected since the number of bidders participating in a bid varies from item to item. Interestingly, ROX was capable of detecting this non-uniform distribution by simply alternating sampling-based optimization and execution steps.

Third Optimization and Execution Steps: It is in this step that we will find out if ROX will detect and exploit the correlation existing in the queried data. In the third optimization phase, the edge in G with the



a The edges in the graph are labeled with the paths they belong to and the arrows indicate the direction of the sampling operations.

Iteration 1	Created paths	p_1	*cost*	6120	*hr*	0.51
		p_2		12000		1.0
	Compared paths				p_1, p_2	
	Absolute superior path				X	
Iteration 2	Created paths	p_3	*cost*	67800	*hr*	4.65
		p_4		24000		1.0
	Compared paths				p_1, p_3, p_4	
	Absolute superior path				X	
iteration 3	Created paths	p_5	*cost*	123600	*hr*	4.65
		p_6		36000		1.0
	Compared paths				p_1, p_5, p_6	
	Absolute superior path				p_1	

b The list of created paths with their cost and hit ratio properties, along with the list of paths checked for superiority by the stopping condition and the outcome of the comparison.

Figure 4.16 The first optimization phase and its chain sampling process performed by ROX.

smallest weight is (`open_auction, itemref`). Since `open_auction` has more than one outgoing unexecuted edge and has a smaller cardinality than `itemref`, then chain sampling is initiated using `open_auction` as the start vertex of the exploration. Figure 4.18 illustrates the chain sampling process. The edges in the graph are labeled with the paths they belong to and the arrows indicate the direction of the sampling operations. For

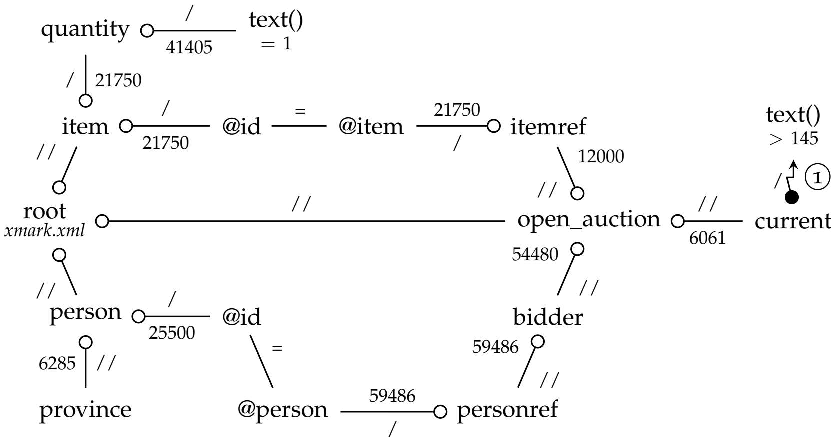


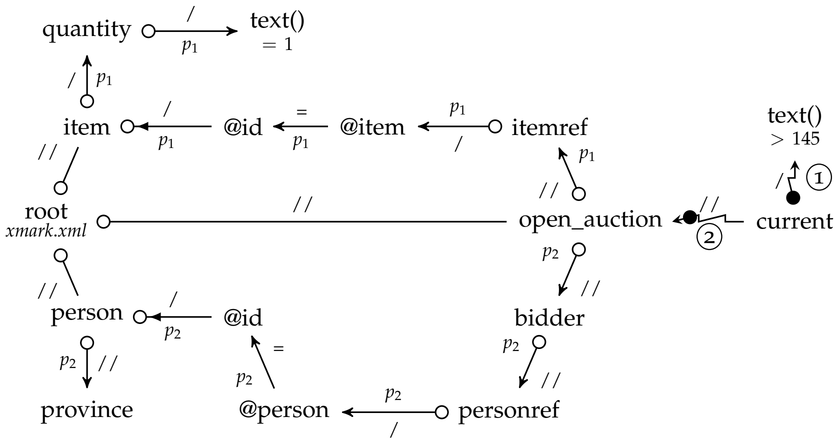
Figure 4.17 The join graph G after the first execution phase. The edge (`current`, `open_auction`) is executed and the weight of the outgoing edge of the vertex `current` is updated using the newly materialized results. The circled number indicates the execution order of the edge in the graph, and the arrow on the saw-shaped edge corresponds to the execution direction.

illustration purposes, instead of assigning a new name to every created path, we name a newly created path after the extended path it is created from. In Figure 4.18b, the explored paths and their properties along with the corresponding sampled edges are shown. The stopping condition is checked for the two paths p_1, p_2 at the end of every iteration, and fails to be satisfied. Therefore, chain sampling proceeds until all edges in the graph are sampled. Then, the SuperiorPath function is called and path p_1 is returned for execution.

After the execution of all the operators in path p_1, ROX proceeds with the alternation of optimization and execution steps until the execution order of all the edges in the join graph is determined. Figure 4.19 shows the resulting join graph after all the edges have been ordered and executed.

Optimization of Q'

The first two optimization and execution phases for the query Q' are similar to those of query Q. This results in the execution of the two edges (`current`, `text()>125`), (`current`, `open_auction`), and the re-computation of the weight of the two edges (`open_auction`, `itemref`), (`open_auction`, `bidder`) to the respective values 5079 and 14068. The weight of the other edges is similar to the weights shown in Figure 4.17.



a The chain sampling process performed during the third optimization step of ROX. Instead of assigning a new name to every created path, we, for illustration purposes, name a newly created path after the extended path it is created from.

	Path	Sampled edge	*cost*	*hr*
iteration 1	p_1	`(open_auction, itemref)`	6061	1.0
	p_2	`(open_auction, bidder)`	36729	6.06
iteration 2	p_1	`(itemref, @item)`	12122	1.0
	p_2	`(bidder, personref)`	73458	6.06
iteration 3	p_1	`(@item, @id)`	18183	1.0
	p_2	`(personref, @person)`	110187	6.06
iteration 4	p_1	`(@id, item)`	24244	1.0
	p_2	`(@person, @id)`	146916	6.06
iteration 5	p_1	`(item, quantity)`	30305	1.0
	p_2	`(@id, person)`	183645	6.06
iteration 6	p_1	`(quantity, text()=1)`	35820	0.91
	p_2	`(person,province)`	190990	1.212

b The list of paths and the corresponding sampled edges at every iteration. The cost and hit ratio properties of the paths are also shown.

Figure 4.18 The third exploration step of ROX while optimizing the query Q.

In the third optimization phase initiated by ROX, the edge with the smallest weight is `(open_auction, itemref)`. Since `open_auction` has more than one outgoing unexecuted edge and has a smaller cardinality than `itemref`, then chain sampling is also initiated using `open_auction` as the start vertex of the exploration. Figure 4.20 shows the join graph G' of Q' and the chain sampling process performed by the third optimization

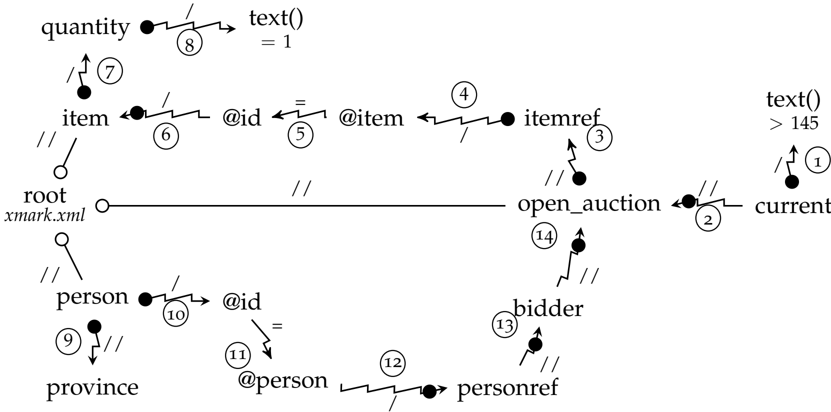
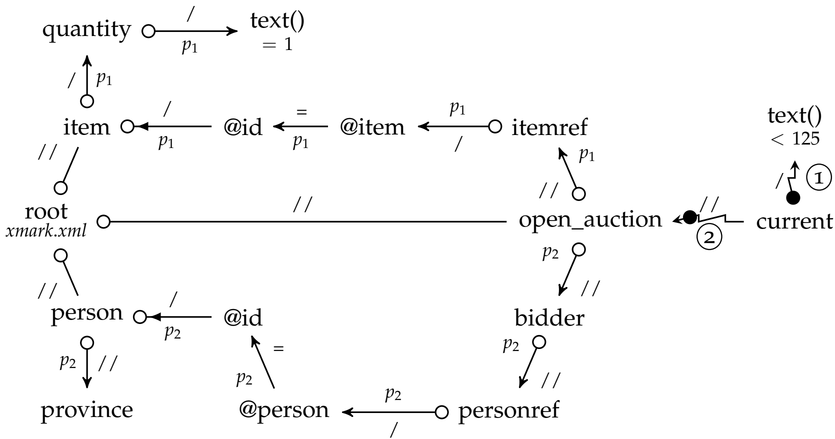


Figure 4.19 The join graph G after all the edges have been ordered and executed. The circled numbers indicate the execution order of the edges in the graph, and the arrows correspond to the execution direction.

phase. The edges in the graph are labeled with the paths they belong to and the arrows indicate the direction of the sampling operations. For illustration purposes, a newly created path is named after the extended path it is created from. In Figure 4.20b, the explored paths and their properties along with the corresponding sampled edges are shown. The stopping condition is checked for the two paths p_1, p_2 at the end of every iteration, and here also fails to be satisfied. When all edges are sampled, the SuperiorPath function is called but now the path p_2 is returned for execution. Therefore, in this case p_2 is chosen for execution instead of p_1, and the edge `(open_auction, itemref)` which was found to be the edge with the smallest weight is not executed: chain sampling climbed the hill to discover a path that is superior and which should be executed first.

We draw the attention of the reader that the number of `current` and `open_auction` nodes satisfying the predicate "> 145" is 6061, while the number of those satisfying the condition "< 125" is 5079, which is not a big difference. Although the difference in the cardinalities of `current` and `open_auction` nodes is small, the difference in the number of bidder nodes is big, going from 41897 in the "> 145" case to 14068 in the second case. This correlation that spans the four nodes `text()` with predicate condition, `current`, `open_auction`, and `bidder` is detected and exploited by ROX generating the good execution plans. This kind of correlation is hard to be detected by a traditional statistics-based optimizer. Although such an optimizer can estimate the number of `current` and possibly `open_auction` nodes satisfying the two different predicate conditions, a detection of the



a The join graph G' of Q' and the chain sampling process performed during the third optimization step of ROX. Instead of assigning a new name to every created path, we, for illustration purposes, name a newly created path after the extended path it is created from.

	Path	Sampled edge	*cost*	*hr*
iteration 1	p_1	`(open_auction, itemref)`	5079	1.0
	p_2	`(open_auction, bidder)`	11021	2.17
iteration 2	p_1	`(itemref, @item)`	10158	1.0
	p_2	`(bidder, personref)`	22042	2.17
iteration 3	p_1	`(@item, @id)`	15237	1.0
	p_2	`(personref, @person)`	33063	2.17
iteration 4	p_1	`(@id, item)`	20316	1.0
	p_2	`(@person, @id)`	44084	2.17
iteration 5	p_1	`(item, quantity)`	25395	1.0
	p_2	`(@id, person)`	55105	2.17
iteration 6	p_1	`(quantity, text()=1)`	30118	0.93
	p_2	`(person,province)`	57364	0.44

b The list of paths and the corresponding sampled edges at every iteration. The cost and hit ratio properties of the paths are also shown.

Figure 4.20 The third exploration step of ROX while optimizing the query Q'.

correlation existing between the four nodes at compile time would require a multi-dimensional histogram that summarizes the complex relation existing among the nodes. Additionally, we stress that the optimizer should decide before the query is submitted to build a histogram on these specific attributes, and identifying which attributes to build a histogram

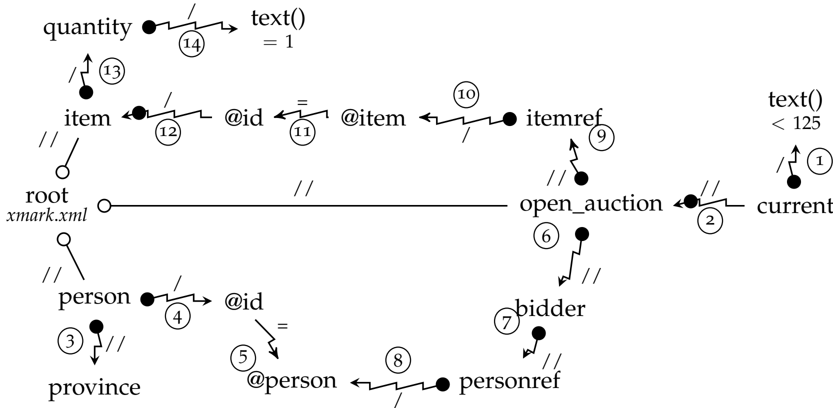


Figure 4.21 The join graph G' after all the edges have been ordered and executed. The circled numbers indicate the execution order of the edges in the graph, and the arrows correspond to the execution direction.

for without knowing the query load is a task that is far from easy. If the correlation between the four nodes is missed, then the cardinality of the XPath step join `(open_auction, bidder)` is miss-estimated resulting in an order of execution that is not optimal.

After the execution of all the operators in path p_2, ROX proceeds with the alternation of optimization and execution steps until the execution order of all the edges in the join graph is determined. Figure 4.21 shows the resulting join graph after all the edges have been ordered and executed. Note that the edges in path p_2 are not executed in the same sequential order with which they were sampled during chain sampling. In fact, ROX treats the path p_2 as a separate join graph, optimizes it, and executes its edges in the most optimal execution order found. First the selective edge `(person, province)` and its chain of operators are executed until reaching the XPath step join `(@person, personref)` which is estimated to have a large result size. Therefore, the step join is skipped, and the more selective edge `(open_auction, bidder)` and its chain of operators are executed until reaching the same step join again. The XPath step join is then executed with a smaller input set. We also note that the execution direction chosen by ROX for some of the edges in the join graph G' differs from their corresponding edges in the join graph G.

In this section, we illustrated that ROX is a robust run-time optimizer capable of using sampling techniques to learn about the queried data. By using two identical XQueries with a slight difference in one of their predicate conditions, we tested the different correlations existing in the data.

ROX reacted in a proper manner to the introduced change by detecting and exploiting the correlations to produce good execution plans.

4.6 Conclusion

This chapter has described ROX, our run-time XQuery optimizer, and has presented the ROX algorithm. We focused in this chapter on explaining ROX in the context of database systems optimized to fully materialize their intermediate results. The design of ROX on top of pipelined database systems is the subject of Chapter 6. We have proven that the paths chosen during the optimization phases of ROX are guaranteed to be superior and safe to execute. We have also presented the differences between the theoretical and implemented chain sampling techniques. Additionally, the power of ROX has been illustrated using an example XQuery. Experimental proof of the robustness of our optimizer is given by the conducted experiments and their results shown in Chapter 5.

In the following, we summarize the most important aspects of ROX:

1. **Beyond the current state-of-the-art:** 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*.

2. **Autonomy:** ROX is an *autonomous* run-time optimizer. It is not dependent on any predefined cost models nor a priori collected statistics. Hence it is independent of any assumptions made about value distributions. ROX makes informed optimization decisions by observing, with the help of sampling techniques, the characteristics of the data in base tables and intermediate results.

3. **Detection of correlations:** The adopted optimization design empowers ROX to detect correlations in the queried data. This is achieved through first the alternation of optimization phases and execution steps after which the knowledge in the join graph is updated using newly up-to-date materialized intermediates. Second, chain sampling through branches in the join graph enables ROX to discover existing correlations among the joined nodes. We note that *our chain-sampling technique provides the first generic and robust method to deal with any type of correlated data*.

4. **Quality of decisions:** During chain sampling, the choice of the path to execute is performed by the StoppingCondition or the SuperiorPath functions. Both functions *guarantee that the path returned for execution is a path superior to all the explored paths*. This is one of the reasons behind the *robustness* of ROX which will be experimentally assessed in Section 5.5.

5. **Seamless handling of XPath steps and relational joins:** ROX is the first optimizer which can seamlessly optimize the execution order and direction of XPath steps and relational joins. Moreover, by *breaking-up and stitching* complex path expressions, ROX can start the execution of an XQuery query from almost any of its XML nodes.
6. **Dynamic environments:** Unlike classical optimizers which can be optimized to handle a specific query load, ROX is not "query specific". Due to its independence of any a priori collected statistics, ROX is capable of adapting to different query loads, hence performing well in dynamic environments where the workload is continuously changing.

5 Prototype and Experiments

To test our proposed run-time optimizer, we implemented a prototype of ROX on top of the database system MonetDB/XQuery [20].[1] We chose for this specific system because of its relational database back-end, its public availability in open-source, and the fact that its Pathfinder XQuery compiler[2] can provide ROX with an isolated join graph as input [59, 60].

To put the conducted experiments in the proper context, this chapter first gives a brief description of the MonetDB/XQuery database system, and the operators it supports. Then it explains the implementation of the sampling techniques and the physical operators used in the sampling and execution of joins in the ROX prototype. Finally, the experiments conducted to prove the robustness and efficiency of the run-time optimizer are presented.

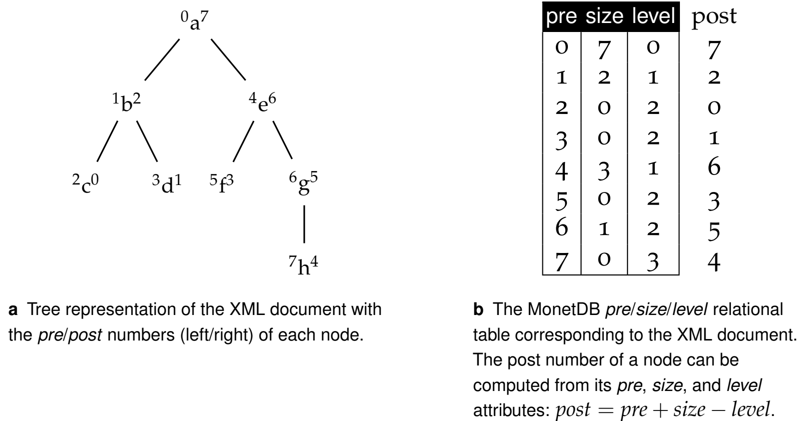
5.1 Prototype Platform: MonetDB/XQuery

A prototype of ROX is implemented on top of MonetDB/XQuery [20]. We use the "Jun2008" release of MonetDB/XQuery as platform for our prototypical implementation of ROX. We implemented our ROX approach in Java; the prototype extracts the Join Graphs that Pathfinder generates from an XQuery, and passes these to its runtime optimization and execution engine. We mention that a demo of ROX has as well been developed using MonetDB/XQuery as backend [8].

This section starts with a short introduction of the XML storage in MonetDB/XQuery, and then it describes the *Staircase Join* operator [57] used for processing XPath steps, and finally the supported indexing structures.

[1] `http://monetdb.cwi.nl/XQuery/`

[2] `http://www-db.informatik.uni-tuebingen.de/research/pathfinder`



pre	size	level	post
0	7	0	7
1	2	1	2
2	0	2	0
3	0	2	1
4	3	1	6
5	0	2	3
6	1	2	5
7	0	3	4

a Tree representation of the XML document with the *pre*/*post* numbers (left/right) of each node.

b The MonetDB *pre*/*size*/*level* relational table corresponding to the XML document. The post number of a node can be computed from its *pre*, *size*, and *level* attributes: $post = pre + size - level$.

Figure 5.1 The tree representation of an XML document and the corresponding stored *pre*/*size*/*level* relational table.

5.1.1 Storage Structure

The MonetDB/XQuery system employs an updatable document representation in which schema-free XML documents are shredded into relational tables using the range-based *pre*/*post* numbering scheme. Relational tables in MonetDB are stored in a vertically fragmented fashion in memory-mapped *arrays*. In the MonetDB *pre*/*post* range-encoding, an XML node n is represented by the triplet *pre*, *size*, and *level*. Note that the *post* number of an XML node can be computed from its *pre*, *size*, and *level* attributes. Figure 5.1 shows the tree representation of an example XML document and its corresponding relational table stored in MonetDB. The *pre* number, the node identifier of the XML node n, is a generated number reflecting the order of opening tags in a pre-order traversal of the XML tree. The attribute *size* represents the number of nodes below n in the XML tree, while *level* consists of the level of n in the tree. Information about every XML node in the document is stored in a separate relational tuple, with the node identifier *pre* as key.

The *pre* node identifier is a virtual, generated column, representing the tuple sequence number in the table. Therefore, when pages are inserted at some position in the table, the data on other pages is not affected, making structural XML updates (insert, delete) relatively cheap [20, 21, 22]. A property of this pre-based encoding scheme is that nodes can be identified with a simple integer, and compared with a fast integer comparison. Another advantage of the *pre*-identifier node representation specific to the MonetDB context is that *pre*-identifiers can be used as direct,

physical table offsets. Since in MonetDB, tables are memory-mapped and vertically fragmented, lookup operations can be done in $O(1)$.

Pathfinder: Pathfinder, the XQuery compiler built on top of MonetDB, takes as input an XQuery query, compiles it into a relational DAG, and then optimizes it. The Pathfinder optimization phase is limited to static compilation, normalization, simplification of the DAG, and the identification of the join graphs. The join graphs isolated during optimization are sent as input to ROX for further optimization. We refer the reader back to Section 3.1.3 for a description of the optimization phase in Pathfinder.

5.1.2 Staircase Join

An advantage of the adopted range-encoding is that all XPath axes can be expressed as standard purely relational operators with predicates on the *pre*, *size*, and *level* attributes [58]. It has been shown, however, both in the context of MonetDB/XQuery [20] as well as Postgres [93] that performance gain in evaluating XPath steps is achievable if a *tree aware* operator is used. As a consequence, the XQuery module of MonetDB has extended the supported relational algebra with the staircase join operator, a structural join capable of exploiting the tree properties of the *pre*/*post* plane.

The staircase join evaluates a single XPath step with linear complexity and at most a single sequential pass over the XML document representation, returning a set of tuples (nodes) duplicate-free and in document order [57]. The staircase join with axis step *axis*::*k* is defined as follows:

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

$$\textbf{where} \begin{cases} k & \in \{\texttt{*, doc, elem, text, attr, comment,} \\ & \quad \texttt{pi}\} \\ axis & \in \{\texttt{anc, ancs, child, parent, desc, self,} \\ & \quad \texttt{descs, foll, folls, prec, precs}\} \end{cases}$$

The notation $s.pre \in axis(c.pre)$ means that the tree relationship between the nodes s are c is of type *axis*. If, for instance, *axis* denotes a `child` relationship, then s is a `child` node of c. The staircase join uses as starting point a set of context nodes C, and takes as right input either the entire document ($S = D$), or a kind restriction on return node kind k ($S = D_k$), 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 returned table $[c, s]$ is sorted in document-order on s. 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*.

The staircase join uses a variety of techniques (*pruning*, *partitioning*, *skipping*) to optimize its execution of an XPath step [57]. If the right input of the staircase join is the full XML document D ($S = D$) or a kind restriction ($S = D_k$), the execution of the step performs at most one sequential scan over the document D while skipping over non-matching nodes. If the right operand is a subset S of nodes from D, the staircase join performs a special step-driven intersection that uses binary search for skipping over both the C and S sides to only process nodes that actually generate a result. Its awareness of the XPath and XQuery semantics makes the staircase join implemented in MonetDB/XQuery significantly more efficient in practice than normal structural joins [20].

Loop-Lifting

In XQuery, XPath expressions occur nested in `for`-loops, which implies that the step must be executed for *multiple* context sequences, and must produce multiple, independent, results. To avoid multiple document traversals, one for each context sequence, all staircase join algorithms in MonetDB/XQuery are *loop-lifted*, which means that query evaluation for all nested iterations can still be performed at once in a single sequential pass with skipping. The use of the loop-lifting strategy allows to optimize away some avoidable duplicate elimination and sorting operators. To still produce the correct results, loop-lifting assigns, during query evaluation, two extra attributes *iter* and *pos* to every XML node. The two attributes denote, respectively, the iteration to which an XML node belongs and its position in a given sequence of nodes. For more details on loop-lifting and the introduction and maintenance of the iteration and position attributes of XML nodes, we refer the readers to [61]. We stress that our join graph is also loop-lifted, that is every vertex in the join graph is aware of the `for`-loop to which it belongs, and the optimization and execution phases of ROX take care of introducing and maintaining the correct identifiers.

5.1.3 Index Structures

In database systems, indexes are used as a tool to efficiently retrieve tuples matching a specific predicate, hence reducing the complexity of query processing. Various XML indexing structures have been proposed, such as element indexes [26, 39, 81, 82], data guides [52, 125], and various kinds of structural and value synopses [10, 45, 46, 88, 104, 105, 106, 127].

MonetDB provides its own collection of indexes built on top of shredded XML documents. Currently, MonetDB supports an element index and a value index that covers the values of all text and attribute nodes in the document. The element index is used to retrieve XML nodes with a specific qualified name. Given a certain equality predicate condition, the value

index fetches the text or attribute nodes satisfying the predicate value. All index-lookups return a list of node identifiers (*pre*), duplicate-free and in document-order. We now formally define the index operations.

Element index: given a document D and a qualified name q, the element index returns all XML nodes in D with qualified name q:

$$\overset{elt}{\bigtriangledown}_D(q) \quad = \quad \{\texttt{pre}(e) \mid e \in D : \texttt{kind}(e) = \texttt{elem} \;\wedge\; \texttt{qname}(e) = q\}$$

Text value index: given a document D and a value v, the text value index returns all candidate text nodes in D with value v:

$$\overset{text}{\bigtriangledown}_D(v) \quad = \quad \{\texttt{pre}(t) \mid t \in D : \texttt{kind}(t) = \texttt{text} \;\wedge\; \texttt{fn:data}(t) = v\}$$

Attribute value index: given a document D, two qualified names q_{elt} and q_{attr}, and a value v, the attribute value index returns all the elements in document D with qualified name q_{elt} that are parent to candidate attribute nodes with qualified name q_{attr} and value v:

$$\begin{aligned}\overset{attr}{\bigtriangledown}_D(q_{elt}, q_{attr}, v) \quad = \quad & \{\texttt{pre}(e) \mid e \in D : \texttt{kind}(e) = \texttt{elem} \;\wedge \\ & \texttt{qname}(e) = q_{elt} \;\wedge\; e/@q_{attr} = v\}\end{aligned}$$

All indexes are stored in a materialized and physically clustered (index organized) tables. The element index is a table of the form [*qname*, *pre*] sorted on the qualified name *qname*. The value index uses a specialized hash function described in [116, 117] which maps a string value into an integer hash value such that hash collisions are kept low. The index is sorted on the key value.

For each vertex in the join graph denoting either an element type or a text node with an equality predicate, the element index and the value index can be respectively used to efficiently retrieve, as well as determine the count of all qualifying matching tuples. Since the indexes are tables ordered on the key value, an index lookup comes down to determining the start and end boundaries of the selected value. The complexity of an index lookup, and consequently the cost of finding the count of qualifying tuples, is *independent* of the result size, and is logarithmic to the index size. As a concluding remark, given a predicate p, the appropriate index can be efficiently sampled to retrieve a sample of nodes satisfying p or to determine the number of qualifying nodes.

We note that it is also possible to use the value indexes to execute equi-join operators. Given a set of number values, the value index can

be probed to evaluate an equi-join between two vertices in the join graph denoting either two text or two attribute node types.

Recently, two new updatable value indexes have been announced in MonetDB/XQuery [116, 117]. The first is hash-based and provides equality lookup on the string value of *any* XML node, including mixed content nodes. The second index allows range-lookup on any XML typed value, *e.g.* xs:double values. Adopting the new indexes in ROX is a straightforward task and will expand the optimization possibilities in ROX.

5.2 Sampling Operations

In this section, we describe the implementation of the sampling operations used by ROX.

5.2.1 Sampling from Tables and Indexes

To build a sample of the nodes corresponding to a vertex v in the join graph, ROX uses two different techniques depending on the vertex type:

1. If v is an *executed vertex* (*i.e.* at least one of its outgoing edges is executed), the full table $TBL(v)$ associated with v is sampled.
2. If v is *not an executed vertex*, the corresponding element or value index is sampled.

In the following, we describe the implementation of sampling from tables and from indexes.

Sampling from tables: To construct a sample from a given table, we chose for the simplest technique which picks in a completely random and uniform manner the tuples to include in the sample set, that is the probability of inclusion of a tuple in the sample set is uniform among all the tuples in the table. We realize that more sophisticated methods [100] (*e.g.* duplicate-free samples, weighted samples, stratified samples, . . .) can be used and will lead to more representative sample sets, and subsequently possible improvements in the current results accomplished by ROX; however, as the experiments show (Section 5.5), ROX achieves a robust performance even when such a simple sampling method is utilized.

Sampling from indexes: In ROX, available XML element and value indexes are sampled to construct the sample table associated with a given vertex in the join graph. Efficient and reliable sampling from indexes, using techniques like *partial sum trees* is well known [100]. Since an element index is a materialized table, the technique to sample from an element index is similar to that of sampling from a table (described above). Therefore, using

the element index to sample a set of XML nodes with the qualified name q boils down to sampling the sub-table consisting of the region between the start and end boundaries of the qualified name q. As the structures that store the value indexes are more complex, a simpler alternative sampling approach is adopted for convenience. To get a sample s of size $\tau = |s|$ for lookups on value v from a document D, we simply use the cutoff-sampling technique on top of the index scan operator: $\trianglerighteq_\tau(\triangledown_D(v))$. This translates to a partial execution of the index scan operator until the required number of satisfying tuples (τ) is retrieved.

5.2.2 Sampling Joins

ROX samples a given join by first picking a random set of tuples from one of the operator's input table and then joining it with the other operand. Two types of joins are found in our join graph: relational joins and XPath steps. In this section, we explain in more detail the sampling techniques used to sample both kinds of joins.

Sampling Relational Joins

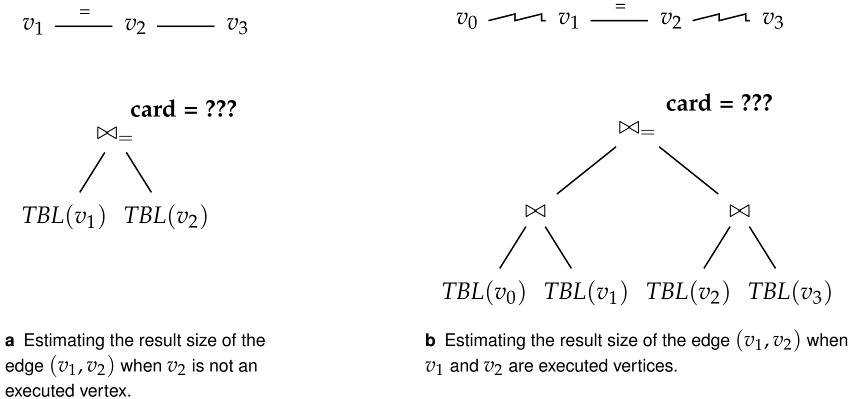
A relational join represented by the edge $e = (v_1, v_2)$ in a join graph occurs between two vertices representing either two text nodes or two attribute nodes. The semantics of the join is a value-based comparison between the tables associated with v_1 and v_2. To sample the relational join, ROX uses two different physical operators based on the type of vertices of the corresponding edge e:

1. **Case 1:** If at most one of the vertices of e is an executed vertex, then an index-based sampling technique is used.
2. **Case 2:** If both vertices of e are executed vertices, then a hash-based sampling technique is used.

In the following, we describe the above two cases in more detail.

Case 1: To sample the relational join corresponding to the edge $e = (v_1, v_2)$, a sample of the values associated with one of the vertices, say v_1, is used to probe the text or attribute value index built on the document from which v_2 is selected. The sampling operation returns a sample of the join $TBL(v_1) \bowtie TBL(v_2)$ and an estimation of the size of its full result as shown in Figure 5.2a. This index-based join sampling operation conforms to the zero-investment property introduced in Section 3.2.3 since its cost depends on its left sampled input and it requires no investment prior to starting sampling.

Case 2: The usage of an index-based sampling is not possible when the two vertices v_1 and v_2 are executed vertices, since the index built on the



a Estimating the result size of the edge (v_1, v_2) when v_2 is not an executed vertex.

b Estimating the result size of the edge (v_1, v_2) when v_1 and v_2 are executed vertices.

Figure 5.2 The sampling of an equality join

document of either v_1 or v_2 does not take into account the result of the executed edges of v_1 or v_2. Consider the join graph in Figure 5.2b in which the vertices v_1 and v_2 are executed vertices (their outgoing edges (v_0, v_1) and (v_2, v_3) have already been executed in a previous execution step of ROX). When sampling the edge (v_1, v_2), ROX wishes to estimate the cardinality of the expression $(TBL(v_0) \bowtie TBL(v_1)) \bowtie (TBL(v_2) \bowtie TBL(v_3))$ as illustrated by the plan in the same figure. Since the full table associated with a vertex in a join graph is updated to the result of the execution of its edges, ROX samples the edge e by joining the sample table $SMPL(v_1)$ with the full table $TBL(v_2)$ associated with v_2. We choose to use a hash-based join to perform the sampling operation, although we realize that it requires hashing the full table $TBL(v_2)$, and as a result does not conform to the zero-investment property. We argue, however, that a zero-investment sampling operation is simulated. In fact, if the edge (v_1, v_2) needs to be executed during a subsequent execution step, the table $TBL(v_2)$ would be hashed before performing the execution. The latter implies that instead of building the hash-table during the full execution of the edge (v_1, v_2), the hashing operation is moved to an earlier time and rescheduled to occur when the edge is sampled. If the hash table is indeed used during the execution of the edge, then building the hash table before the sampling operation is similar to using a sampling operation that is compliant to the zero-investment property.

The hashing operation will add a small cost (linear to the size of $TBL(v_2)$) to the sampling process. We note that this cost of hashing is amortized, since the hash table will be used by subsequent operations sampling another equi-join edge of v_2. Additionally, the full table of a vertex that has no unexecuted edges does not have to be hashed. Since the

relational join in our prototype are restricted to equi-joins, it is possible to restrict ourselves to hash-based sampling operation. To sample joins with inequalities, a sort-merge implementation can be used. In that case, the full table is sorted instead of being hashed.

Sampling XPath Step Joins

In the ROX prototype, an XPath step is executed using the staircase join operator. The staircase join operator is defined in Section 5.1. To sample the XPath step represented by the edge (v_1, v_2) having the label ax, a sample of nodes associated with one of the vertices, say v_1, is joined with all the nodes corresponding to the other vertex v_2. The type of staircase join used depends on whether the full table associated with v_2 is already materialized. We therefore discuss the following two situations:

- $TBL(v_2) \neq$ `NULL` : The edge (v_1, v_2) is sampled with the following staircase join operator:

  $$\overset{D_k/ax}{⫡}(SMPL(v_1), TBL(v_2)) \quad \textbf{where } k \text{ is the type of the vertex } v_2$$

  The staircase join takes as left and right input the sample table $SMPL(v_1)$ and full table $TBL(v_2)$, respectively.

- $TBL(v_2) =$ `NULL` : In this case, the sampled staircase join cannot take as right input the full table associated with v_2. Let D be the document from which the vertex v_2 is selected, then the edge (v_1, v_2) is sampled with the following staircase join operator:

  $$\overset{D_k/ax}{⫡}(SMPL(v_1), D) \quad \textbf{where } k \text{ is the type of the vertex } v_2$$

Due to its efficient implementation and its linear evaluation of an XPath step, the staircase join operator satisfies the zero-investment property as long as both its inputs are ordered.

We have already mentioned in Section 5.1.2 that the staircase join requires its left and right input to be in document-order. When sampling an XPath step, the right input R of the staircase join is sorted if R is the full document or a set of nodes directly selected with an index lookup. When R is the result of a previously executed edge e', there are two cases to consider. In the following, we suppose that R is the full table associated with the vertex v_2:

1. If the edge e' is an XPath step, and e' was executed with v_2 as its right operand, then the resulting relation R is already sorted. As we know from Section 5.1.2, the result of a staircase join is a set of tuples sorted in document-order on the attribute of its right operand.

2. In all other cases, the relation R is sorted while updating the full table of v_2 to the intermediate result generated by e' and materializing the output. As discussed in the previous section about sampling relational joins, a zero-investment sampling operation is simulated since the relation R would be sorted anyway during the execution of any of the edges of v_2. Moreover, the cost of sorting is amortized since the sorted table R will be re-used in subsequent operations sampling any of the outgoing edges of v_2.

Note about the execution of joins in the join graph: We have described the physical operators used to implement the *sampling* of the relational joins and XPath steps in the join graph. We note that the same physical operators are used to *execute* the two types of joins except that the execution uses as left input the full table (instead of the sample table) associated with the corresponding vertex of the edge.

5.2.3 Cutoff-Sampling

In this section, we describe the implementation of cutoff-sampling a join operator in the ROX prototype. We then explain a drawback in the adopted implementation, and describe one potential solution.

Given as input the join $r \bowtie T$, cutoff-sampling $\trianglerighteq_{\texttt{LIMIT}}$ matches iteratively in a sequential manner one tuple from r with the tuples in T. The matching process stops when the size of the generated result S reaches the value `LIMIT`. Therefore, only a fraction of the tuples in r are consumed by the joining process.

The adopted implementation of cutoff-sampling is biased to the front tuples in r. Let A be the set of tuples processed from the relation r. For each tuple a in A, all matching tuples in T are added to the result S. Let h_a denote the number of inner tuple hits from T on the current outer tuple a. Potentially, for some tuple $a \in A$, the number of hits might be greater than one ($h_a > 1$), that is a joins with at least two tuples in T. Therefore, and since the result generation of the joining process is restricted to an upper bound limit, a statistical bias towards the first tuples in the sample set r is introduced. The higher the value of h_a for each tuple $a \in A$, the greater the bias. This problem is illustrated in Figure 5.3. The sampled table r consists of eight tuples and only half of those are consumed ($A = \{1, 2, 3, 4\}$) by the cutoff-sampling operation to generate the required result size. Therefore, the last four tuples in r are not represented in the sampling result.
This front-biased cutoff sampling implementation has two consequences:

- The hit ratio of the join $r \bowtie T$ estimated and returned by the cutoff-sampling operation might be off the real value, resulting in less accurate result size estimations.

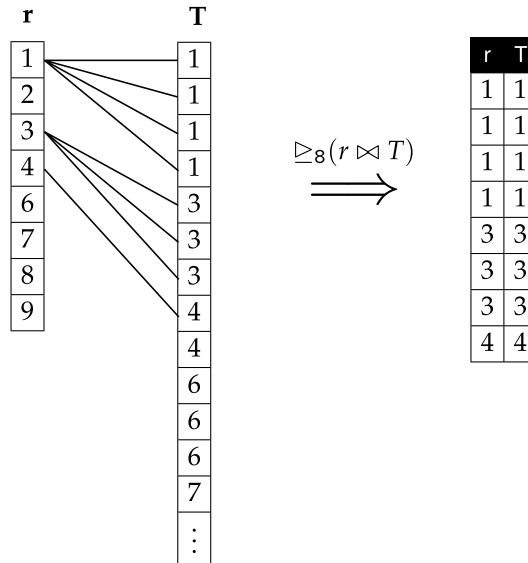


Figure 5.3 The result of cutoff-sampling the join between r and T with the operation $\trianglerighteq_8(r \bowtie T)$. The set of processed tuples from r is $A = \{1, 2, 3, 4\}$. For most tuples $a \in A$, the number of inner tuple hits h_a is a considerably large percentage of the cutoff limit 8. This creates a bias towards the front tuples in r: the remaining tuples $\{6, 7, 8, 9\}$ are not represented in the sampling output result.

- The result generated by the cutoff-sampling operation might not be representative of the full result of the join $r \bowtie T$. This can become a problem in chain sampling as the bias accumulates over subsequent sampled operators.

A potential solution to the problem, illustrated in Figure 5.4, is to adopt a cutoff-sampling technique that is not front-biased. The approach would observe the amount h_a of inner tuple hits on the current outer tuple a, and add only a number n of all the h_a tuples to the result. The other $h_a - n$ following inner tuples are skipped. The number of added tuples is a percentage of the pre-defined cutoff limit: $n = \left\lfloor \frac{\texttt{LIMIT}}{|r|} \right\rfloor$. If, after all the tuples in r are consumed, the required cutoff limit has not been reached, then the skipped inner matching tuples are added to the result. Although $h_a - n$ inner matching tuples are disregarded for a given tuple a, the value h_a would be noted by the cutoff-sampling technique to derive the exact hit ratio hr of the join $r \bowtie T$. The hit ratio hr is computed by summing up the individual h_a and dividing the sum by the size of the sampled input:

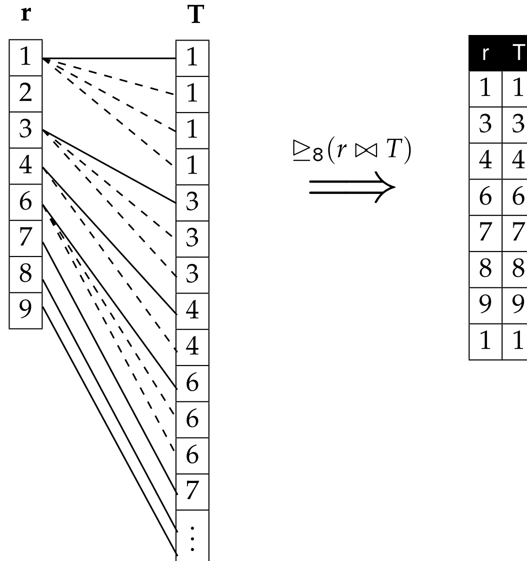


Figure 5.4 The non front-biased cutoff-sampling technique. For every tuple a in r, the number h_a of matching tuples in T is observed, and only n inner tuples are added to the sampling result. The following $h_a - n$ inner tuples are skipped. In this example, we have $n = \frac{8}{8} = 1$. The solid lines indicate the matching tuples added to the sampling result while the dashed lines correspond to the disregarded matching tuples. Since all the tuples in r are processed before reaching the cutoff limit, some of the skipped inner matching tuples are subsequently added to the result. In this example, only a single previously disregarded tuple is added to the sampling output.

$hr = \frac{1}{|r|} \sum_{a \in r} h_a.$

This implementation of cutoff-sampling guarantees that the sampled join result stays in line with the sampled input without front bias, rendering the observations made during the chain sampling process more reliable. It also ensures a more accurate estimation of the result size of the join since all the tuples in r are processed and the number of matches h_a is noted down to compute the exact value of the hit ratio of $r \bowtie T$.

In the implemented ROX prototype, we adopt the front-biased cutoff-sampling technique. Since our focus is on run-time query optimization rather than on new sampling methods, we refrained from extending the underlying database system MonetDB/XQuery with such new physical cutoff-sampling-aware XPath step and relational join operators, and accepted the front-bias risk. Fortunately, the conducted experiments have shown that ROX performs well, even in the presence of such a risk.

Wrap-up

We have, in the previous sections, described MonetDB/XQuery, the database backend on top of which the ROX prototype is built. We have also explained the staircase join physical operator used to execute the XPath steps in the join graph, and the type of indexes supported by MonetDB. The implementation of the sampling and execution operations of relational joins and XPath steps as well as the cutoff-sampling technique were also presented. We pointed out that the cutoff-sampling technique implemented in MonetDB is front-biased, and we have described a potential solution proposing another cutoff-sampling approach which is not front-biased. We again emphasize that the operators used in the sampling of joins comply to our introduced zero-investment property. In some cases, however, the sampling operators require the hashing or sorting of the input tables before initiating the sampling operation. For these operators, we have stressed that they still simulate the use of zero-investment operators since the hashing or sorting operations would have been performed anyway in a subsequent execution step, and the cost of these two operations is amortized as their result might be used in later sampling operations.

In the following, we describe the conducted experiments and report the observed results.

5.3 Overview of Experiments

In the conducted experiments, our run-time optimizer ROX is tested against different data-sets and queries. We use documents containing on the one hand synthetic data and on the other hand real-life data (DBLP data set). The following points are the subject of the conducted experiments:

1. The main objective of the experiments is to assess the robustness of ROX in always picking (near-)optimal plans and invariably avoiding the bad ones.
2. We take a close look whether ROX is successfully defining a good execution order and direction for both the relational joins and XPath steps in the join graph.
3. Another tested point is the ability of ROX to detect and adapt to different correlations in the queried data, and to exploit them to produce good plans.
4. The efficiency of ROX is also investigated by measuring the amount of sampling overhead incurred during the optimization steps.
5. Finally, we examine the impact of different sample sizes on the sampling overhead, and the sensitivity of the robustness of ROX to the used sample sizes.

5.4 XMark Experiment

The objective of our first set of experiments is to make a quick comparison between the performance of MonetDB-with-ROX and MonetDB-without-ROX, and to show the ability of ROX in detecting the correlations and exploiting them to generate good execution plans.

We reconsider the two queries on the XMark document presented in Section 4.5. We once again present the query Q and its variant Q':

Query Q

```
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 $o
```

Query Q′

```
let $d := doc(‘‘xmark.xml’’)
for $o in $d//open_auction[.//current/text() < 125],
    $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 $o
```

We execute the two queries in MonetDB/XQuery with and without ROX using an XMark document of size 112 MB, and we measure the elapsed running time of the generated plans. Figure 5.5 reports the execution times of the plan generated by ROX excluding and including the sampling overhead, and the plan generated by the Pathfinder compiler for the two queries Q and Q'. The shown times are normalized relative to the fastest of the 3 plans.

We notice that the plan generated by Pathfinder is about 4 orders of magnitude slower than the ROX plan. However, we emphasize that the optimizer built in Pathfinder is not equipped with a module which re-orders select and join operators in the compiled plan based on their selectivities. In fact, Pathfinder produces the same plan for both queries Q and Q', and in those compiled plans, the selections are not pushed down, (*e.g.* the selection on the `current` value is executed last). We also note

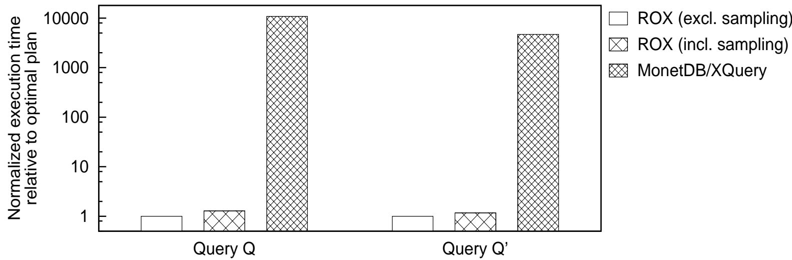


Figure 5.5 Execution times of the plans generated by ROX including and excluding the sampling overhead, and the compile-time optimized plan generated by MonetDB/XQuery without ROX running on top. The plotted times are normalized to the fastest of the 3 considered plans.

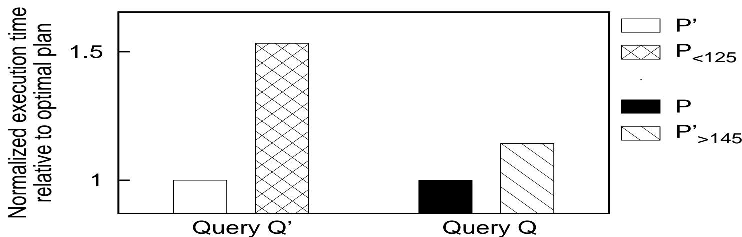


Figure 5.6 Execution times of the plans P' and $P_{<125}$ generated for the query Q' by respectively ROX and the considered compile-time optimizer. Similarly, the plans P and $P'_{<145}$ produced by the two optimizers for the query Q are shown. The plotted times are normalized to the fastest of the 2 generated plans for each query.

that, for both queries, the overhead introduced by the sampling in ROX is around 20% of the execution time.

To have a fairer comparison, we consider, in our second XMark experiment, a compile-time optimizer with optimization capabilities better than those in the Pathfinder optimizer. We also stress that the considered optimizer is capable of generating better plans than a typical compile-time optimizer. We refer to the plans generated by ROX for Q and Q' as, respectively, P and P'. The considered optimizer successfully optimizes the query Q producing the same plan P generated by ROX, but unable to detect the difference in correlation between the two queries Q and Q', it fails to generate the plan P' for Q'. Instead it generates a plan $P_{<125}$, which shares the same operators and ordering as P, but in which the predicate condition on the `current/text()` node is replaced with < 125.

Figure 5.6 illustrates the elapsed time of the two plans P' and $P_{<125}$ generated for the query Q' by respectively ROX and the considered compile-time optimizer. The plotted times are normalized to the fastest of the two plans. The execution time of $P_{<125}$ is 54% slower than the ROX plan P'. We note that the main difference concerning the ordering of operators between the two plans P' (refer back to join graph G' in Figure 4.21) and $P_{<125}$ (refer back to join graph G in Figure 4.19) is the execution order of the two longer sequences of operators branching from the vertex `open_auction`, *i.e.* which of the branches is executed first (the order of the operators inside each branch is almost similar in the two plans). This explains the fact that the gap between the execution time of the two plans is not in the order of magnitude (the plan $P_{<125}$ is already close to optimal as all the selective selections are already pushed down). *ROX; however, succeeded in identifying that reversing the execution order of the two branches results in a better plan*. We, therefore, conclude that ROX is highly sensitive to the differences in the correlation between the two queries Q and Q', and has the ability to exploit it to find better orderings of the operators in the plan.

We have repeated the same experiment for the query Q, and we came to the same aforementioned conclusions. In this experiment, the considered optimizer successfully optimizes Q' generating the same plan P' produced by ROX, but fails to generate the plan P when optimizing Q. Instead, it produces the plan $P'_{>145}$ which shares the same operator ordering as P, but in which the predicate condition on the `current/text()` node is replaced with > 145. Figure 5.6 plots the execution time of the two plans P and $P'_{>145}$, normalized to the fastest of the two. The execution time of $P'_{>145}$ is 12% slower than the ROX plan P.

5.5 DBLP Experiment

The XMark experiment we just described illustrates the principles and potentials of ROX concerning data correlations. However, it represents only one specific case with limited potential for variation. To prove the robustness of ROX, its stability and ability to detect and exploit the correlations existing in the queried XML documents, we need to assess the behavior and decisions made by our run-time optimizer on a set of queries that exhibits a large variety of different constellations. To achieve this, we use the DBLP XML dataset[3] which contains real-world data with different types of correlation among its nodes.

In addition to testing the robustness of ROX, the experiments also investigate the importance of correctly ordering the XPath steps among equi-joins, the amount of sampling overhead incurred in ROX, and the

[3] `http://dblp.uni-trier.de/xml/`

impact of the sample size on the sampling overhead and the quality of plans generated by ROX.

For all experiments presented in this section, 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. We note that all the times shown in our plots are normalized.

5.5.1 Dataset and Query Template

The DBLP document, composed of a sequence of entries for articles, inproceedings, books, *etc.*, is divided 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 who have published in all four 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 corresponding to our query template is depicted in Figure 5.7. The solid edges arise from the original join graph as extracted by the Pathfinder compiler from the XQuery template. The dotted lines denote join equivalences implied from the three equality joins defined in the query template, and are added by ROX to extend the search space of plans allowing for more flexibility to find a (near-)optimal plan. We note that this optimization technique of equality equivalences is not specific to ROX, and might very well be in use in other relational database systems.

The ∼4500 XML documents are categorized into different classes, each class representing a different research area (*i.e.* Databases, Data Mining, Information Retrieval). The idea of the experimental setup is to replace the four documents in the query template with documents chosen from one or more research areas. This results in a variation in the degree of correlation in the query: it is in general more likely that authors publish in various journals and/or conferences belonging to the same research area, rather than that an author publishes in different research areas. Suppose that in our template join graph, `DOC1.xml` is assigned a database conference while

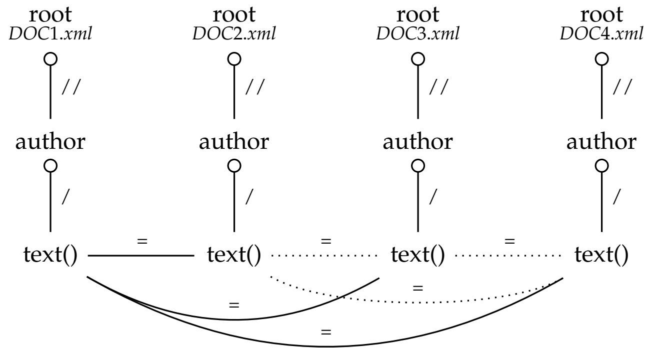


Figure 5.7 Join graph corresponding to the DBLP query template. The solid lines correspond to the edges in the join graph generated by the Pathfinder XQuery compiler. The dotted lines are added by ROX and represent join equivalences implied from the three equality joins defined in the query template.

the other 3 documents are replaced by 3 different information retrieval conferences. Therefore, it can be predicted that the join between the authors of the first document and any of the other 3 documents has a considerably lower selectivity than a join between any two information retrieval conferences. ROX will need to accurately estimate the result size of the equi-joins to detect the correlation among the queried documents.

The ∼4500 documents yield a total number of 409515972723000 of 4-documents combinations. We believe that testing this big number of combinations will take unnecessarily a large amount of time without much impact on the experimental results, we, therefore, restrict ourselves to 23 "representative" documents chosen from 5 research areas. The selected documents and their research area(s) are listed in the first 2 columns of Figure 5.8. The original DBLP dataset consists of a ∼450 MB XML document, and the size of each of the ∼4500 journals and conference series ranges from 300 B to 4.8 MB in size. Even when choosing the four largest journals/conferences, ROX + MonetDB/XQuery manages to evaluate the defined query in less than 50 milliseconds. To achieve more reliable performance measurements, we scale the selected 23 documents $10\times$ and $100\times$ 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 $[1, \ldots, n]$. The total size of the 23 documents included in our experiments is 15 MB (× 1) and 1.5 GB (× 100). Figure 5.8 lists the size of each of the 23 documents and the number of `author` nodes contained in each document, before (× 1) and after (× 100) the scaling up.

	journal/conference name	research area(s)(1)		document size × 1(2)	× 100(3)	# author tags × 1(2)	× 100(3)
1-	Fuzzy Logic in Artificial Intelligence	AI		12 KB	1.2 MB	62	6200
2-	Artificial Intelligence in Medicine	AI		332 KB	33 MB	2264	226400
3-	AAAI	AI		1.1 MB	105 MB	6832	683200
4-	CANS	AI	BI	32 KB	3.1 MB	214	21400
5-	BMC Bioinformatics		BI	440 KB	44 MB	3547	354700
6-	Bioinformatics		BI	2.1 MB	205 MB	15019	1501900
7-	BIOKDD	DM	BI	22 KB	2.1 MB	139	13900
8-	MLDM	DM		99 KB	9.9 MB	575	57500
9-	ICDM	DM		348 KB	35 MB	2205	220500
10-	KDD	DM		460 KB	46 MB	3201	320100
11-	WSDM	DM	IR	13 KB	1.2 MB	95	9500
12-	INEX		IR	54 KB	5.4 MB	342	34200
13-	SPIRE		IR	124 KB	13 MB	724	72400
14-	TREC		IR	304 KB	31 MB	2541	254100
15-	SIGIR		IR	811 KB	81 MB	4584	458400
16-	ICME		IR	828 KB	83 MB	5757	575700
17-	ICIP		IR	1.2 MB	113 MB	7935	793500
18-	CIKM	DB	IR	629 KB	63 MB	3684	368400
19-	ADBIS	DB		294 KB	29 MB	947	94700
20-	EDBT	DB		389 KB	39 MB	1340	134000
21-	SIGMOD	DB		1.8 MB	173 MB	5912	591200
21-	ICDE	DB		1.7 MB	163 MB	6169	616900
23-	VLDB	DB		2.1 MB	204 MB	6865	686500

(1) AI = artificial intelligence, BI = bioinformatics, DB = database, DM = data mining, IR = information retrieval

(2) Before scaling up the DBLP XML document

(3) After scaling up the DBLP XML document 100 times

Figure 5.8 The 23 chosen documents, the research areas they belong to, their size, and the number of `author` nodes contained in each.

Using the 23 selected documents, we create a large variety of 831 4-document combinations which we classify into the following 3 groups:

1. **Group 2:2** This group contains all combinations of 4 documents such that two documents are chosen from the same research area and the other two are picked from another research area. For instance, belonging to this group is the document combination: ICDE, VLDB,

SIGIR, and TREC. The first two documents correspond to database conferences while the last two are information retrieval conferences.

2. **Group 3:1** This group contains all combinations of 4 documents such that 3 documents are chosen from the same research area and one from a different area.
3. **Group 4:0** This group contains all combinations of 4 documents such that all 4 documents are picked from the same research area.

The idea is that these groups roughly cluster the 4-document combinations according to the type of correlation existing among the 4 documents. Omitting document combinations that yield empty results with our template query, group 2:2 contains 469 combinations, group 3:1 contains 337 combinations, and group 4:0 contains 25 combinations.[4]

By replacing the 4 document vertices in the join graph with the 831 created document combinations, we test the robustness of ROX and its ability in handling, detecting and exploiting different degrees of correlations. We note that during the run-time optimization, a single chain sampling process explores up to 15 different path segments in the join graph with a length ranging between 2 and 4 edges.

5.5.2 Tested Query Execution Plans

For each of the 831 document combinations, a number of query execution plans corresponding to the query template are created, executed, and their execution time is compared. The first and second considered execution plans are the ones generated by ROX including and excluding the sampling overhead respectively. These two execution plans are referred to as "ROX full run" and "ROX pure plan".

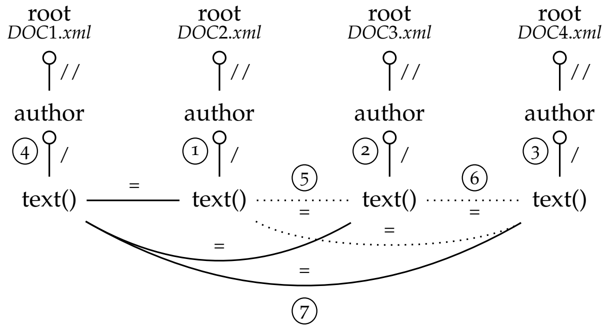
To assess the quality of the query plans generated by ROX, we have implemented a tool that enumerates all possible execution plans in the search space of our query template. The enumeration tool varies the order of equi-joins, the placement of location steps among the equi-joins, the direction of path steps, and the use of indexes. In this way, it enumerates a total of 88880 different physical plans for our 4-way join DBLP query. Obviously, we cannot compare the 2 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 our 4-way join query, there are 18 different ways to order the equi-join, and consequently 18 categories. Execution plans with the same equi-join order belong to the same category. The second categorization level

[4] We note that the 25 document combinations in the group 4:0 consist of 5 combinations created using database conferences and 20 combinations created with information retrieval conferences. Any 4 document combination chosen from the other 3 research areas generates an empty result set and is therefore excluded from the experiment.

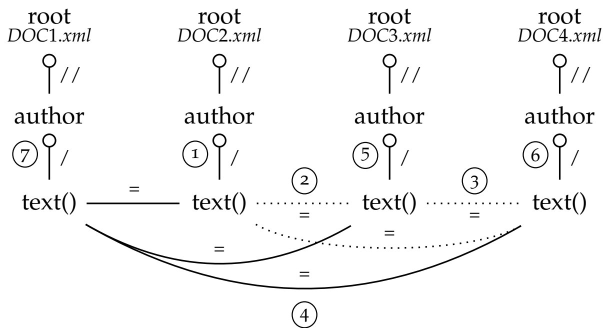
is the placement of steps among the equi-joins. In total, there are 804 different ways to place the `author/text()` steps among the equi-joins. For our experiments, we limit our considerations to 3 canonical plans each exhibiting a specific step placement. The first denoted as "SJ" consists of executing the steps of all 4 documents before the joins in the same order of the joins execution ($SJ=S_aS_bS_cS_dJ_aJ_bJ_cJ_d$). The second, referred to as "JS", corresponds to first executing one step to provide the initial input for the join sequence, then all joins are evaluated, and the remaining 3 steps are executed last ($JS=S_aJ_aJ_bJ_cJ_dS_bS_cS_d$). The last canonical step placement is denoted S_J, and consists of first executing the initial step and join, then a step corresponding to a certain document is executed right after the document has been joined to the already generated intermediate result ($S_J=S_aJ_aJ_bS_bJ_cS_cJ_dS_d$). The three canonical steps are illustrated in Figure 5.9 in which we consider the category of equi-join ordering where the join between the second and third document is executed first, then the fourth and first document are consecutively joined with the intermediate result. The first canonical step placement SJ is depicted in Figure 5.9a. The circled numbers represent the order of execution of the edges in the graph. The second and third canonical step placements are illustrated in, respectively, Figure 5.9b and Figure 5.9c. Note that the edges from the root vertices are not executed since they correspond to XPath steps with a descendant axis and can therefore be skipped without generating erroneous results. Also note that the redundant equi-join edges are not executed.

From the 88880 categorized plans, we pick two execution plans, named respectively *smallest* and *largest*. The *smallest* execution plan belongs to the equi-join ordering category that yields the smallest cumulative intermediate result sizes (*i.e.* the sum of all generated intermediate result sizes and the final result size is the smallest). The step placement in the *smallest* plan corresponds to the canonical step placement that has the smallest execution time. The equi-joins in the *largest* plan are ordered such that the largest cumulative intermediate result is generated, and the position of the step operators matches the canonical step placement with the slowest execution time. These two execution plans can be viewed as an approximation of the plans at the lower and upper bounds of the search space of the query template.

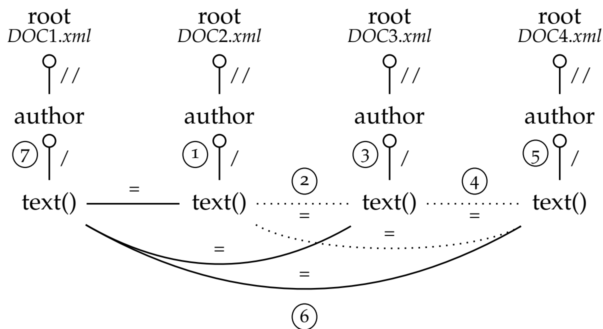
In addition to the *smallest* and *largest* plans chosen from the enumerated search space, we assume a "*classical*" best effort compile time optimizer equipped with an accurate cardinality estimation module. This translates, in our DBLP example, to the optimizer correctly estimating the result size of a join operator executed in the context of a single document, but lacking the ability to estimate the cardinality of operations joining two different documents. Consequently, the optimizer falls back on a simple "smallest-input-first" heuristic to determine an appropriate equi-join order. This



a The SJ canonical step placement. All steps are executed before the equi-joins. The order of execution of the steps corresponds to the order in which the documents are joined together.



b The JS canonical step placement. One XPath step is executed to provide the initial input to the first join operation. Then all joins are executed before executing the remaining three steps. The order of the execution of the last three steps corresponds to the order in which the documents are equi-joined.



c The S_J canonical step placement. One XPath step is executed to provide the initial input to the first join operation. Then the step of every document is executed directly after the document is equi-joined to the already generated intermediate result.

Figure 5.9 The three canonical step placements for the equi-join category in which the second and third documents are first joined, and then the fourth and first are consecutively joined with the intermediate result.

results in a linear join order such that the two smallest sets of *author/text()* values are joined first, which is then joined with the second largest set, and finally joined with the largest one. We note that this join ordering technique with which the classical optimizer is equipped results in execution plans that are faster than those generated by the compile-time optimizer of the MonetDB/XQuery system.

Viewing the assumptions we made for the considered classical compile-time optimizer as fair and reasonable, we believe that our considered optimizer is comparable to a typical relational optimizer. In fact, enabling the relational optimizer to accurately estimate the result size of an equi-join between the different documents requires the construction of a synopsis that summarizes the *total number* of authors shared between the four documents. Although building such a structure for only 2 documents is not hard (*e.g.* a two dimensional table with the different documents listed on both the x and y axes, and in which the entered value is the number of authors common to every two documents), to also accurately estimate the result size of the 3 and 4 equi-joins will require a far more complex synopsis. We believe that a typical relational optimizer would not maintain the aforementioned synopsis for the following reasons:

1. With the numerous possible document combinations, the synopsis consumes a large amount of space and requires a high maintenance cost.
2. The synopsis is highly query specific: any modification to the DBLP query by adding either a selection on the author name or another attribute from the document, will render the synopsis useless. Therefore, building such a synopsis does not pay off in general. Having a more generic synopsis requires a more detailed view of the distribution of the authors between the documents, and a representation of the correlation between the author nodes and the other attributes in the document. This type of generic synopsis is even more complex, space consuming, and requires a higher maintenance cost.

5.5.3 Influence of Equi-Join Order on Intermediate Result Size

Our first experiment is a demonstration of how the order of equi-joins influences the cumulative intermediate result sizes (and hence execution costs). We consider the DBLP query template in which the queried documents are set to the following 4 selected conferences: VLDB, ICDE, ICIP, ADBIS. ICIP is an *information retrieval* conference while the other 3 are *database* conferences. We calculate the sum of the number of tuples generated by all equi-joins for every possible equi-join ordering. We stress that the sum takes into account only those tuples generated by the equi-joins. As for the

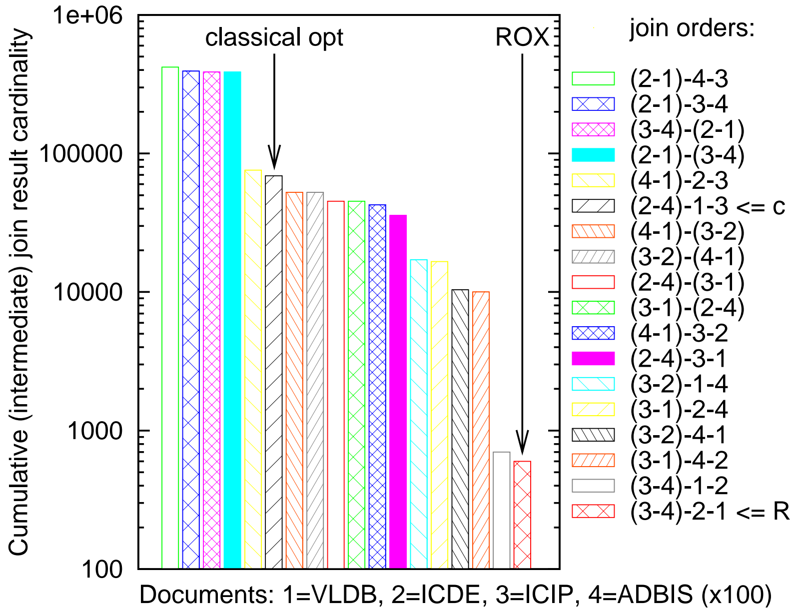


Figure 5.10 Impact of equi-join ordering on intermediate result sizes.

XPath steps, they are executed before the equi-joins providing the required input to the equality joins.

Figure 5.10 shows the results for the $\times$ 100 scaled dataset. The 18 different equi-join orderings as listed in the legend of the figure. The join ordering chosen by ROX and the classical optimizer are indicated, respectively, by the "R" and "c" arrows. The numbers represent the queried documents, that is the join ordering "(3-4)-1-2" represents the plan in which the third (ICIP) and fourth (ADBIS) documents are joined, then the intermediate result is joined with the first (VLDB) and finally second (ICDE) documents. Individual joins are (logically/semantically) symmetric, but we distinguish linear and bushy plans. Parentheses in the join order notation indicate precedence, and hence bushiness. Therefore, a join ordering containing two pairs of parentheses represents a bushy plan, while one with a single pair forms a linear plan.

Due to the correlation among the 3 database conferences, all join orders that consider the information retrieval conference ICIP only at the end, need to process considerably (between 2 and 3 orders of magnitude) larger intermediate data volumes than those join orders that start with ICIP. Moreover, the linear join orders which consider first ICIP and ADBIS (*i.e.* (3-4)) generate less intermediate data than those which first join ICIP with

one of the other two database conferences (*i.e.* (3-1) and (3-2)). This is logical since ADBIS is a smaller database conference, and hence it is likely it has less common authors with an information retrieval conference.

ROX manages to detect and select the equi-join order that creates the smallest size of intermediates, while the classical optimizer is not able to recognize and exploit the correlation. ROX chooses a join order which considers ICIP and ADBIS first, and the classical optimizer joins first ICDE and ADBIS, keeping the join with ICIP till the end. We also note that 16 out of the 18 equi-join orderings generate an intermediate result that is more than an order of magnitude larger than the optimal one. We therefore conclude that *ROX was well able to selectively pick the plan with the best join ordering even though the bigger portion of the search space contains plans that are far from optimal*.

5.5.4 Robustness of ROX

In this experiment, we assess the robustness of ROX in always picking a good plan while invariably avoiding the bad ones.

We measure the elapsed execution time of the different defined plans for the 831 document combinations using the $\times$ 100 scaled dataset. For every combination of 4 documents, we compare the following five execution plans: (1) ROX full run, (2) ROX pure plan (3) *smallest* equi-join order plan + fastest performance step placement, (4) *largest* equi-join order plan + slowest performance step placement, (5) *classical* compile-time optimizer plan.

We also compute a "correlation" measure for each combination of 4 documents. The measure represents the standard deviation of the selectivity of the join between every pair of documents. For the document combination $D = \{d_1, d_2, d_3, d_4\}$, we note by A_i the set of author nodes in the document d_i ($i \in [1,4]$). The correlation C of the document combination D is computed as follows:

$$
\begin{aligned}
js(d_i, d_j) &= \frac{|A_i \bowtie A_j|}{\max\{|A_i|, |A_j|\}} \\
mean &= \frac{1}{3!} \sum_{i=1}^{4} \left(\sum_{j=i+1}^{4} js(d_i, d_j) \right) \\
diff(d_i, d_j) &= (js(d_i, d_j) - mean)^2 \\
C &= \frac{1}{3!} \sum_{i=1}^{4} \left(\sum_{j=i+1}^{4} diff(d_i, d_j) \right) \\
\textbf{where} \quad & d_i, d_j \in D
\end{aligned}
$$

The computed correlation measures the degree of dispersion of the selectivity of the equi-joins in our DBLP query template. The higher the correlation measure, the larger the selectivity dispersion, which means that the choices of equi-join ordering made by the optimizer have a bigger impact on the quality of the final execution plan. In other words, the quality of the final plan is more sensitive to the made choices. The computed correlation measures are used to quantify and illustrate the different correlations tested during the experiment.

Figure 5.11 presents the elapsed execution time of the 5 different considered plans for each of the 831 document combinations. The plotted times are normalized to the fastest of the 5 considered plans. 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 the *smallest* equi-join order and *classical* plans, the normalized execution time of the fastest of the SJ, JS, and S_J plans is plotted. For the *largest* equi-join order plan, the time of the slowest of the three canonical steps placement is drawn. We remind the reader that the *smallest* and *largest* equi-join order plans are used as an approximation of the plans at, respectively, the lower and upper bounds of the search space of the query.

The reader of this plot should not try to examine specific cases but should observe the trend created by 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 overhead with the pure plan, we notice that on *average the overhead imposed by sampling is around 27%, and almost always lower than a factor two*.

We also observe that ROX behaves roughly the same across the various types of queries (2:2, 3:1 and 4:0) showing that it is highly adaptive to the correlation and is robustly exploiting the existence of a selective correlation between the joined documents. *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 measures in group 3:1*. On average, the classical plan exceeds the ROX results by a factor 3.4 in group 2:2, a factor 6 in group 3:1, and even a 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. In fact, although the 4 documents belong to the same research area, the high correlation measure is due to the fact that authors usually favor publishing in specific conferences in their area of interest resulting in less visibility in the other conferences.

The *smallest* and *largest* equi-join order plans represent an approximation of the best and worst plans in the search space. Comparing the

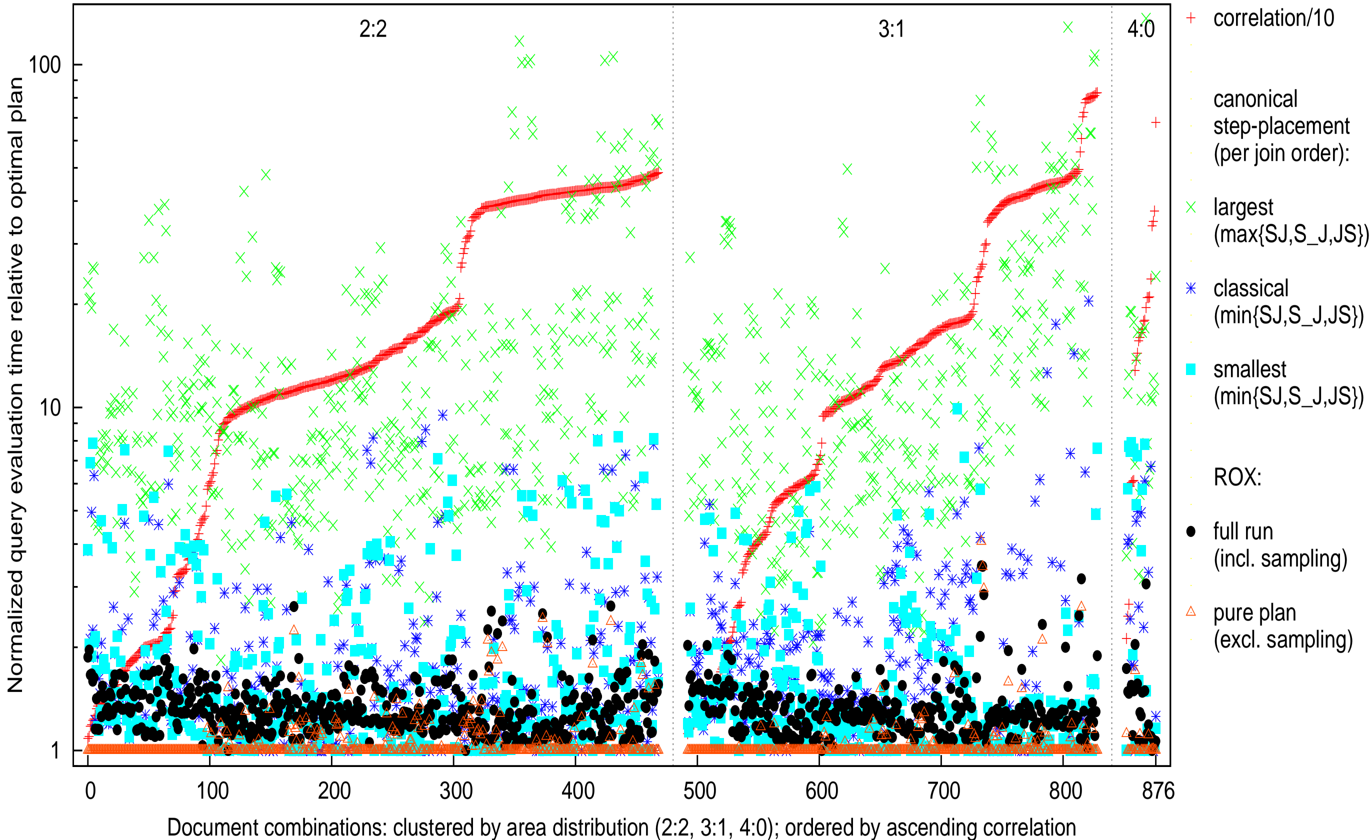


Figure 5.11 Normalized execution time of the 2 plans generated by ROX and the three introduced plan types (*smallest*, *largest*, and *classical*) for each of the 831 document combinations. The document combinations are organized by research area grouping (2:2, 3:1, 4:0), and ordered by correlation measure within each group.

ROX and *classical* plans to these plans, we see that ROX is steadily picking an execution plan that is close to optimal and invariably avoiding the bad plans which the *classical* optimizer, in some cases, ends up picking.

As we have noted earlier, the performance of the classical plan is worse for queries in group 3:1 with high correlation measures (it reaches 2 orders of magnitude difference with the fastest plan). The same behavior is noticeable for the *largest* equi-join order plan, and spans to group 2:2 as well. We have already explained that a higher correlation measure indicates a larger dispersion in the selectivity of the equi-joins. This means that the quality of the final plan is highly sensitive to the join ordering choices. Therefore, any difference in the order between, on the one hand, the operators in the classical and *largest* equi-join order plans, and, on the other hand, the operators in the best shown plan results in the observed steep degradation in the performance. This is not the case with lower correlation measures since the impact of wrongly ordering the joins on the performance of the final plan is not as high.

Taking a closer look at the plot in Figure 5.11, we notice that for few document combinations (around 140 out of 831), the ROX pure plan is not the fastest among all the shown plans. For these 140 queries, ROX is on average 30% slower than the fastest plotted plan. In the worst case, the execution time of the ROX plan is 4 times slower. For 59% of the 140 queries, the fastest plan corresponds to the *smallest* equi-join order plan, while the classical plan is the fastest for the rest 41%. We note that in the latter case, the difference between the execution times of the classical and *smallest* equi-join order plans is minimal. The reason behind the non-optimal choices made by ROX is explained in Section 5.5.8, and stems from the inaccurate estimations returned by the adopted sampling methods.

Overall, we conclude that our ROX optimizer adapts to the correlations effectively, and exploits the selective ones in the optimization process. It has proven to be robust, reliable, and stable in avoiding bad plans and picking execution plans that are (near-)optimal.

5.5.5 Step Placement

In this experiment, we assess the impact of the step placement among the equi-joins on the quality of the execution plan, and the ability of ROX in optimally placing the steps among the joins. Using the $\times 100$ scaled dataset, we compare the ROX pure plan (excluding sampling cost) with the ROX equi-join order plan. The latter shares the same ordering of equi-joins as that of the pure plan chosen by ROX; however, the location of its XPath step operators among the equi-joins corresponds to the fastest of the SJ, JS, and S_J canonical step placements. We note that the step direction between the two plans is also different. In the ROX equi-join order plan, all

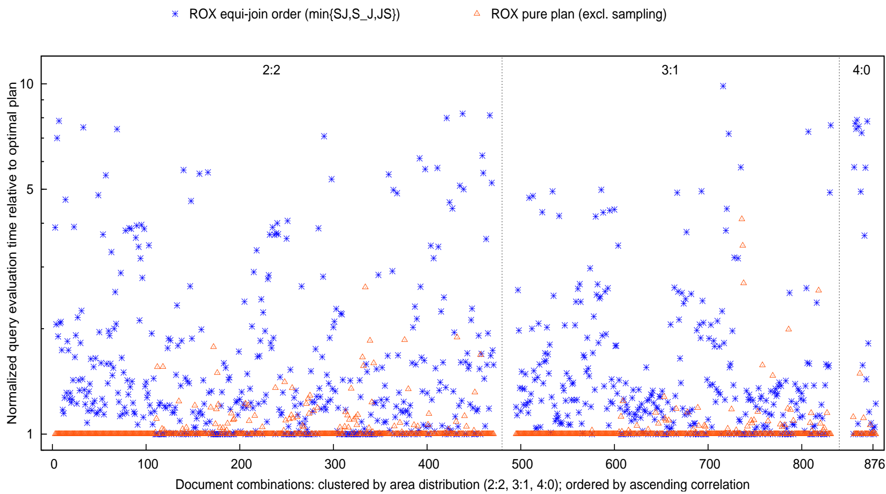


Figure 5.12 Normalized execution time of the ROX pure plan and the ROX equi-join order plan.

steps `author/text()` are executed with a child axis, while the execution direction of the steps in the ROX pure plan is optimized adaptively.

Figure 5.12 illustrates the elapsed execution time of the two considered plans normalized to the fastest plan. The red almost straight line corresponds to the execution time of ROX pure plan. The ROX equi-join order plans shown with blue stars, are in general very close to the ROX pure plans, but for certain document combinations where the XPath step placement and direction matter, the performance is much worse. The ROX equi-join order plans are on average 2 times slower than the ROX pure plan times, while the latter reaches up to 1 order of magnitude faster execution times for some queries. Therefore, we conclude that, *for this set of tested queries, the step placement optimization has less impact on the quality of the plan than the equi-join optimization; however, for some of the queries, determining the wrong order and direction of the steps results in a considerably slower performance*. Our ROX optimizer did not only manage detecting the correlation existing between the authors in the different queried documents, but also *succeeded in determining the best position in which to insert the XPath steps among the equi-joins, and the best execution direction of these steps*.

Examining the reported execution times in Figure 5.12, it appears that, for few document combinations, the ROX pure plan is not the fastest plan among the 2 tested plans. This means that in those cases, ROX makes a non-optimal choice during the optimization process of the query. We explain the origin behind these decisions in Section 5.5.8.

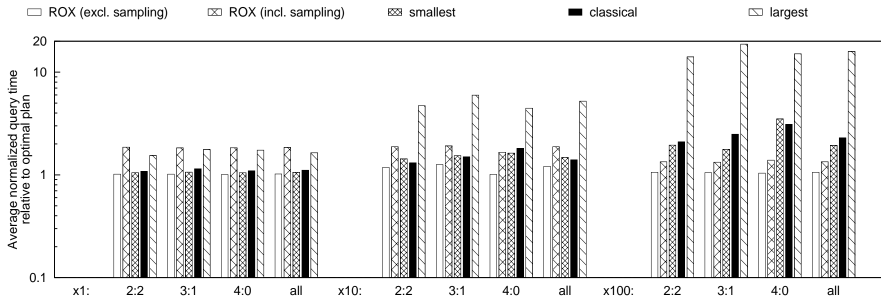


Figure 5.13 Impact of Document Sizes on Sampling Overhead

5.5.6 Sampling Overhead at Different Document Scales

While focusing on the $\times 100$ scaled dataset in the previous experiments, we now down-scale the dataset size to analyze the impact of the document sizes on the performance of ROX, and more specifically the overhead imposed by sampling. Although we expect that with smaller data sets, the time spent on sampling will decrease (but less steep since the sample size stays the same), we hypothesize that the overhead of sampling compared to the query's cost might grow more visible if the query becomes so cheap to execute that the difference between the execution time of the best and worst plans gets considerably smaller.

In this experiment, we again compare the ROX full run plan (including sampling cost) with the ROX pure plan (excluding sampling cost) to the plans corresponding to the *smallest*, *largest*, and *classical* equi-join order classes on the same 831 document combinations. As before, the plan used in the experiment for both the *smallest* and *classical* equi-join order classes corresponds to the fastest canonical step placements (SJ, JS, S_J), while we use the slowest for the *largest* equi-join order class. Figure 5.13 shows the average execution times of each of the plans normalized to the fastest of the 5 using the three datasets: the original DBLP set (scale $\times 1$), as well as scale $\times 10$ and scale $\times 100$.

We notice that the pure plan of ROX is close to optimal for all datasets and the different correlation clusters. While the sampling overhead in the full ROX run is a small portion of the plan's cost in the case of the $\times 10$ and $\times 100$ scaled datasets, it is equal to the pure plan's execution time in the $\times 1$ dataset, making the execution time of the ROX full run slower than the *largest* equi-join order plan. This is caused by the fact that the 4-way join query is so cheap to execute that any plan from the search space would run sufficiently fast. Examining the plot, we in fact see that the difference between the ROX pure plan and the *largest* equi-join order plan is very small in the $\times 1$ dataset (contrary to the $\times 10$ and $\times 100$

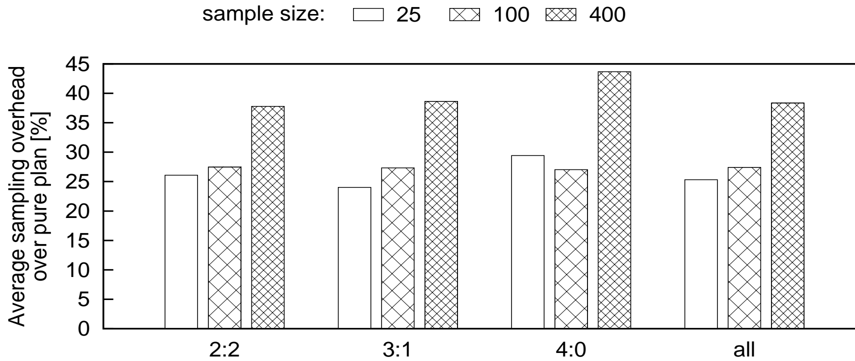


Figure 5.14 Impact of sample size and cutoff limit on sampling overhead.

datasets). We, therefore, conclude that, when using a fixed sample size, the sampling overhead indeed does not reduce as fast as the execution time when querying smaller document sizes. As any optimizer, the use of ROX only pays off with larger document scales.

In fact, less time should have been spent on optimizing the query when using the $\times 1$ dataset since the quality of the final execution plan is almost insensitive to the decisions made by the optimizer. To achieve this, the solution is to equip ROX with a module that adaptively determines the amount of time to grant to the optimization steps. This adaptive decision on the total amount of time to be spent on optimizing a query is considered as future research and will be discussed further in Section 5.5.9.

5.5.7 Impact of Sample Size

In this last experiment, we analyze the impact of the sample size and cutoff limit value on the sampling costs during the ROX query evaluation, and on the quality of the produced ROX plan. We run ROX on the 831 document combinations as before, using the $\times 100$ scale dataset and we vary the value of the sample size and cutoff limit between 25, 100, and 400 tuples. We note that all previous experiments were run with a sample size and a cutoff limit value equal to 100.

With R denoting the execution time of the full ROX run (including sampling) and r denoting the execution time excluding the sampling cost, we define the relative sampling overhead in % as $100 * (R - r)/r$. Figure 5.14 shows for each sampling size and cutoff limit value, the average overhead per correlation cluster. As expected, the overhead increases with the sample size and cutoff limit. The difference between 25 and 100 is marginal, while samples of 400 tuples cause significantly more overhead

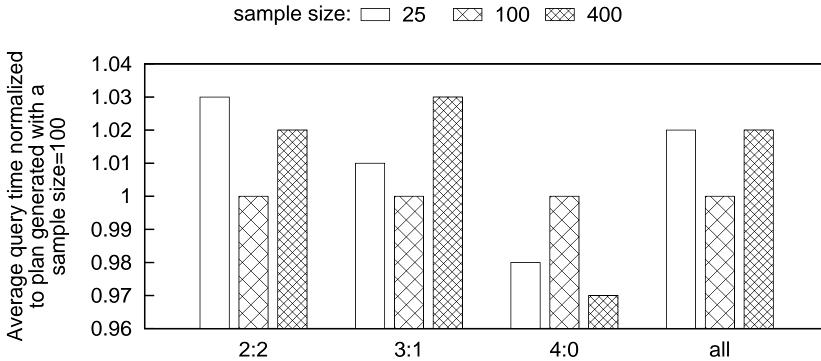


Figure 5.15 Impact of sample size and cutoff limit on quality of produced plans.

than samples of 100 tuples. This observation supports our initial intuitive choice of setting the sample size and the cutoff limit to the value 100 in the previous experiments.

Figure 5.15 shows the average execution time of the plans generated by ROX per correlation cluster when using the three values sample size and cutoff limit. The plotted times are normalized to the execution time of the plan produced with a sample size = 100. We notice that the sample size and cutoff limit value has almost no impact on the quality of the plans chosen by ROX. The difference between the execution times of the 3 generated plans is on average around 2%. We conclude that *ROX is able to robustly find near-optimal plans even when using a smaller sample size*.

5.5.8 Behind the Shortcomings of the Current Implementation of Sampling in ROX

In the experiments shown in Figure 5.11 and Figure 5.12, we discovered few cases in which the plan chosen by ROX was not robustly near-optimal. In this section we identify and describe the reasons behind the occurrence of such cases.

We found two issues that cause the ROX optimizer to fail in finding the optimal plan. The first is the implementation adopted for the cutoff-sampling operation while the second is the use of non-representative starting samples in the chain sampling process. We explain each of these reasons next.

Cutoff-Sampling

In Section 5.2, we have described the implementation of the cutoff-sampling technique stressing that it is front-biased and might generate non-representative

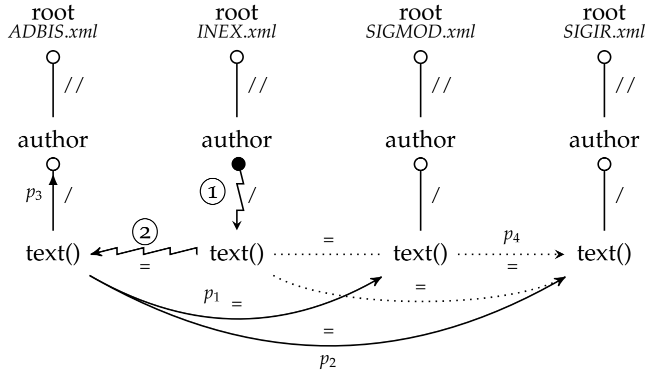


Figure 5.16 Join graph corresponding to the example DBLP query. Two edges are already executed. The circled numbers denote the execution order. The current chain sampling process is illustrated with arrows indicating the sampling direction. The labels on the edges present the path to which a sampled edge belongs to.

Iteration	Cutoff limit	Path	Result size	# of distinct tuples	*hr*
1	200	p_1	200	18	15
		p_2	128	27	0.8
		p_3	100	100	1
2	300	p_4	0	0	0

Figure 5.17 Details about the 2 chain sampling iterations illustrated in the join graph of Figure 5.16. The current implementation of the cutoff-sampling technique results in the introduction of a large amount of duplicates in the sampling result of path p_1. This leads to erroneously estimating the hit ratio of path p_4 to be 0.

sampling results. The latter occurs when the first tuples in the outer sampled table match with multiple tuples in the inner table, possibly producing duplicates in the output. This results in a smaller number of distinct tuples in the sampling output, hence decreasing the representativeness of the sample. With these few distinct tuples used as input to the next sampling operation, the chance is high that the hit ratio of the sampled join is erroneously estimated to be extremely selective or even 0, which might result in ROX picking the wrong sequence of joins for execution.

As an example of such a situation, we consider our DBLP query template using the documents representing the conferences ADBIS, INEX, SIGMOD, and SIGIR. After two optimization and execution steps, ROX initiates a new chain sampling process which is illustrated in the query's join graph shown in Figure 5.16. The circled numbers indicate the order of execution of the already executed edges. Chain sampling begins its explor-

ation from the vertex `text()` (corresponding to the document ADBIS). The label on the edges indicates the path the edges belong to, and the arrows indicate the direction of the chain sampling operations.

Figure 5.17 enumerates the details concerning each of the two executed chain sampling iterations. We note that more paths are sampled during the second iteration, but we omit these from our description. Starting chain sampling with a cutoff limit equal to 200 and a sample of size 100, the ROX algorithm increases the cutoff limit by 100 at the end of each sampling iteration. Among others, the table presents the result size and the number of distinct tuples generated by each sampling operation. In the first iteration, the result of size 200 generated from sampling the edge in p_1 contains only 18 distinct tuples. In the second iteration, the path p_4 is created from the extension of p_1 with a new edge. When sampling the new edge, none of the inner tuples join with the 18 distinct tuples generated by p_1, therefore, an empty result is returned. This is to be expected since with a sample of size 18 as input, the chance to generate a non-empty result is small. With a hit ratio erroneously estimated to be equal to 0, the path p_4 is chosen for execution, although it is the worst path to execute in the join graph (note that the hit ratio of the first edge in path p_4 is estimated to be equal to 15).

Summarizing, the current implementation of the cutoff-sampling technique might introduce duplicate tuples in the result of a sampling operation. If the latter occurs during a chain sampling process, then the subsequent sampling operation might generate only few tuples, if any. As a result, chain sampling will erroneously estimate the hit ratio of the newly created path to be very selective or even equal to 0. This might lead the optimizer to decide to execute a path that is in reality far from the best path to execute. A potential solution to the front-biased cutoff-sampling implementation has been presented in Section 5.2.

Non-Representative Samples

To illustrate the second cause behind the possible shortcoming of ROX in finding a good execution plan, we use the DBLP 4-way join query with XML documents representing the conferences ICDE, WSDM, VLDB, and ICDM. Figure 5.18 illustrates the join graph corresponding to the query after one optimization and execution step have been completed and a new chain sampling iteration has been initiated. Chain sampling uses the vertex `text()` (corresponding to the document `WSDM.xml`) as the starting point of exploration, and uses a sample of nodes S randomly picked from the vertex as input to the sampling operations executed during the first iteration. The figure shows three sampling iterations, and in each iteration a new path is created by extending the path created during the previous iteration with a newly sampled edge. At the end of the third sampling iteration, the path p_3

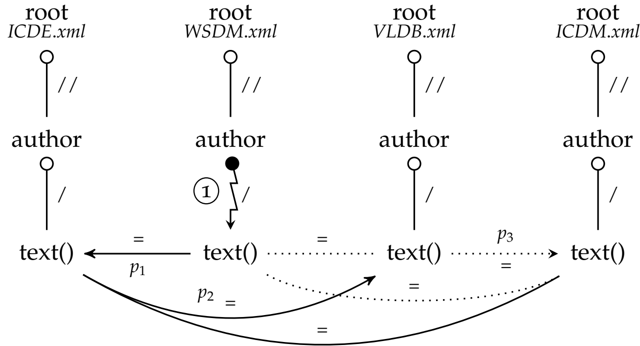


Figure 5.18 Join graph corresponding to the example DBLP query. One edge is already executed as indicated with the circled number. The current chain sampling process is partially illustrated. The labels on the sampled edges denote the path to which the edge belongs, and the arrows represent the sampling direction.

Iteration	Path	Estimated hit ratio	Real hit ratio
1	p_1	0.32	1.5
2	p_2	0.04	13.18
3	p_3	0.16	19.5

Figure 5.19 Details about the chain sampling iterations illustrated in the join graph of Figure 5.18. Using a non representative sample table as input to the chain sampling process results in an estimated hit ratio that is far off the real value. The error in the estimation is then propagated and accumulated as chain sampling progresses, leading to bad paths chosen for execution.

is found superior and is returned for execution. Note that only a selection of the sampled paths is depicted in the figure. Figure 5.19 shows the estimated hit ratio of the sampled paths, along with the real hit ratio. We notice that there exists a considerable gap between the estimated and real values of the hit ratio of path p_1. The reason behind this difference is that the used initial sample S happens to be in this case, not a representative sample of the nodes associated with the starting vertex `text()`. The error in the estimation propagates to the newly created paths. In fact, during the second chain sampling iteration, the error accumulates and grows to two orders of magnitude off the real values. This might lead the optimizer to make wrong decisions and choose the execution of a path, in this case p_3, that appears to be highly selective but is in reality far from the best (note the actual high hit ratio of the path p_3).

Two solutions can be imagined to the potential problem of non-representative samples:

1. The first is to replace our primitive sampling techniques with more advanced and robust ones which return more representative sample sets. A good survey on sampling techniques can be found in [100].

2. A second more efficient solution is to observe, while executing the chosen path, the actual hit ratio of the edges. If the noted values do not match the estimated values, then the execution is stopped and a new optimization step of ROX is initiated. This latter solution is comparable to the "re-optimization" approaches proposed in [79, 91], and is left as possible future extension to ROX. We discuss the applicability of such a technique in ROX in Chapter 7.

5.5.9 Balancing Between the Optimization and Execution Times of a Query

As we have seen in Section 5.5.6, for certain queries that are very cheap to execute and therefore require little or no optimization, the sampling overhead might reach a large percentage of the execution time of the query. ROX, in its current form, does not attempt to strike a balance between the optimization time and the execution time of a query, but since ROX moves optimization to run-time, it becomes possible to define a good balance between the amount of time to spend on both optimizing and executing a query.

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. To overcome the above, the optimizer should, during the search space exploration, decide at every point in time whether to proceed with optimizing the query or stop and return for execution the best plan found so far. Making such a decision at compile-time is not trivial, as it requires a good prediction scheme that estimates the potential running time of the query.

Our ROX method of intertwining query optimization and evaluation finally provides a way to balance resources spent on query optimization and evaluation. ROX can adaptively decide the amount of time to invest on optimization by estimating the potential execution cost of the query based on the result size and execution time of the joins observed through sampling. If, on the one hand, the estimated execution cost is small, then the amount of time to spend on optimization should be kept limited. If, on the other hand, the optimization time spent so far reaches a considerable percentage of the estimated execution cost, then optimization should be stopped and the *current plan* can be returned for execution.

To adopt the above strategy, the following questions need to be first addressed. For a given query Q:

1. How can ROX determine the *current plan* corresponding to Q and how can it estimate its execution cost?
2. How to determine whether the execution cost of the current plan is satisfyingly fast, such that optimization can be stopped and the plan returned for execution?
3. What is the acceptable time to spend on optimizing the query Q?

The *current plan* of a given query Q is the plan that ROX would return for execution if it would decide for some reason to stop the optimization of Q. The best possible plan which is easily determined is a plan which orders the edges in the join graph in a greedy manner. This translates to a plan in which the edge with the smallest weight is executed first, followed by the edge with the second smallest weight, and so on until all edges are executed.

A naive approach to estimate the cost of the current plan is to estimate the execution cost of each individual edge, to extrapolate the costs, and then compute the sum. The estimation of the execution time of every edge can be performed as part of the weight computation process. The cost of the current plan is then equal to the sum of the execution time of the first edge in the plan and the extrapolated execution time of the subsequent edges. If the join hit ratio of the first edge in the plan is 0.5, then the execution time of the second edge in the plan is linearly extrapolated to half its originally estimated cost. Similarly, the execution time of the third edge in the plan is linearly extrapolated using the join hit ratio of the first two operators, and so on.

After each execution step in ROX, some edges in the graph are executed and the knowledge in the join graph is updated. Consequently the current plan is redefined: it consists of the already executed edges in the order of execution determined by ROX, while the unexecuted edges are ordered in a greedy manner. The cost of the plan is accordingly updated to the sum of the time spent so far on the execution, with the execution time of the first unexecuted edge in the plan, and the linearly extrapolated execution time of the other unexecuted edges in the plan.

To position the current plan in the search space of plans, the cost of a possible bad plan needs to be determined. To estimate the cost of a possible bad plan, a simple technique is to estimate the cost of the plan in which the edges in the join graph are ordered in the worst greedy manner. If the execution costs of the current plan and the bad plan are close, then it might either be the case that the query is considerably cheap such that the difference between the optimal and worse plan is small, or that the current plan is far from optimal and optimization should proceed. In both cases,

ROX will spend some time on optimization to improve the current plan while keeping in mind that the cost of the current plan and the time spent on optimization should be and remain far below the execution time of the bad plan.

Knowing the execution cost of the current plan C, and the time O spent so far by ROX on optimization, ROX can then decide if optimization should proceed or stop based on the following formula: $O = x\% \times C$. If the optimization cost of the query reaches a certain percentage x of the estimated execution time of the current plan, then optimization should be stopped. It remains to determine the best value of the percentage x. This is left for future work.

5.6 Conclusion

In this chapter, we have given a brief description of MonetDB/XQuery, the database backend on top of which the ROX prototype is implemented. We have described the storage structure in MonetDB, the staircase join operator, and the type of indexes supported by MonetDB. We have also explained the implementation of the sampling and cutoff-sampling techniques used by ROX. We mention that the sampling techniques and the introduced cutoff-sampling approach implemented in the ROX prototype are very primitive. More advanced and robust techniques can certainly be adopted, and will most probably improve the results currently achieved by ROX. Our decision to stick to these basic techniques stems from our wish to test and prove the robustness of ROX, even under primitive conditions. Although ROX is implemented and tested in the context of MonetDB/XQuery, we emphasize that the ideas in ROX can be used with other systems and do not require the presence of the structural staircase join operator nor indexes of MonetDB. In fact, the applicability of ROX depends only on the existence of efficient and cheap sampling techniques that satisfy the zero-investment property.

In the conducted experiments, ROX was tested using more than 800 queries in the context of two documents: XMark, a synthetic document used to benchmark XML database systems, and DBLP, a document containing real-life data. The following points were made while observing the experimental results:

1. **Robustness of ROX:** Even when using primitive sampling approaches and a front-biased cutoff-sampling technique, ROX has proven to be a robust optimizer: it successfully and consistently picked execution plans that are (near-)optimal, always avoiding the bad ones, even with smaller sample sizes. On the other hand, the considered classical compile-time optimizer has shown strong variations in the quality of

the chosen plans, reaching, for some queries two orders of magnitude slower performance. For around 84% of the tested queries, ROX was capable of finding an optimal execution plan. For the other 16%, the optimal plan is missed due to erroneous estimations returned by the currently adopted primitive sampling techniques and the front-biased cutoff-sampling approach. Solutions have been proposed to the above two problems. For the 16% cases, the experiments have shown that the ROX plan is on average only 30% slower than the fastest tested plan, and still far from the worst plan.

2. **Seamless handling of XPath steps and relational joins:** Although the step placement optimization in the context of the tested DBLP queries has less impact on the quality of the plan than the equi-join optimization, ROX was capable of defining a good ordering of the relational joins in the join graph as well as placing the XPath steps in strategic locations in the execution plan. Moreover, ROX dynamically determined the best execution direction of both types of joins. We stress again that the possibility to optimize several join graphs in a single plan allows ROX to handle the full XQuery language.
3. **Detection of correlation:** We have mentioned in Chapter 4 that our chain-sampling technique provides the first generic and robust method to deal with any type of correlated data. In fact, the experiments have proven that ROX is indeed capable of adaptively detecting the correlations existing among the queried data, and of exploiting it to generate good execution plans.
4. **Efficiency of ROX:** Moving optimization to run-time comes with the challenge of keeping resource usage under control. The experiments have shown that the sampling overhead imposed by the run-time optimization in ROX is kept limited, and is on average around 27% of the full execution time for a sample size equal to 100.

As a summary, the experiments have shown that ROX, our autonomous run-time optimizer for XQueries, is robust and efficient, capable of overcoming the challenges that classical compile-time optimizers are still facing.

6 ROX-sampled: Towards a Pipelined Execution

Basically, ROX consists of iteratively alternating optimization phases with execution steps in which complete tables are joined and intermediate results are fully materialized. Noticeably the execution of operators using full tables and the materialization of full results are behind the robustness of ROX, but they also restrict ROX to systems that support full materialization, and hence exclude systems with a pipelined execution scheme from adopting the ROX approach.

In a pipelined system, a plan is executed iteratively. In each iteration, an operator processes a small set of data and then pipes the result to the next operator in the plan. This means that only small amounts of data are processed and kept in memory at each iteration. As each execution step in ROX processes complete tables and materializes full results, it becomes impossible to directly apply ROX in a pipelined system.

Since most of the existing database management systems, *e.g.* Oracle RDBMS, IBM DB2, Microsoft SQL server, MySQL, PostgreSQL, *etc.*, adopt a pipelined execution style, it is important to develop a variant of ROX that is suitable for these systems. In this chapter, we first present this variant of ROX in which the execution of operators with full tables is not required reducing the amount of materialized intermediate results. This is followed by a detailed description of the algorithm, and an experimental section that compares the evaluation of the new variant of ROX to ROX with full materialization. Finally, we take a look at the different requirements that a pipelined system should support to be able to efficiently implement ROX.

6.1 General Description of ROX-sampled

This section introduces ROX-sampled, a variant of ROX that is suited for systems with a pipelined execution style. The main difference between ROX-sampled and ROX-full is in their execution phases. In ROX-full,

execution steps manipulate full tables, while in ROX-sampled sample tables are processed. This difference impacts the types of tables associated with vertices, and the way edges are sampled and executed in ROX-sampled. This section is structured as follows:

- A general description of ROX-sampled is given in Section 6.1.1.
- The tables associated to vertices in a join graph are introduced in Section 6.1.2.
- The methods used by ROX-sampled to execute and sample edges are explained in Section 6.1.3 and Section 6.1.5
- A special case of executing and sampling edges with two executed vertices is discussed in Section 6.1.4 and Section 6.1.6.

6.1.1 Global Overview of ROX-sampled

One of the main reasons behind the robustness of ROX, referred to hereafter as ROX-full, is that every optimization phase is followed by an execution step which implements the decisions made by the optimizer, and materializes the full intermediate results. This gives the next optimization phase the benefit of using the up-to-date intermediates to retrieve accurate information about data characteristics, which in turn allows for an accurate estimation of cardinalities and the detection of correlations. But this strategy is not suitable for pipelined systems in which the execution of operators using full tables and the materialization of the complete intermediate result is avoided as much as possible.

To generalize the ROX approach to systems with a pipelined execution, we modify ROX-full into a new variant which we name ROX-sampled. *ROX-sampled runs the ROX algorithm using only data samples while recording the decisions made during each optimization phase, and when the whole join graph is optimized, it executes the saved decisions using the full tables.*

Figure 6.1 illustrates the two variants of ROX. ROX-sampled follows the same steps as ROX-full, with the main difference that only samples are used throughout the whole algorithm. Therefore, unlike ROX-full, where the optimization phase works with a data sample while the execution phase processes full tables, both the optimization and execution phases of ROX-sampled manipulate data samples. This means that during the execution phases of ROX-sampled, operators are sampled instead of being executed with full tables. We note that the size of the sample sets used during the execution phases of ROX-sampled might be larger than those used during the optimization phases. We also stress that although an operator *op* in ROX-sampled is *sampled* during an execution phase, we still refer to the process as *executing op*. Therefore, when during an execution phase, the term *executing* the operator *op* is used, it in fact means that *op* is being *sampled* (*i.e. not executed* with full tables).

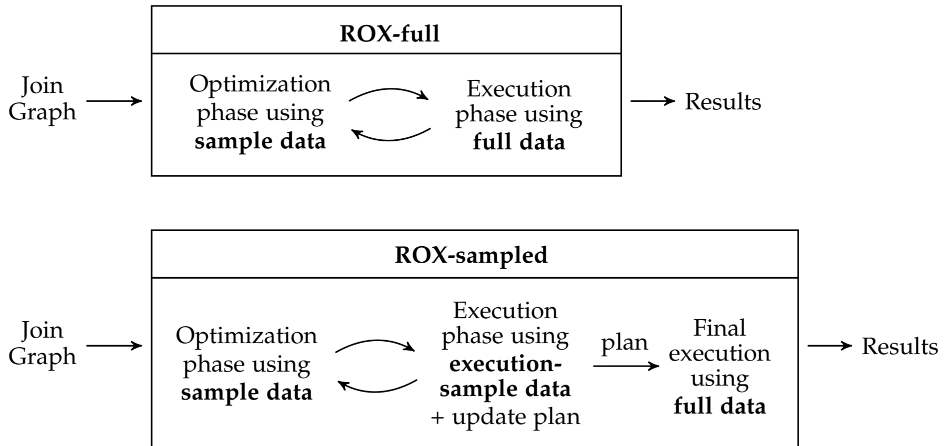


Figure 6.1 An illustration of the steps of ROX-full and ROX-sampled. Both techniques define the execution plan iteratively, but each execution phase in ROX-full processes full tables while execution phases in ROX-sampled process only data samples. As a result, in ROX-full, the plan is defined and executed incrementally, and in ROX-sampled, the execution plan is first entirely defined and then evaluated.

So, in ROX-sampled, every optimization phase decides which operators should be executed, and the subsequent execution phase executes these operators using data samples instead of full tables. *The execution decisions made during each optimization phase are recorded. These decisions make up the final execution plan*. When ROX-sampled completes the optimization of the whole join graph, *i.e.* when the execution order of each operator in the join graph has been determined, the final execution plan is executed with the *full tables* and the result is returned for subsequent processing. As shown in Figure 6.1, both variants construct their execution plans incrementally; however, ROX-full executes directly each operator added to the plan, while ROX-sampled executes the whole plan once it is completely defined.

The fact that the execution phases of ROX-sampled process data samples instead of full tables results in intermediate results of a considerably smaller size. By limiting the number of tuples accessed from base tables, processed and materialized at every execution step, it becomes possible to apply the ROX idea to pipelined systems. Accessing fewer tuples from base tables, and manipulating less data throughout the algorithm allows the use of ROX in pipelined systems, but might also increase the risk of constructing bad execution plans. In fact, our proposed ROX-sampled approach raises some questions. Does the use of only small samples during both the optimization and execution steps jeopardize the robustness of the algorithm? Will the small generated intermediates be representative enough

to detect data correlations? As will become clear later in Section 6.1.4 and Section 6.1.6, ROX-sampled needs, in some situations, to perform redundant operations. Will this reduce the efficiency of ROX-sampled? These concerns will be addressed in Section 6.3 which presents the results of a set of experiments conducted to assess the quality of the execution plans chosen by ROX-sampled, and to investigate the performance and efficiency of the new algorithm.

6.1.2 Tables Associated With a Vertex

As we have seen in Chapter 4, in ROX-full two types of tables are associated with each vertex in the join graph. We redescribe these below.
For a given vertex v in a ROX-full join graph, we denoted:

- $TBL(v)$ as the *full table* containing all XML nodes corresponding to the vertex v. $TBL(v)$ is used during the execution phases of ROX-full as input to operators. After the execution of an edge, the content of $TBL(v)$ is updated with the tuples in the generated result.
- $SMPL(v)$ as a *sample* randomly chosen from $TBL(v)$. $SMPL(v)$ is used as input to the sampling operations during the optimization phases of ROX-full.

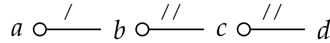
In the case of ROX-sampled, three types of tables are associated with each vertex v in the join graph:

- $TBL(v)$ is the *full table* containing all XML nodes corresponding to the vertex v.
- $ESMPL(v)$ denotes the *execution-sample table* of v. It is used *during the execution phases* of ROX-sampled as input to operators. Initially, $ESMPL(v)$ contains a random sample picked from $TBL(v)$, and later it consists of the output of the execution operations. Therefore, after each execution, the content of the table $ESMPL(v)$ is updated with the tuples in the generated result.
- $SMPL(v)$ is a *sample* randomly chosen from $ESMPL(v)$. $SMPL(v)$ is used as input to the sampling operations *during the optimization phases* of ROX-sampled.

Note that although the execution-sample table $ESMPL(v)$ and the sample table $SMPL(v)$ are both data samples, they are in fact two different physical tables with two main differences:

- $ESMPL(v)$ is used during the execution phases of ROX-sampled, while $SMPL(v)$ is used during the optimization phases of the algorithm.
- The size of $ESMPL(v)$ is slightly bigger than $SMPL(v)$. We choose to increase the number of tuples in $ESMPL(v)$ to allow for more

Join graph



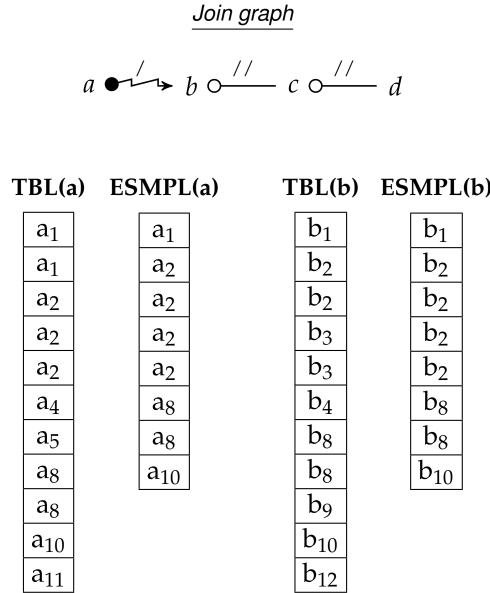
Initialized tables

TBL(a)	ESMPL(a)	SMPL(a)	TBL(b)	ESMPL(b)	SMPL(b)
a_1	a_1	a_1	b_1	b_2	b_3
a_1	a_2	a_2	b_2	b_3	b_9
a_2	a_2	a_{10}	b_2	b_3	b_{10}
a_2	a_8		b_3	b_9	
a_2	a_{10}		b_3	b_{10}	
a_4			b_4		
a_5			b_8		
a_8			b_8		
a_8			b_9		
a_{10}			b_{10}		
a_{11}			b_{12}		

Figure 6.2 An illustration of the types of tables associated with vertices in a ROX-sampled join graph. The initialized full, execution-sample, and sample tables associated with the vertices a and b are shown. For a given vertex v, the full table $TBL(v)$ contains all XML nodes in the document that correspond to v. The execution-sample table $ESMPL(v)$ is initialized to a sample randomly chosen from the full table $TBL(v)$, while the sample table $SMPL(v)$ consists of a random sample picked from $ESMPL(v)$.

representative samples, and for the generation of larger results. The intuition behind this is to counteract the risk of generating estimation errors through sampling, and the propagation of these errors to subsequent execution steps. The size of $SMPL(v)$ is not increased to keep the sampling overhead limited.

Example 6.1.1. The above definitions are illustrated in Figure 6.2 in which a join graph is depicted along with the full, execution-sample, and sample tables of the two vertices a and b. The full table $TBL(a)$ (respectively, $TBL(b)$) contains all the XML nodes in the document corresponding to the vertex a (respectively, b). Each tuple in the tables represents an XML node. The subscripts represent the relations between XML nodes belonging to different tables, *i.e.* the tuples represented by a_1 match with the tuples designated by b_1. The execution-sample table $ESMPL(a)$ (respectively, $ESMPL(b)$) is initialized to a sample randomly picked from the full table $TBL(a)$ (respectively, $TBL(b)$), and the sample table $SMPL(a)$ (respectively,



Tables after executing edge (a, b)

Figure 6.3 The full and execution-sample tables associated with the vertices a and b after edge (a, b) has been executed. The execution step uses the table $ESMPL(a)$ as context set for the execution of the step operator. The execution-sample tables of a and b are updated to reflect the execution result, while the data in the full tables is not modified.

$SMPL(b)$) is initialized to a random sample chosen from the table $ESMPL(a)$ (respectively, $ESMPL(b)$).

Content of Execution-Sample Tables after the Execution of Edges

In this section, we take a closer look at execution-sample tables, more specifically their content and use in execution phases of ROX-sampled.

Example 6.1.2. We reconsider the join graph in Figure 6.2, and we suppose that the optimization phase of ROX-sampled decides to execute the edge (a, b). The resulting join graph and the content of the full and execution-sample tables of the two vertices a and b after the execution of (a, b) is depicted in Figure 6.3. Since execution phases in ROX-sampled consist of sampling operators, they therefore manipulate and update execution-sample tables instead of full tables. The arrow in the join graph indicates the execution direction, therefore the table $ESMPL(a)$ - with the content shown in Figure 6.2 - is used as the context set for the executed step

operator. Note that the execution sample table $ESMPL(b)$ is not used for the execution of the edge. In fact, as we will explain in the following section, the edge is executed by matching the execution sample table $ESMPL(a)$ with the full table $TBL(b)$. After the execution of the edge, the tables $ESMPL(a)$ and $ESMPL(b)$ are updated to the result of the execution (as shown in Figure 6.3). We note that the data in the full tables $TBL(a)$ and $TBL(b)$ is not updated. In fact, in ROX-sampled, the content of full tables is never modified.

Definition 6.1.3. (refined in Definition 6.1.4) The content of the execution-sample table $ESMPL(v)$ of a vertex v is defined as:

- a random sample of tuples chosen from the full table $TBL(v)$, if v is *not* an executed vertex.
- the tuples that have matched all the executed edges of v, if v is an executed vertex.

We recall the definition of an executed vertex presented in Section 3.3. An executed vertex is a vertex that has at least one of its edges already executed: v is an executed vertex if $|edges^{+}(v)| > 0$.

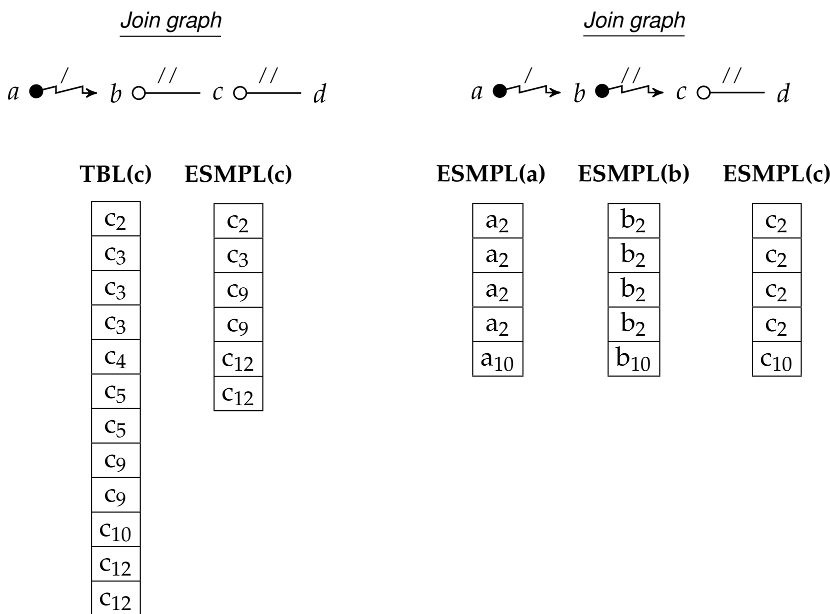
Continuation of Example 6.1.2. One optimization step further, the ROX-sampled algorithm decides to execute the edge (b, c). The full table and the execution-sample table of the vertex c are illustrated in Figure 6.4a. Since c is not an executed vertex, $ESMPL(c)$ contains a random sample chosen from $TBL(c)$. Figure 6.4b shows the join graph after the edge (b, c) is executed by matching the tables $ESMPL(b)$ and $TBL(c)$. The execution-sample tables of b and c are updated to the result of the execution. Note that the content of $ESMPL(a)$ is also updated. Therefore, the content of $ESMPL(a)$ reflects the execution result of not only the executed edges outgoing from a, but also the execution result of the chains of executed edges branching from a: the tuples in $ESMPL(a)$ are those random tuples that have matched the execution of both edges (a, b) and (b, c).

We therefore refine our previous definition (**Definition 6.1.3**) of the content of the execution-sample table of a vertex v.

Definition 6.1.4. The content of the execution-sample table of a vertex v is defined as:

- a random sample of tuples chosen from the full table $TBL(v)$, if v is *not* an executed vertex.
- the tuples that have matched the chains of executed edges branching from v, if v is an executed vertex.

Briefly said, the execution-sample tables in ROX-sampled have the same role as full tables have in ROX-full, with the main difference that the size



a The join graph of Figure 6.3 after the execution of edge (a, b). The full table and the execution-sample table associated with the vertex c are shown. Since vertex c is not an executed vertex, its execution-sample table is a random set of tuples chosen from $TBL(c)$.

b The join graph after the execution of edge (b, c). The execution-sample tables of the vertices a, b, and c are shown. The edge (b, c) is executed using as context set the execution-sample table $ESMPL(b)$. Note that the content of $ESMPL(a)$ is also updated to the result of the execution of (b, c).

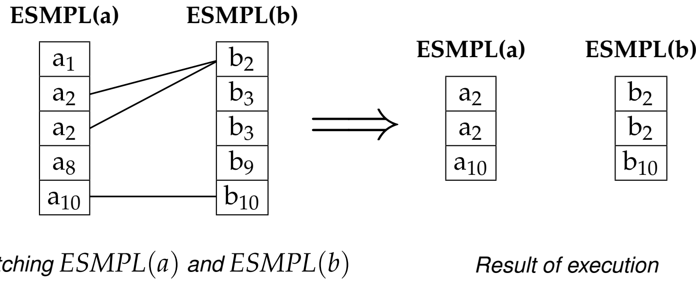
Figure 6.4 The execution-sample table of an executed vertex v contains all the tuples that match the previously executed chains of edges branching from v.

of execution-sample tables is considerably smaller than that of full tables. We stress that in ROX-sampled, full tables are never modified during the complete algorithm.

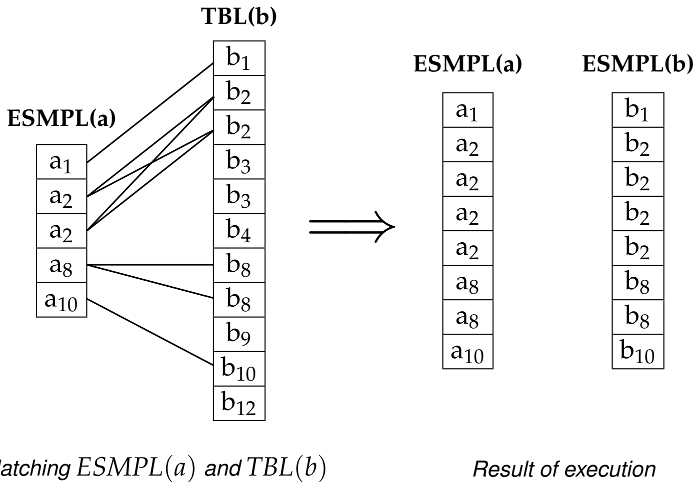
6.1.3 The Execution of Edges

In this section, we describe the method used by ROX-sampled to execute a given edge in the join graph. We first explain the general case, then focus in the next section on the special case of executing an edge with two executed vertices.

Let us take a closer look at Figure 6.3, more specifically at the result of the execution of the edge (a, b). We note that the edge was not executed as a join between the execution-sample tables $ESMPL(a)$ and $ESMPL(b)$, but as a join between $ESMPL(a)$ and $TBL(b)$ (the 3 tables are shown in Fig-



a Executing edge (a, b) of the join graph in Figure 6.2 using as input the execution-sample tables $ESMPL(a)$ and $ESMPL(b)$.



b Executing edge (a, b) of the join graph in Figure 6.2 using as input the execution-sample table $ESMPL(a)$ and the full table $TBL(b)$.

Figure 6.5 Two ways to execute edge (a, b) of the join graph in Figure 6.2.

ure 6.2). If the edge was executed by matching the execution-sample tables $ESMPL(a)$ and $ESMPL(b)$, the resulting tables would have the content shown in Figure 6.5a.

We have already explained in Section 3.2.1 that joining two samples, each picked from a given table, does not result in a representative sample of the output of the join of the two tables. In other words, $\triangleright_{\mathtt{r}}(R) \bowtie \triangleright_{\mathtt{t}}(T) \neq \triangleright_{\mathtt{LIMIT}}(R \bowtie T)$. Since the tables $ESMPL(a)$ and $ESMPL(b)$ are both sample sets, the edge (a, b) should not be executed as a join between the execution-sample tables $ESMPL(a)$ and $ESMPL(b)$.

Since the execution of an edge in ROX-sampled consists of merely

a sampling operation, the same technique employed to sample a join operator in ROX-full is used to *execute* an edge in ROX-sampled. Sampling an operator is performed by joining a sample set chosen from one of the operands with the full table of the other operand. Therefore, the edge (a, b) is executed using as input the execution-sample table $ESMPL(a)$ and a step operator into the full table $TBL(b)$. The matching of the tables is illustrated in Figure 6.5b. The same approach is used to execute the edge (b, c) in Figure 6.4b: the edge is executed with as context set the execution-sample table $ESMPL(b)$ and a step operator into the full table $TBL(c)$.

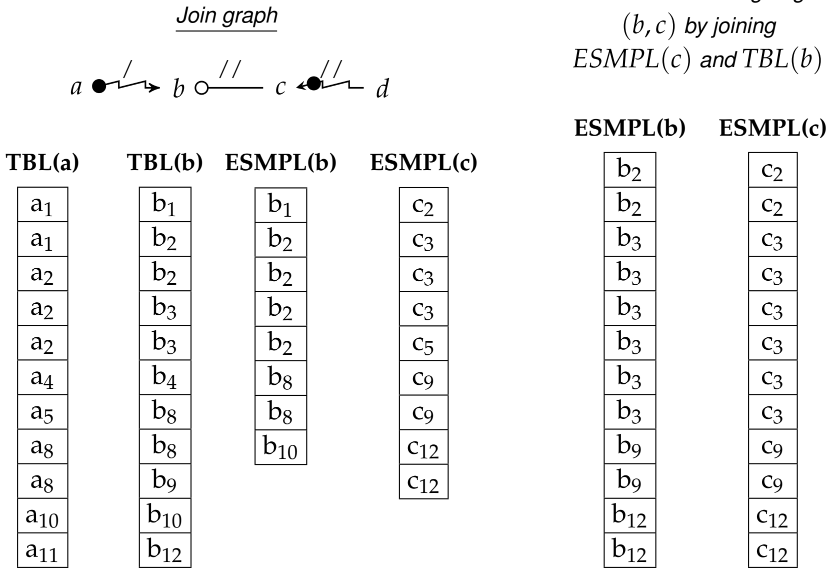
6.1.4 Executing an Edge with Two Executed Vertices

In this section, we elaborate on the method used by ROX-sampled to execute an edge with two executed vertices. We will first explain, using an example, why the execution approach we have described in the previous section is not suitable to execute edges with two executed vertices. Then, we present the technique used by ROX-sampled to execute this kind of edges.

We consider the join graph shown in Figure 6.6a, in which the edges (a, b) and (c, d) are already executed. The arrows indicate the execution direction of the edges. In the figure, the execution-sample tables of the vertices b and c are depicted, along with the full tables of a and b. The ROX-sampled algorithm decides now to execute the edge (b, c), using the vertex c as the context set of the execution. The previously described solution says that the execution-sample table $ESMPL(c)$ should be joined with the full table $TBL(b)$. The result of such an execution is shown in Figure 6.6b.

Since the content of full tables in ROX-sampled is never modified, $TBL(b)$ consists of those XML nodes corresponding to the vertex b. Therefore, the join between $ESMPL(c)$ and $TBL(b)$ results in all the tuples that match the previously executed edge (c, d) and the edge (b, c). This means that the performed execution of the edge (b, c) did not take into account the previously executed edge (a, b). In fact, since b is an executed vertex, the content of its execution-sample table $ESMPL(b)$ after the execution of (b, c) should consist of all the tuples that match the chains of previously executed edges branching from b, namely (a, b), (b, c), and (c, d). If we examine the content of the full table $TBL(a)$ shown in Figure 6.6a and that of the execution-sample table $ESMPL(b)$ in Figure 6.6b, we notice that some of the b tuples, more specifically b_3, b_9, and b_{12}, do not have a match in $TBL(a)$ and would not make it to the result if the executed step (a, b) would have been taken into account.

We conclude that, since the execution-sample table of an executed vertex consists of those tuples that match the chains of executed edges branching from the vertex, the method described in the previous section



a Join graph where edges (a, b) and (c, d) are executed. The full tables associated with vertices a and b are shown, along with the execution-sample tables of vertices b and c.

b Executing edge (b, c) by matching $ESMPL(c)$, the execution-sample table of c, with $TBL(b)$, the full table of b. The shown tables represent the result of the matching.

Figure 6.6 Join graph in which the edge (b, c) is an edge with two executed vertices. The edge (b, c) is executed by joining $ESMPL(c)$ with $TBL(b)$, and the result of the execution is shown. The execution ignores the previously executed chains of edges branching from b, namely (a, b).

is not suitable when executing an edge with two executed vertices, as it results in ignoring the chains of previously executed edges branching from one of the edge's vertices.

Problem statement - The problem we are solving in this section is the following: given an edge $e = (v, v')$ where v and v' are executed vertices, how should ROX-sampled execute e to generate a reliable sampled result which reflects the real data characteristics, taking into account the chains of all previously executed edges branching from the vertices of e.

The execution depicted in the example shown in Figure 6.6 is incomplete since it did not take into account the previously executed edge (a, b), and therefore information about the tuples that have matched (a, b) is lost. To take the execution result of (a, b) into account, we propose to re-execute the

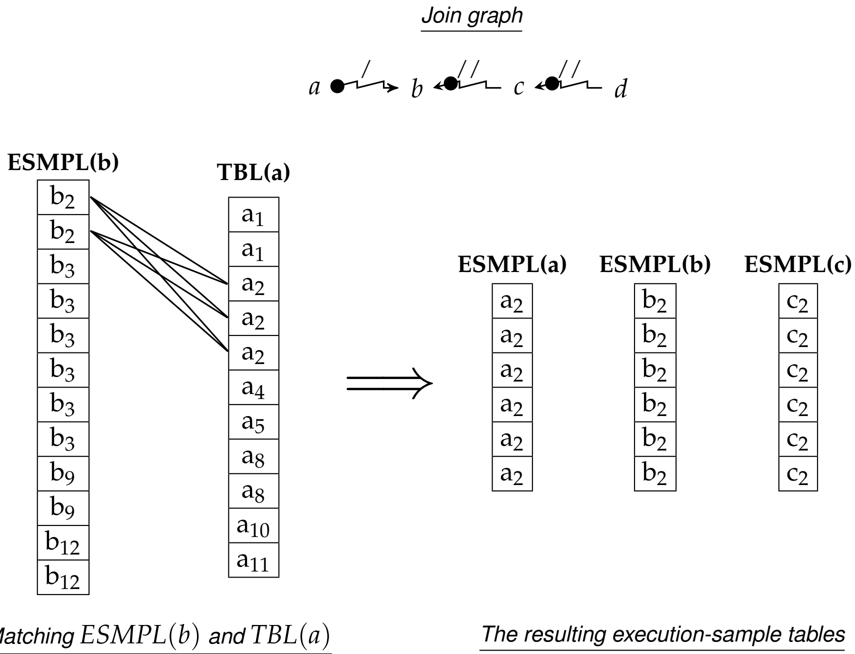
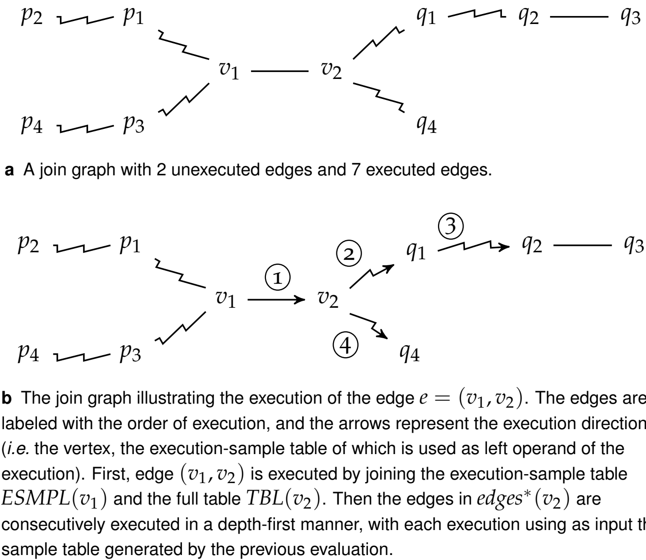


Figure 6.7 Completing the execution of edge (b, c) by re-executing the edge (a, b). After having executed the edge (b, c), edge (a, b) is executed using as input the execution-sample table $ESMPL(b)$ and the full table $TBL(a)$. The content of the execution-sample tables $ESMPL(a)$, $ESMPL(b)$, and $ESMPL(c)$ is updated with the result of the execution. The three execution-sample tables contain the tuples that have matched the execution of the edges (a, b), (b, c), and (c, d).

edge (a, b) using as input the execution-sample table $ESMPL(b)$ generated from the execution of (b, c). This solution is illustrated in Figure 6.7. After executing the edge (b, c), edge (a, b) is re-executed by joining the execution-sample table $ESMPL(b)$ with the full table $TBL(a)$. The resulting execution-sample tables $ESMPL(a)$, $ESMPL(b)$, and $ESMPL(c)$ now consist of the tuples that match the edges (a, b), (b, c), and (c, d).

Solution - To execute an edge $e = (v, v')$ with two executed vertices, we first execute e by joining the execution-sample table of one of the vertices, say v, with the full table $TBL(v')$. We then re-execute the chains of previously executed edges branching from v'. In other words, all edges in $edges^*(v')$ are re-executed.

The decision which vertex (v or v') of the edge e to use as right operand is made by comparing the number of executed edges in the branches of each



a A join graph with 2 unexecuted edges and 7 executed edges.

b The join graph illustrating the execution of the edge $e = (v_1, v_2)$. The edges are labeled with the order of execution, and the arrows represent the execution direction (*i.e.* the vertex, the execution-sample table of which is used as left operand of the execution). First, edge (v_1, v_2) is executed by joining the execution-sample table $ESMPL(v_1)$ and the full table $TBL(v_2)$. Then the edges in $edges^*(v_2)$ are consecutively executed in a depth-first manner, with each execution using as input the sample table generated by the previous evaluation.

Figure 6.8 An example of executing an edge with two executed vertices.

of v and v', *i.e.* by comparing the number of edges in the two sets $edges^*(v)$ and $edges^*(v')$. To keep the cost of re-execution limited, the right operand is chosen to be the vertex v with the smallest number of executed edges. A more detailed description of the method used to make the choice is given in Section 6.2.2.

Example 6.1.5. We illustrate now the proposed solution using the join graph of Figure 6.8a. We suppose that the current optimization phase of ROX decides to execute the edge $e = (v_1, v_2)$. Since both v_1 and v_2 are executed vertices, the execution of e will require the re-execution of the chains of executed edges branching from one of its vertices. Since $|edges^*(v_1)| > |edges^*(v_2)|$, the vertex v_2 is chosen to be the right operand of the execution. Figure 6.8b illustrates the execution process. Edges are executed in a depth-first manner, and are labeled with the execution order, while the arrows represent the execution direction (*i.e.* the vertex that is used as left operand in the execution). First, the edge e is executed using as input the execution-sample table $ESMPL(v_1)$ and the full table $TBL(v_2)$. Then all the edges in $edges^*(v_2)$ are re-executed consecutively, in a depth-first manner. Each execution operation uses as left input the execution-sample table generated by the previous execution. For instance, the edge (v_2, q_1) is re-executed by joining the table $ESMPL(v_2)$ generated by the execution of (v_1, v_2) with the full table $TBL(q_1)$. This gives ROX

the ability to capture the relation and the existing correlations between the different vertices. It also allows to correctly handle the re-execution of possible cycles in the graph.

One question to ask concerns the amount of overhead incurred by the re-execution of edges? We note that these re-executions consist of cutoff-sampled operations which use execution-sample tables as input. The latter are originally a small set of tuples sampled from the full tables, and their size is kept small through cutoff-sampling throughout the complete ROX-sampled algorithm. Therefore, we believe that all the extra re-executions will not add a considerable amount of overhead. We investigate this matter in the experiments presented in Section 6.3.

6.1.5 Differences Between the Execution and Sampling of Edges

Since edges are sampled in both the execution and optimization phases of ROX-sampled, one may wonder if the sampling of an edge e performed during an optimization phase differs from the sampling of e during an execution phase.

The first difference between the two sampling operations is that the one performed during the execution phase uses as left operand an execution-sample table, while during optimization the left operand consists of a sample table. We have already mentioned in Section 6.1.2 that the size of execution-sample tables is larger than that of sample tables, to allow for more representative data samples and for the generation of more tuples to counteract the possible creation and propagation of estimation errors during sampling. Therefore, the value of the cutoff limit used to sample edges during execution phases is chosen to be bigger than the one used during the optimization phases. The intuition behind our choice is to allow for the generation of a larger number of tuples, in an attempt to keep the data in the execution-sample tables a representative sample of the full result of the executed joins, such that the chance of producing future estimations errors is kept small. However, the catch is that the generation of more tuples increases the run-time overhead. In the experiments presented in this Section 6.3, we investigate the impact of increasing the cutoff limit on both the quality of the selected plan and the run-time overhead.

The execution and sampling of an edge $e = (v_1, v_2)$ consists therefore of respectively the following two operations:

$$\trianglerighteq_{\tau'}(op(e, \mathbf{ESMPL}(\mathbf{v_1}), TBL(v_2))) \text{ during an execution phase}$$
$$\trianglerighteq_{\tau}(op(e, \mathbf{SMPL}(\mathbf{v_1}), TBL(v_2))) \text{ during an optimization phase}$$

As we have stressed in Note 4.2.1 and explained in Section 5.2, although the execution and sampling operations are represented as a join between

a sample table and a full table, the physical implementation of these two operations might be performed with different strategies, *e.g.* an index-based join. This means that the pre-materialization of the full table $TBL(v_2)$ is not a requirement to carry out the sampling and execution of the edge e.

6.1.6 Sampling an Edge With Two Executed Vertices

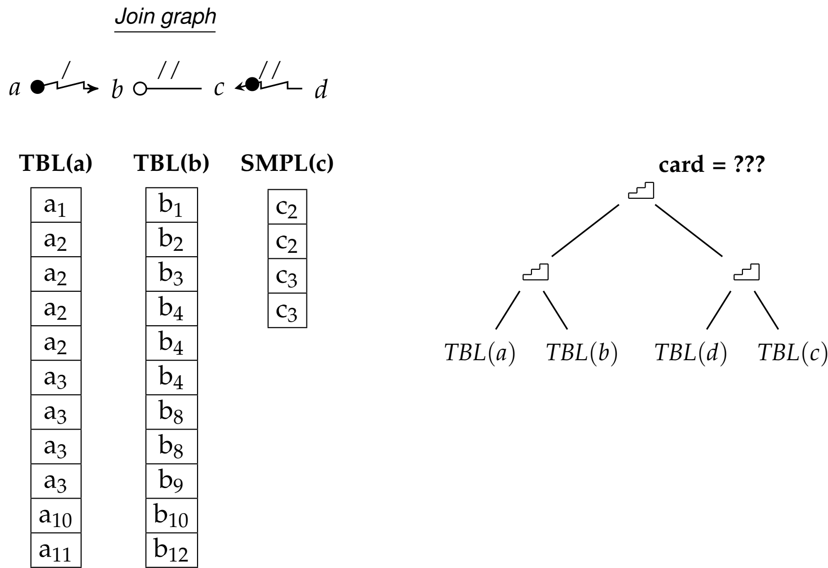
In Section 6.1.4, we have described the method used by ROX-sampled to execute an edge with two executed vertices. The same approach is used to *sample* an edge with two executed vertices.

Therefore, to sample an edge $e = (v, v')$ where v and v' are executed vertices, the following steps are applied:

- First the sizes of the two sets $edges^*(v)$ and $edges^*(v')$ are compared to determine the vertex to use as the left operand of the sampling operation (*i.e.* the vertex with the largest number of executed edges in its branches). Let v be the chosen vertex.
- The sample table of the vertex v is joined with the full table $T(v')$.
- All the edges in the set $edges^*(v')$ are re-sampled. Each of the sampling operations uses as input the output generated by the previous sampling operation.

The objective of sampling an edge e is to estimate the cardinality of the output of the operator associated to e. Therefore, while performing the above sampling operations, ROX-sampled keeps track of the join hit ratio of each of the sampled edges. Using the estimated hit ratios, ROX sampled can then compute the hit ratio of the whole set of sampled edges, and subsequently the cardinality of the edge e using the technique described in Section 3.2.2.

Example 6.1.6. Figure 6.9a depicts a join graph in which edges (a, b) and (c, d) are executed. The full tables corresponding to the vertices a and b, and the sample table associated to c are presented. Suppose that the current optimization phase of ROX-sampled needs to sample the edge (b, c) to estimate its weight. Figure 6.9b shows the join tree, the result size of which is to be estimated by ROX-sampled. Two operators have already been executed: ($TBL(a) \bowtie TBL(b)$) and ($TBL(d) \bowtie TBL(c)$), and ROX-sampled needs to estimate the cardinality of the expression $((T(a) \bowtie T(b)) \bowtie (T(d) \bowtie T(c)))$. Since edge (b, c) is an edge with two executed vertices, ROX-sampled first compares the size of the sets $edges^*(b)$ and $edges^*(c)$. Since both sets are equal in size, it does not matter which vertex is used as the right operand. ROX-sampled chooses the vertex b, and the edge (b, c) is sampled by matching the sample table $SMPL(c)$ with the full table $TBL(b)$. The result is shown in Figure 6.10a. The join hit ratio of the edge (b, c) is estimated to be 1. The join tree shown in Figure 6.10a depicts the plan, the result

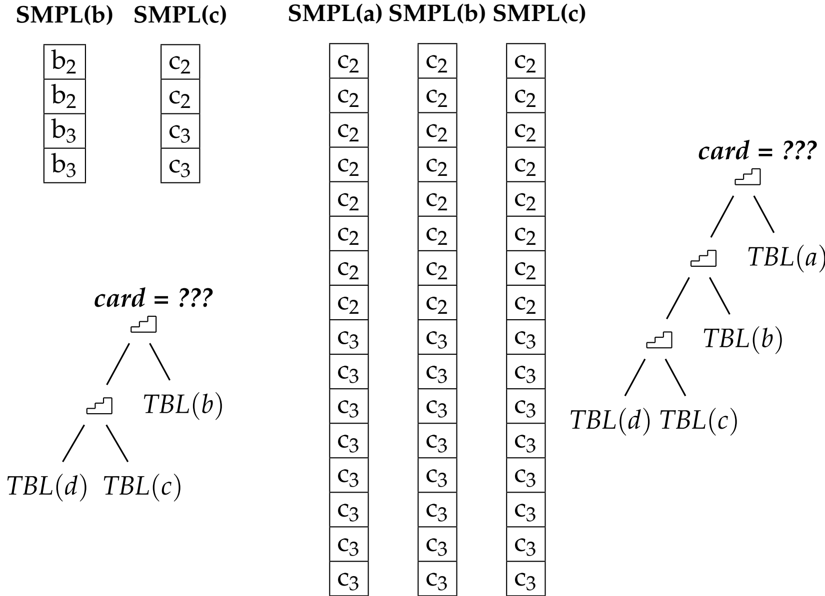


a Join graph where edges (a, b) and (c, d) are executed. The full tables associated to vertices a and b are shown, along with the sample table of vertex c.

b The join tree depicting the plan, the result size of which is to be estimated by ROX-sampled. Two step operators have already been executed, and ROX-sampled needs to estimate the cardinality of the step operator that matches the result of the two executed operators.

Figure 6.9 A join graph with 2 executed edges (a, b) and (c, d). ROX-sample needs to sample the edge (b, c) to estimate its cardinality. The join tree illustrates the plan, the result size of which is to be estimated by ROX-sampled.

size of which can now be computed using the estimated hit ratio. It can be noticed that the depicted plan does not include the previously executed edges branching from b, and therefore the estimated size does not take into account the fact that the edge (a, b) has already been executed. The next step of the sampling process is to re-sample the edges in $edges^*(b)$. The edge (a, b) is re-sampled by matching the sample table $SMPL(b)$ with the full table $TBL(a)$. Figure 6.10b shows the result of the sampling, and the join tree, the result size of which can now be estimated by ROX-sampled. The executed edge (a, b) is now included in the plan. In fact, this plan is equivalent to the join tree presented in Figure 6.9b. The result size of the sampling is equal to 16, and therefore the join hit ratio is estimated to be 4. This example shows the importance of re-sampling the edges in $edges^*(b)$. By doing so, ROX-sampled estimates the size of the edge (b, c) while taking



a The result of sampling (b, c) by matching $SMPL(c)$ with $TBL(b)$. ROX-sampled can now estimate the cardinality of the depicted plan.

b Completing the sampling of edge (b, c) by resampling the edges in $edges^*(b)$. The result of the sampling is shown. Depicted is the plan, the result size of which can now be estimated by ROX-sampled, using the estimated hit ratio.

Figure 6.10 The sampling of edge (b, c). By re-sampling the edges in $edges^*(b)$, ROX-sampled estimates the size of the edge (b, c) while taking into account the previously executed edges branching from b.

into account the chains of previously executed edges branching from the vertex b.

In this section, we have introduced the ROX-sampled variant. ROX-sampled uses samples during both its optimization and execution phases. Decisions made during the optimization steps are recorded, shaping iteratively the final execution plan. Once the order of all edges in the join graph is optimized, the defined plan is executed on full tables. We have also described the types of tables associated with vertices, and the approach used by ROX-sampled to execute and sample edges in a join graph, including the special case of edges with two executed vertices. In the next section, we present the ROX-sampled algorithm giving a complete and detailed description of this new variant.

6.2 The ROX-sampled Algorithm

ROX-sampled, presented in Algorithm 11, follows the same main steps as ROX-full. The first phase initializes the join graph, while the second phase alternates between optimization and execution steps. Optimization initiates, through the chain sampling technique, a search for the superior path to execute, while execution executes the chosen sequence of operators. Next, we describe each of these phases in detail.

Phase 1 (Algorithm 11: lines 1-11)

The first part of Phase 1 acquires knowledge about the vertices in the join graph: it defines the tables associated to a vertex, and estimates the number of XML nodes corresponding to the vertex. For a vertex v, the corresponding index is sampled to build its execution-sample table $ESMPL(v)$ of size τ' **(line 3, line 7)**. The index is also used to estimate the cardinality $card(v)$ of XML nodes corresponding to v **(line 4, line 8)**. Next, the sample table $SMPL(v)$ is initialized to a sample of size τ randomly chosen from the execution-sample table $ESMPL(v)$ **(line 5, line 9)**. In principle, the above operations are performed for all kinds of vertices as long as a sampling technique that satisfies the zero-investment property is available. In the ROX-sampled prototype and due to the supported types of indexes, only vertices that represent either an XML element with a given qualified name q **(line 2)** or a text node with an equality condition **(line 6)** are considered in this process.

The second part of Phase 1 computes the weight of the edges in the graph using the WEIGHT function presented in Section 6.2.1 **(lines 10-11)**. Again, analogously to ROX-full, a weight cannot be computed for all the edges in the graph. Since the weight of an edge is estimated through sampling, only those edges with at least one vertex that has a sample table that has been initialized in the previous step will be assigned a weight.

Phase 2 (Algorithm 11: lines 13-23)

The second phase of ROX-sampled is the core of the algorithm, and consists of interleaving optimization and execution steps, until all edges in the join graph are executed (in other words, until every edge is assigned an execution order). Recall that the execution of an edge e in ROX-sampled corresponds to sampling the edge e. In every optimization phase, the following steps are made:

- Pick the unexecuted edge $e = (v_1, v_2)$ in the join graph that has the smallest weight **(line 15)**.
- If at least one of the vertices of the edge e has more than one unexecuted edge **(line 16)**, then the chain sampling process is initi-

Algorithm 11: ROX-sampled: ROX for Pipelined Systems

INPUT : Join Graph $G = (V, E)$, Int τ, Int τ'

// $\tau =$ size of sample tables and limit of cutoff-sampling operations used in optimization phases, $\tau' =$ size of execution-sample tables and limit of cutoff-sampling operations used in execution phases

// Initialization phase

1 FOREACH $v \in V$ DO
2 IF v is an element type with qualified name q THEN
3 $ESMPL(v) \leftarrow \rhd_{\tau'}(\overset{D_{elt}}{\bigtriangledown}(q));$
4 $card(v) \leftarrow EstCard\left(\overset{D_{elt}}{\bigtriangledown}(q)\right);$
5 $SMPL(v) \leftarrow \rhd_{\tau}(ESMPL(v));$
6 ELSE IF v is a text node with predicate "$= x$" THEN
7 $ESMPL(v) \leftarrow \unrhd_{\tau'}(\overset{D_{text}}{\bigtriangledown}(x));$
8 $card(v) \leftarrow EstCard\left(\overset{D_{text}}{\bigtriangledown}(x)\right);$
9 $SMPL(v) \leftarrow \rhd_{\tau}(ESMPL(v));$

10 FOREACH $e = (v_1, v_2) \in E \mid SMPL(v_1) \neq \texttt{NULL} \vee SMPL(v_2) \neq \texttt{NULL}$ DO
11 $w(e) \leftarrow$ Weight(e);

12 Int $order \leftarrow 1$;

// Core phase: alternation of optimization and execution

13 WHILE $\exists$ more edges to execute DO
14 Path p ; // $p =$ sequence of operators to be executed
15 Edge $e = (v_1, v_2) \mid e \in E \wedge w(e) = \min_{e_i \in E} w(e_i);$
16 IF $|edges^-(v_1)| > 1 \vee |edges^-(v_2)| > 1$ THEN
17 $p \leftarrow$ ChainSample(e);
18 ELSE
19 $p \leftarrow \{e\}$;
20 FOREACH Edge $e \in p$ DO
21 SetExecOrder$(e, order)$;
22 $order \leftarrow order + 1$;
23 ExecPath&UpdateJG(p);
24 ExecutePlan();

Algorithm 12: WEIGHT

```
INPUT  : Edge e = (v, v')
OUTPUT: Int weight

   // Determine the sampling direction of e
1  Vertex List (lopd, ropd) ← SAMPLINGDIRECTION(e);

   // Cutoff-sampling of e
2  (S, hr) ← ⊵τ(op(e, SMPL(lopd), TBL(ropd)));

   // Sample chains of executed edges branching from ropd
3  Double branch_hr ← 1 ;
4  IF |edges+(ropd)| > 0 THEN
5  |  branch_hr ← SAMPLEEXECUTEDEDGES(ropd, {}, branch_hr, τ * 2);

   // Compute the hit ratio of all sampled edges including
      the edge e, then estimate the weight of e
6  branch_hr ← branch_hr × hr ;
7  Int weight ← card(lopd) × branch_hr;

8  RETURN weight;
```

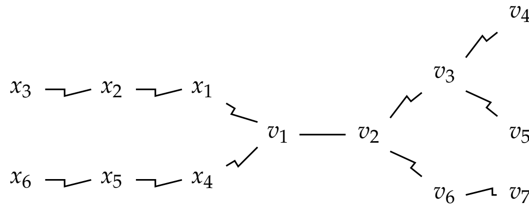
ated **(line 17)**. The *ChainSample* function, explained in Section 6.2.4 searches for the superior path p and returns it for execution.

- If the two vertices of the edge e have at most one unexecuted outgoing edge each, then chain sampling will not take place, and the path p to be executed consists of only edge e **(line 19)**.

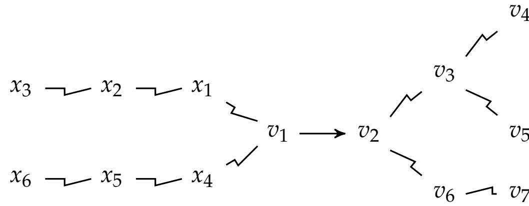
- Every edge in the chosen path p is assigned an execution order **(lines 20-22)**. The execution order of the edges defines the final execution plan chosen by ROX. Then all edges in p are executed and the knowledge in the join graph is updated using the function EXECPATH&UPDATEJG described in Section 6.2.5 **(line 23)**

When all edges in the join graph are ordered, then the defined final execution plan is executed using full tables as input **(line 24)**.

We have given a global description of the ROX-sampled algorithm. The only difference we have seen so far between ROX-full and ROX-sampled is in the definition of the tables associated with vertices. In the following, we give a detailed explanation of the building blocks of the ROX-sampled algorithm.



a ROX-sampled needs to compute the weight of the edge $e = (v_1, v_2)$ in the join graph. We suppose that the SAMPLINGDIRECTION function chose v_1 to be the left operand of the sampling operation of the edge e.



$$card(v_1) = 20000$$
$$|SMPL(v_1)| = 100$$

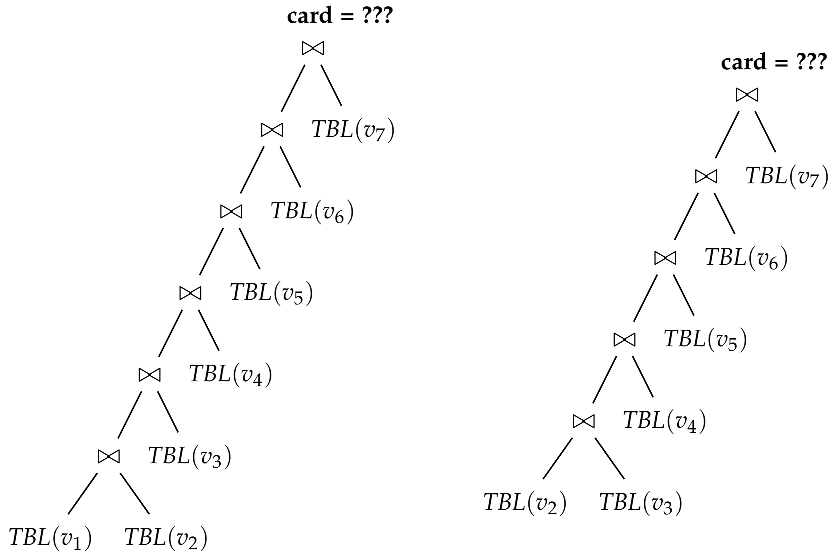
b The join graph illustrating the sampling of the edge $e = (v_1, v_2)$. The arrow on the edge denotes the sampling direction. Since v_2, the right operand of the sampling operation, is an executed vertex, the chains of executed edges branching from v_2 should be sampled.

Figure 6.11 An example join graph illustrating the WEIGHT algorithm.

6.2.1 The WEIGHT Function

Given as input an edge $e = (v, v')$, the WEIGHT function, presented in Algorithm 12, computes the weight of the edge, by sampling the operator associated with e. First the sampling direction of the edge e is defined, *i.e.* the left and right operands (*lopd* and *ropd*) of the sampling operation are determined **(line 1)**. This is performed by the SAMPLINGDIRECTION function, presented in Section 6.2.2. Then the operator associated with the edge e is cutoff-sampled using the limit τ and as left input the sample table $SMPL(lopd)$ and as right operand the full table $TBL(ropd)$ **(line 2)**.

If the right operand *ropd* of the sampling operation is an executed vertex, then the chains of executed edges branching from *ropd* needs to be sampled to take these previously executed edges into account in the weight estimation process **(lines 4-5)**. The sampling of the executed edges is performed by the function SAMPLEEXECUTEDEDGES and is explained in Section 6.2.3. The SAMPLEEXECUTEDEDGES function returns the hit ratio of



a The sequence of operators that needs to be sampled and their hit ratio estimated to compute the weight of the edge $e = (v_1, v_2)$ of the join graph in Figure 6.11a.

b The sequence of operators sampled by the SampleExecutedEdges function. The function estimates and returns the join hit ratio of the shown plan.

Figure 6.12 The sequence of operators that needs to be sampled and their hit ratio estimated to compute the weight of the edge $e = (v_1, v_2)$ of the join graph in Figure 6.11a. The join $TBL(v_1) \bowtie TBL(v_2)$ is sampled and its hit ratio estimated by the Weight function, while the other joins are sampled and their hit ratio estimated by the SampleExecutedEdges function.

the set S of edges it has sampled, which is then used to compute, with the technique described in Section 3.2.1, the hit ratio of the set of edges $S \cup \{e\}$ **(line 6)**. Using the newly computed hit ratio *branch_hr*, the weight of the edge e is then estimated **(line 7)**.

Example 6.2.1. Figure 6.11a shows an example join graph in which the weight of the edge $e = (v_1, v_2)$ is to be computed. We suppose that the sampling direction of the edge e is determined to be (v_1, v_2), therefore the edge e is sampled using as left input the sample table $SMPL(v_1)$. The sampling process is depicted in Figure 6.11b where the arrow indicates the sampling direction. We suppose that the hit ratio hr of the edge e is estimated to be 2. To compute the weight of e, since v_2 is an executed vertex, ROX-sampled needs to estimate the result size of the plan depicted in Figure 6.12a. The join $TBL(v_1) \bowtie TBL(v_2)$ is sampled and its hit ratio estimated by the Weight algorithm as we have already seen, while all other

joins illustrated in the plan in Figure 6.12b are sampled and their hit ratio estimated by the SampleExecutedEdges function. Let the estimated hit ratio of the plan in Figure 6.12b be 3. The Weight function then computes the hit ratio of the sequence of operators in the plan of Figure 6.12a as $3 \times 2 = 6$ **(Algorithm 12: line 6)**. Knowing that $card(v_1) = 20000$, the weight of the edge e is finally estimated to be: $20000 \times 6 = 120000$.

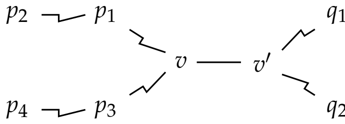
6.2.2 The SamplingDirection Function

Given an edge $e = (v, v')$ which is to be sampled, the SamplingDirection function, presented in Algorithm 13 determines, by checking the characteristics of the two vertices of e, the left and right operands of the sampling operation of e. The decision is based on two criteria. The first is the number of executed edges branching from each of the vertices. If no choice can be made using the first criterion, then the decision is derived based on the size of the execution-sample table associated with each vertex. The algorithm examines several situations listed below:

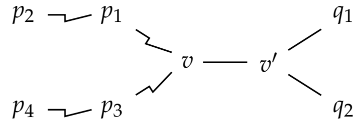
- **Case 1:** $|edges^*(v)| \neq |edges^*(v')|$ - **(lines 1-5)**.
- **Case 2:** $|edges^*(v)| = |edges^*(v')| \land SMPL(v) \neq \texttt{NULL} \land SMPL(v') \neq \texttt{NULL}$ - **(lines 6-10)**.
- **Case 3:** $|edges^*(v)| = |edges^*(v')| \land (SMPL(v) \neq \texttt{NULL} \oplus SMPL(v') \neq \texttt{NULL})$ - **(lines 11-14)**. The $\oplus$ symbol represents the exclusive or.

Example 6.2.2. To describe the above cases, we use the example join graphs shown in Figure 6.13. In each join graph in the figure, we suppose that the samplig direction of the edge $e = (v, v')$ is to be determined by the function SamplingDirection.

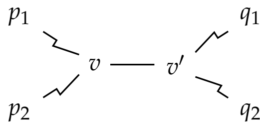
- **Case 1 (Figure 6.13a and Figure 6.13b):** In this case, at least one of the two vertices of e is an executed vertex. The decision about the sampling direction in this situation is based on reducing the number of executed edges to be sampled. We already know that when sampling an edge, if the right operand of the sampling operation is an executed vertex, the chains of executed edges branching from the vertex should be sampled. Therefore, the function SamplingDirection picks the vertex with the smallest number of branching executed edges to be the right operand. In Figure 6.13a and Figure 6.13b, we have $|edges^*(v)| > |edges^*(v')|$, therefore v' is chosen to be the right operand **(lines 1-5)**.
- **Case 2 (Figure 6.13c and Figure 6.13d):** In this case, the two vertices v and v' have the same number of executed edges in their branches ($|edges^*(v)| = |edges^*(v')|$), therefore the criterion here is not about reducing the number of executed edges which ROX-sampled has to sample. Since both vertices are initialized ($SMPL(v) \neq \texttt{NULL}$ and



a Case 1. $|edges^*(v)| = 4$ and $|edges^*(v')| = 2$. The vertex v' is chosen to be the right operand of the sampling operation to reduce the number of executed edges which ROX-sampled should sample.

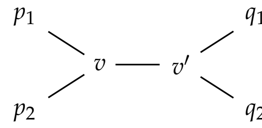


b Case 1. $|edges^*(v)| = 4$ and $|edges^*(v')| = 0$. The vertex v' is chosen to be the right operand of the sampling operation to reduce the number of executed edges which ROX-sampled should sample.



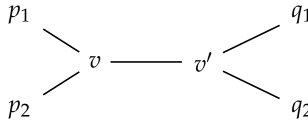
$$|ESMPL(v)| = 200$$
$$|ESMPL(v')| = 450$$

c Case 2. $|edges^*(v)| = |edges^*(v')| = 2$, $SMPL(v) \neq \texttt{NULL}$, and $SMPL(v') \neq \texttt{NULL}$. To get accurate estimation from the sampling operation, v is chosen to be the left operand. In fact, v has a smaller execution-sample table, therefore its sample table $SMPL(v)$ randomly chosen from $ESMPL(v)$, will be more representative.



$$|ESMPL(v)| = 150$$
$$|ESMPL(v')| = 400$$

d Case 2. $|edges^*(v)| = |edges^*(v')| = 0$, $SMPL(v) \neq \texttt{NULL}$, and $SMPL(v') \neq \texttt{NULL}$. To get accurate estimation from the sampling operation, v is chosen to be the left operand. In fact, v has a smaller execution-sample table, therefore its sample table $SMPL(v)$ randomly chosen from $ESMPL(v)$, will be more representative.



$$|SMPL(v)| \neq \texttt{NULL}$$
$$|SMPL(v')| = \texttt{NULL}$$

e Case 3. $|edges^*(v)| = |edges^*(v')| = 0$, $SMPL(v) \neq \texttt{NULL}$, and $SMPL(v') = \texttt{NULL}$. Since the sample table of vertex v' is not initialized yet, the only possibility is to choose v to be the left operand of the sampling operation.

Figure 6.13 Join graph examples illustrating different cases examined by the function SAMPLINGDIRECTION. In each join graph, the sampling direction of the edge $e = (v, v')$ is to be determined.

Algorithm 13: SamplingDirection

INPUT : Edge $e = (v, v')$
OUTPUT: Vertex List $(lopd, ropd)$

```
1  IF |edges*(v)| ≠ |edges*(v')| THEN
2  |  IF |edges*(v)| > |edges*(v')| THEN
3  |  |  RETURN {v, v'};
4  |  ELSE
5  |  |  RETURN {v', v};
   |
6  IF SMPL(v) ≠ NULL ∧ SMPL(v') ≠ NULL THEN
7  |  IF |ESMPL(v)| ≤ |ESMPL(v')| THEN
8  |  |  RETURN {v, v'};
9  |  ELSE
10 |  |  RETURN {v', v};
   |
11 ELSE IF SMPL(v) ≠ NULL THEN
12 |  RETURN {v, v'};
13 ELSE
14 |  RETURN {v', v};
```

$SMPL(v') \neq$ NULL), the algorithm compares the size of the execution-sample table of the two vertices and chooses the vertex with the smallest cardinality to be the left operand **(lines 6-10)**. The reasoning behind this choice is that the smaller the set from which a sample is picked, the more representative the chosen sample, and hence the more accurate the result of the sampling operation.

- **Case 3 (Figure 6.13e):** Similarly to Case 2, the two vertices v and v' have the same number of executed edges in their branches ($|edges^*(v)| = |edges^*(v')|$). But in this case, only the tables of the vertex v have been initialized. Hence, the vertex v' cannot be used as left operand since its execution-sample and sample tables have not been defined yet. Therefore, the vertex v is chosen to be the left operand of the sampling operation **(lines 6-10)**.

We now summarize our description of the *SamplingDirection* function. When deciding about the sampling direction of an edge, the following two points are taken into account:

Algorithm 14: SampleExecutedEdges

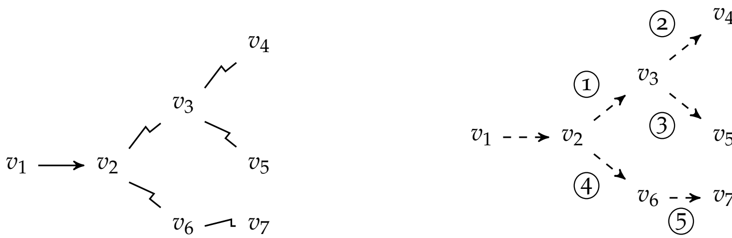
```
INPUT  : Vertex v, List sampled_edges, Double branch_hr, Int LIMIT
// v = the vertex of which the chains of executed edges
   needs to be sampled, sampled_edges = list of executed
   edges sampled so far, branch_hr = the hit ratio of all
   the executed edges sampled so far, LIMIT = limit for
   cutoff-sampling
OUTPUT: Double branch_hr

1 FOREACH edge e = (v, v') ∈ (edges+(v) \ sampled_edges) DO
2     (SMPL(v'), hr) ← ⊵_LIMIT(op(e, SMPL(v), TBL(v')));
3     sampled_edges.INSERT(e);
4     branch_hr = branch_hr × hr;
5     LIMIT ← LIMIT + τ ;
6     IF |edges+(v') \ sampled_edges| > 0 THEN
7         branch_hr ←
          SAMPLEEXECUTEDEDGES(v', sampled_edges, branch_hr, LIMIT);
8 RETURN branch_hr;
```

1. **First priority:** Reduce the number of executed edges to be sampled to keep the sampling cost small.
2. **Second priority:** Use a more representative sample table as left input.

6.2.3 SampleExecutedEdges Function

Given a vertex v, the function SampleExecutedEdges, presented in Algorithm 14, samples the chains of executed edges branching from v, and estimates their hit ratio. The sampling is performed in a depth-first manner: first an executed edge $e = (v, v')$ outgoing of v is sampled using as left input the sample table $SMPL(v)$ **(lines 1-2)**, then if the vertex v', the right operand of the sampled edge e, is an executed vertex, the executed edges of v' are also sampled. This is done by calling the algorithm recursively with the vertex v' as argument **(lines 6-7)**. Note that in the recursive function call, the value of the passed cutoff limit is increased by τ **(line 5)**. This allows for the generation of results with larger sizes, which reduces the chance of estimation errors being propagated through the sampling process, hence making the estimated cardinalities more accurate. We note that the cutoff limit can also be increased with a value other than τ; however, we have not tested which value would be best to use.

The hit ratio hr estimated by the cutoff-sampling of the edge e is used



a A join graph in which the edge $e = (v_1, v_2)$ is sampled to compute its weight. The other edges of v_1 have been ommitted from the figure for illustration reasons. The edge e is sampled with v_1 as left operand. Since v_2, the right operand of the sampling operation, is an executed vertex, the chains of executed edges branching from v_2 are sampled.

b The join graph illustrating the sampling of the chains of executed edges branching from v_2. The number on the edges denotes the order in which the edges are sampled, and the arrows indicate the sampling direction. The edges are sampled in a depth-first manner, with each sampling step using as input the output result of the previous sampling operation.

Figure 6.14 An example join graph illustrating the sampling process of the function SampleExecutedEdges.

to derive the hit ratio *branch_hr* of all the edges sampled so far **(line 4)**. The computation of *branch_hr* is performed using the method described in Section 3.2.1. Note that the hit ratio *branch_hr* is passed to every call of the SampleExecutedEdges function.

In join graphs with cycles, it is possible to encounter the same edge several times through different routes. In order not to sample the same edge multiple times, the algorithm keeps track of the edges that are sampled through the list *sampled_edges*. Every edge sampled by the function is inserted in the list *sampled_edges* **(line 3)** which is also passed to every call of the SampleExecutedEdges function.

Continuation of Example 6.2.1 We reconsider the example join graph presented in Figure 6.11. Recall that the edge $e = (v_1, v_2)$ has already been sampled by the Weight function as part of estimating the weight of e. The current situation is depicted in Figure 6.14a. We omit some of the edges of the vertex v_1 from the figure for illustration purposes. Since v_2, the right operand of the sampling operation, is an executed vertex, the chains of executed edges branching from v_2 are also sampled using the SampleExecutedEdges function. The sampling process is illustrated in Figure 6.14b. The number on the edges denotes the order of sampling, and the arrows represent the sampling direction. The edges are sampled recursively, in a depth-first manner. Figure 6.15 lists the sampled edges, their hit ratio *hr* estimated through cutoff-sampling, and the derived hit ratio *branch_hr* of the collection of edges sampled so far.

Sampled Edge	*hr*	*branch_hr*
Before sampling starts		1
(v_2, v_3)	2	2
(v_3, v_4)	0.5	1
(v_3, v_5)	3	3
(v_2, v_6)	2	6
(v_6, v_7)	0.5	3

Figure 6.15 The estimation of the hit ratio of the sampled chains of edges branching from the vertex v_2 in the join graph of Figure 6.14b. The table presents each sampled edge, its estimated hit ratio *hr*, and the derived hit ratio of the set of edges sampled so far *branch_hr*. The data in the first row corresponds to the value of the *branch_hr* argument passed to the SampleExecutedEdges function.

6.2.4 The ChainSample Function

The ChainSample function of ROX-sampled follows the same steps as its counterpart in the ROX-full algorithm. The few differences existing between the two are explained in this section.

The first difference is in the method employed to pick the vertex to use as the starting point of the chain sampling exploration. As we have seen in Section 6.2.2, the highest priority when picking the sampling direction of a to-be-sampled edge is to reduce the number of executed edges that needs to be sampled to keep the sampling overhead limited. Therefore, line 1 in Algorithm 5 is replaced with the following statement:

Vertex *start_vertex* $\leftarrow$ SamplingDirection(e);

The second difference occurs when chain sampling cutoff-samples an edge $e' = (v, v')$. In ROX-sampled, whenever an edge is sampled, the algorithm needs to check if the right operand of the sampling operation is an executed vertex and if so the chains of executed edges of the vertex should be sampled. Therefore, the following statements are added after line 16 in Algorithm 5:

```
Double branch_hr ← 1;
// If v' is an executed vertex, sample the chains of executed edges
    branching from v'
IF (|edges⁺(v')| > 0)
  branch_hr ← SampleExecutedEdges(v', {}, branch_hr, LIMIT + τ);
```

// Compute the hit ratio of all sampled edges including the edge e'
$hr \leftarrow branch_hr \times hr$;

The above piece of code is similar to the one in Algorithm 12 between lines 3 and 6, therefore for a detailed explanation, we refer the reader back to Section 6.2.1.

6.2.5 The EXECPATH&UPDATEJG Function

After picking the edge with the smallest weight, the ROX-sampled algorithm either initiates a chain sampling process to search for a superior path and return it for execution, or executes the path consisting of the singleton edge e. The function responsible for executing a given path p and then updating the knowledge in the join graph using the new up-to-date generated results is EXECPATH&UPDATEJG presented in Algorithm 15. We again remind the user that execution here refers to a sampling operation using an execution-sample table as left input.

The EXECPATH&UPDATEJG algorithm in ROX-sampled is similar to its counterpart in ROX-full. First, for each edge e in p, the execution direction of e is determined. Since execution here is nothing more than a sampling operation, the SAMPLINGDIRECTION function is used **(line 2)**. The edge e is then executed by cutoff-sampling with limit τ' the join between the execution-sample $ESMPL(lopd)$ and the full table $TBL(ropd)$ **(line 3)**.

As previously explained in Section 6.1.4, if the right operand *ropd* of the execution operation of an edge is an executed vertex, then the chains of previously executed edges branching from *ropd* should be re-executed. This is performed with the REEXECUTEEDGES function **(lines 4-6)**, which is described in Section 6.2.6. Additionally, the REEXECUTEEDGES function estimates and returns the hit ratio of all re-executed edges **(line 6)**, which is then used to compute the hit ratio of the set of re-executed edges along with the edge e **(line 7)**.

After executing the edge e, the knowledge in the join graph is updated using the new data in the execution-sample tables **(lines 8-12)**. First, for each vertex v of e, the execution-sample table of v is sampled to create a new sample table corresponding to v **(line 9)**. Then the estimated hit ratio of the executed edge e and potentially of those re-executed edges is used to compute an estimation of the cardinality of v **(line 10)**. Finally, for every unexecuted edge of v, the weight of the edge is (re-)computed to derive a new weight using the more accurate samples **(lines 11-12)**.

Algorithm 15: EXECPATH&UPDATEJG

```
INPUT : Path p
1  FOREACH edge e ∈ p DO
       // Determine execution direction of e
2      Vertex List (lopd, ropd) = SAMPLINGDIRECTION(e);
       // Execute e
3      (S, hr) ← ⊵_τ'(op(e, ESMPL(lopd), TBL(ropd)));
       // If ropd is an executed vertex, re-execute the chains
          of executed edges branching from ropd
4      Double branch_hr ← 1 ;
5      IF |edges⁺(ropd)| > 0 THEN
6          branch_hr ← REEXECUTEEDGES(ropd, {e}, branch_hr, τ' × 2) ;
           // Compute the hit ratio of all the re-executed
              edges and the edge e
7          branch_hr ← branch_hr × hr;
       // Update knowledge in join graph
8      FOREACH v ∈ {v1, v2} DO
9          SMPL(v) ← ⊳_τ(ESMPL(v));
10         card(v) ← card(lopd) × branch_hr;
11         FOREACH e ∈ edges⁻(v) DO
12             w(e) ← WEIGHT(e);
```

6.2.6 The REEXECUTEEDGES Function

The REEXECUTEEDGES function is similar to the SAMPLEEXECUTEDEDGES function. Given a vertex v, the two algorithms re-execute/sample the chains of previously executed edges branching from v and estimate the hit ratio of the set of re-executed/sampled edges. The main difference between the two is that the execution operation uses a different cutoff limit τ' instead of τ, and an execution-sample table instead of a sample table as its left input. Therefore, the REEXECUTEEDGES algorithm is the same as the SAMPLEEXECUTEDEDGES algorithm, presented in Section 6.2.3 (Algorithm 14), modulo the following three modifications:

- Line 2 in Algorithm 14 is replaced with:

$$(ESMPL(v'), hr) \leftarrow \trianglerighteq_{\text{LIMIT}}(op(e, ESMPL(v), TBL(v')));$$

- Line 5 in Algorithm 14 is replaced with:

  $\texttt{LIMIT} \leftarrow \texttt{LIMIT} + \tau'$;

- Line 7 in Algorithm 14 is replaced with:

  $branch_hr \leftarrow \textsc{ReExecuteEdges}(v', sampled_edges, branch_hr, \texttt{LIMIT})$;

In the previous sections, we have described in detail the algorithm for ROX-sampled, our run-time optimizer targeting systems with a pipelined execution strategy. In the next section, we present the experiments we have conducted to assess the robustness of ROX-sampled in general and in comparison to ROX-full.

6.3 Experiments

The main difference between ROX-sampled and ROX-full is that the first uses samples throughout the whole algorithm while the latter alternates between the use of sample tables during the optimization phases and full tables during the execution steps. One of the reasons behind the robustness of ROX-full is that the optimization decisions are made using up-to-date, newly materialized data that is generated during previous execution steps. By basing, in ROX-sampled, the decisions about the execution order of the operators in the join graph on only sampled data, we run the risk of jeopardizing the quality of decisions made by the new ROX variant and consequently its robustness. Therefore, the target of the experiments we present in this section is to assess the robustness and quality of decisions made by ROX-sampled by comparing them with the ROX-full decisions. We also expand our assessment to include a comparison with a classical compile-time optimizer. We finally investigate the impact of the cutoff limit value on the performance of ROX-sampled.

6.3.1 Experiments Setup

Similar to ROX-full, a prototype of the ROX-sampled algorithm has been implemented on top of MonetDB. We stress that the objective of this chapter is to propose a variant of ROX-full that is suitable for pipelined database systems, and to test and compare the quality of plans generated by both the new variant and ROX-full. Therefore, we refrain from implementing the ROX-sampled algorithm in a database system with a pipelined execution strategy, and direct and focus our experiments on assessing the robustness and quality of our approach within the same database system.

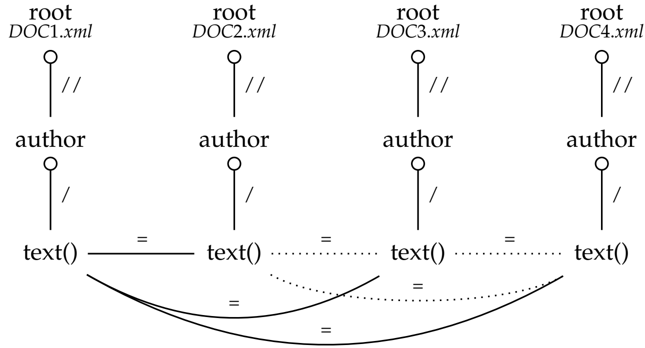


Figure 6.16 Join graph corresponding to the DBLP query template. The solid lines correspond to the edges in the join graph generated by the Pathfinder XQuery compiler. The dotted lines are added by ROX and represent join equivalences implied from the three equality joins defined in the query template.

The experiments use the same DBLP XML dataset[1] and XQuery template used for the ROX-full experiments presented in Section 5.5. We give here a brief review of the experiment setup, and refer the reader back to Section 5.5 for a detailed description.

The DBLP document is divided into ∼4500 single XML documents, one for each journal or conference. The different documents are then clustered based on their research area. The used XQuery template, presented below, asks for authors that have published in four 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 corresponding to the above query is illustrated in Figure 6.16. By replacing the 4 documents in the XQuery template 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.

[1] http://dblp.uni-trier.de/xml/

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 testing the 4500 documents and all 409515972723000 combinations of 4 documents will take unnecessarily a large amount of time without much impact on the experimental results, we select 23 "representative" documents from 5 research areas (Database, Data mining, Information retrieval, Bioinformatics, Artificial Intelligence), which results in 831 document combinations yielding non-empty results. The size of the documents extracted from the original DBLP document ranges from 300 B to 4.8 MB. ROX + MonetDB/XQuery evaluates all 831 queries in less than 50 milliseconds. To achieve more reliable performance measurements, we scale the complete DBLP dataset to 45 GB by replicating each article 100 times. 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 $[0, \ldots, 100]$.

6.3.2 Execution Order of Operators

In our first experiment, we investigate whether, given a join graph, ROX-full and ROX-sampled choose the same execution order for the operators in the graph.

We run the two ROX variants using the 831 document combinations, and compare the chosen execution order of operators. The size of sample tables, execution-sample tables, and the cutoff limit τ used in sampling operations during the optimization phases of both ROX-full and ROX-sampled is varied along the values $100, 500$, and 1000. We refer to the aforementioned values as "*sample size*". In this experiment, the sampling operations of the execution phases of ROX-sampled are performed without a cutoff limit, *i.e.* all tuples in the sample input are consumed by the sampling operation. The reason behind this choice is to experiment with ROX-sampled without limiting the generation of results during its execution phases. We note that a subsequent experiment (Section 6.3.4) tests whether introducing the cutoff limit jeopardizes the quality of the produced plans, and it proved that when using a large enough cutoff limit value the quality of plans remains the same.

Figure 6.17 shows the percentage of the 831 queries optimized to the same plan by the two ROX variants. With a *sample size* equal to 100, only 20% of the generated plans are the same. This number increases to 48% when a *sample size* of 1000 is used. The figure also presents the percentage of

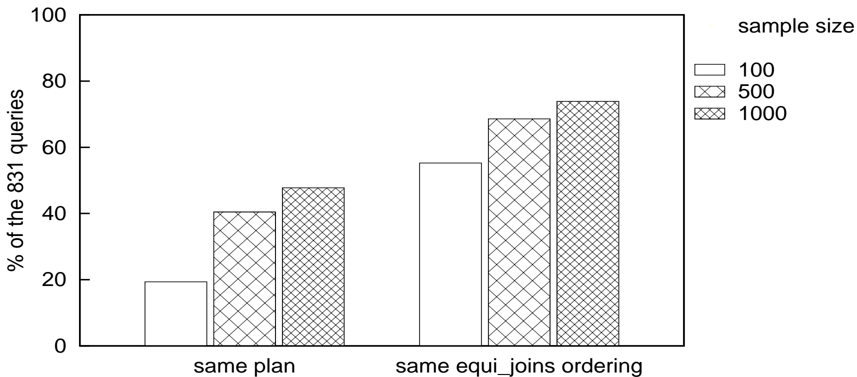


Figure 6.17 Percentage of the 831 queries optimized to the same plan by both ROX-full and ROX-sampled, and percentage of those queries for which the plans generated by the two variants share the same equi-join ordering but a different step placement among the equi-joins.

plans generated by the two variants that share the same equi-joins ordering but that differ in the placements of the XPath steps (`author/text()`) among the equi-joins. This is of interest since the correlations between the 4 queried documents is detected by correctly estimating the result 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. Noticeably, the two ROX variants, in their search for the best execution plan, manage to similarly order the equi-joins more often than generating the same plan. The percentage of plans with the same equi-joins ordering 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 two 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.

6.3.3 Execution Time of Plans

In the previous experiment, we compared the ordering of operators in the plans generated by the ROX variants. We noticed that with a larger *sample size*, the gap between ROX-sampled and ROX-full becomes smaller. Since the two variants do not always generate the same plan, the next point we investigate is whether the quality of a ROX-sampled plan is considerably worse than its counterpart ROX-full plan. Therefore, this second experiment compares the execution time of the plans generated by the two ROX variants.

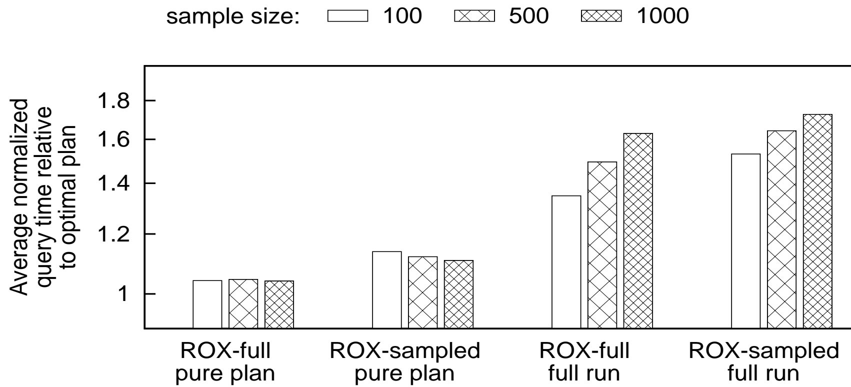


Figure 6.18 Average normalized execution time of the pure plan and full run plan generated by the two ROX variants for the 831 document combinations using different values of *sample size*. The normalization is relative to the fastest of the four considered execution plans.

We run ROX-full and ROX-sampled on the 831 document combinations, varying the *sample size* as explained in Section 6.3.2. For each of the two ROX variants, we consider and record the execution time of 2 plans: the pure plan and the full run. Pure plan represents the generated plan excluding the sampling cost, while the full run stands for the chosen plan including the sampling overhead. The sampling overhead in the full run plan of ROX-sampled includes the cost of the sampling operations performed during both the optimization and execution phases. Figure 6.18 shows the average execution time of the 4 considered plans, normalized to the plan with the smallest recorded time. The execution time of the pure plan of ROX-sampled decreases when a bigger *sample size* is used, while ROX-full is almost insensitive to the *sample size*. With a *sample size* of 100, the execution time of ROX-sampled is on average 9% slower than ROX-full, and it decreases to 6% when a *sample size* of 1000 is used. This indicates that even though more than 50% of the 831 queries are optimized to different plans by the two ROX variants (Figure 6.17), the quality of the chosen plans is comparable. The execution time of the full run plans of both variants increases when a larger sample is used, which is to be expected since more data is joined during the optimization phases.

We note that the sampling overhead in ROX-sampled is higher than in ROX-full. This is caused by the fact that in ROX-sampled all operations performed not only during the optimization phases but also during the execution phases and while re-executing and sampling previously executed edges contribute to the optimization cost. The sampling overhead in ROX-sampled is on average only 6% higher, which leads us to conclude that the

overhead created by the sampling performed during the execution phases, and by the redundant work when re-executing and sampling previously executed edges is limited.

Figure 6.19 shows the normalized execution times of the four plans generated for each of the 831 document combinations, while using a *sample size* of 1000. We also assume in this experiment a *classical* compile time optimizer, similar to the one described in Section 5.5.2. The classical optimizer is able to accurately estimate the cardinality of operations carried on a single document, but lacks the ability to estimate the correlations existing between two or more documents. This results in an order of the equi-joins that reflects a *smallest-input-first* heuristic where the two smallest `author` sets are joined first, the result is then joined with the third smallest `author` input, and the remaining `author` set is finally joined with the generated output. For the plan exhibiting this *smallest-input-first* equi-join ordering, we consider three canonical step placements among the equi-joins: SJ, JS, and S_J. The first consists of executing the steps of all 4 documents before the joins in the same order of the joins execution. The second corresponds to first executing one step to provide the initial input for the join sequence, then all joins are evaluated, and the remaining 3 steps are executed last. The last canonical step placement consists of first executing the initial step and join, then a step corresponding to a certain document is executed right after the document has been joined to the already generated intermediate result. The normalized execution time of the fastest of the SJ, JS, and S_J plans is plotted.

The plot in Figure 6.19 shows that the pure plan of ROX-full is the fastest almost all the time. The performance of ROX-sampled is closely similar to ROX-full, less than two times slower for most queries. However, for few queries, it can be 3 to 4 times slower. The sampling overhead is on average around 30% of the execution time of the full run plan, with a barely noticeable small difference between the overhead generated in ROX-full and ROX-sampled. The two ROX variants are robust and insensitive to the different correlations, their performance is stable across and within the three clusters 2:2, 3:1, and 4:0, while the classical optimizer shows strong variations, sometimes picking plans that are 2 orders of magnitude slower than the ROX plans.

For few queries, ROX-full picks a plan that is 2 times slower than the fastest plotted plan. Similarly, ROX-sampled is in some cases 3 to 4 times slower than ROX-full. The reasons behind this degradation in the performance have been explained in detail in Section 5.5.8, and consist mainly of the front-bias nature of the cutoff-sampling implementation and of the use of non-representative starting sample sets while chain sampling. Potential solutions to these problems have been described in Section 5.5.8.

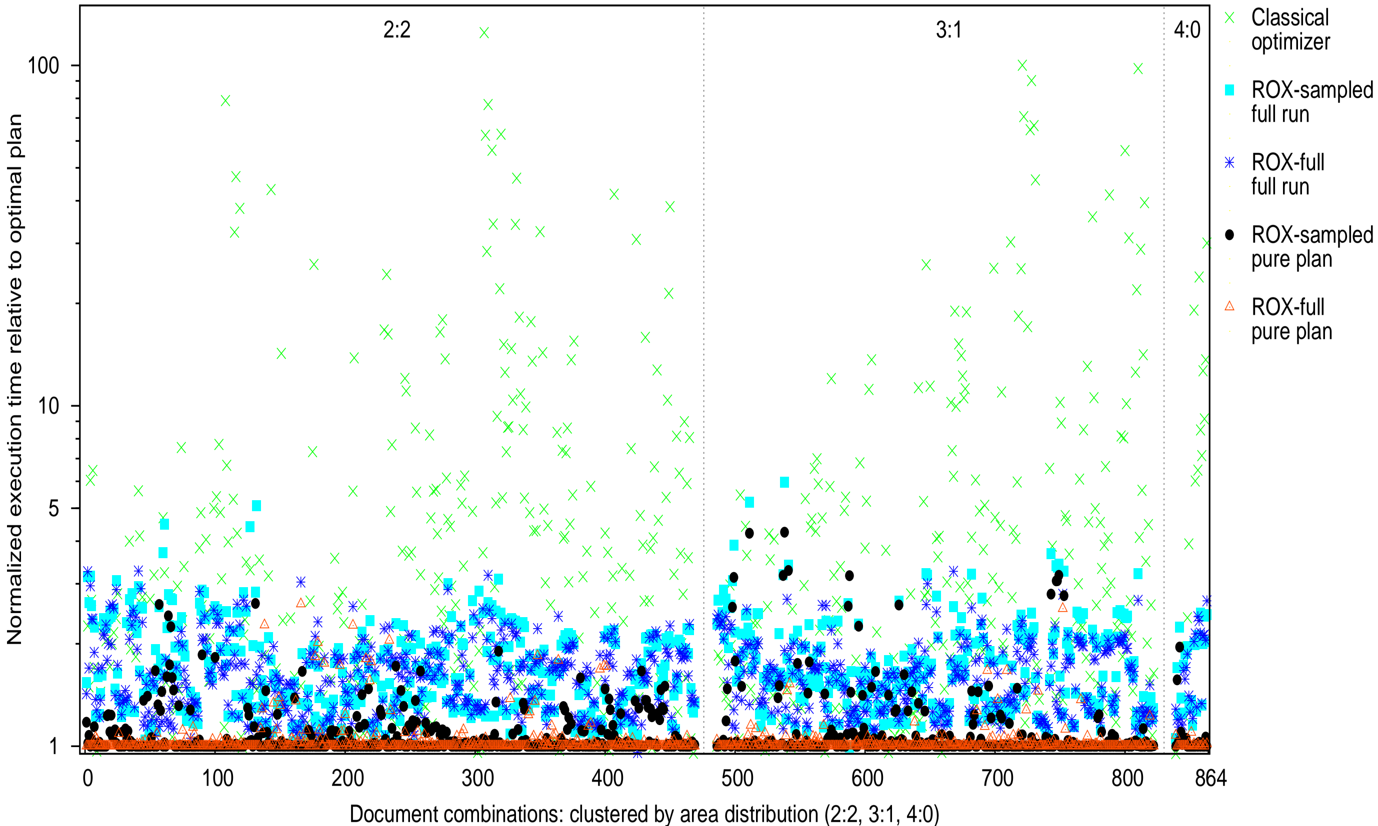


Figure 6.19 Normalized execution time of the pure plan and full run plan generated by the two ROX variants and the plan chosen by the assumed classical optimizer for the 831 document combinations. The *sample size* is set to the value 1000. The normalization is relative to the fastest of the five considered execution plans.

6.3.4 Impact of the Cutoff Limit

In our last experiment, we vary the cutoff limit used during the execution steps of ROX-sampled, to study its impact on the performance of ROX-sampled.

In the previous experiments, sampling during the execution steps of ROX-sampled was performed with an unlimited cutoff limit: all tuples in the sampled input were consumed by the sampling operation. Therefore, we wish to investigate whether using a specific cutoff value to limit the generation of intermediates tuples will jeopardize the quality of plans chosen by ROX-sampled. In this experiment, we reduce the value of the cutoff limit τ' used during the execution phases of ROX-sampled from unlimited to the double of the *sample size*. Analogously to the previous experiments, we vary the *sample size* along the values $100, 500$, and 1000.

We measure the elapsed time of the pure and full run plans of ROX-full, ROX-sampled, and ROX-sampled with cutoff, using the 831 document combinations. Figure 6.20 shows the noted execution times of the 6 plans, normalized to the fastest plan among the 6 considered ones. The symbols (-) and (+) denote, respectively, pure plans and full run plans. We notice that the use of a small cutoff limit τ' results in a small increase in the execution time of ROX-sampled, while the use of a cutoff limit of 2000 does not. This means that the quality of the chosen plan in case of a small τ' value is slightly worse. This is to be expected since with fewer data being generated during the execution phases of ROX-sampled, the chance of less representative samples used during the optimization steps is higher. However, a cutoff limit τ' of 2000 proves to be enough, which means that it is possible to replace an unlimited cutoff limit with an appropriate value without affecting the performance of ROX-sampled. Note that the overhead in the case of ROX-sampled with cutoff is higher than that of ROX-sampled when τ' takes the value 200 and 1000. This is because the pure plans themselves chosen by ROX-sampled are already slower.

In the previous experiments, we have focused on studying the performance of ROX-sampled, while using the DBLP data set. We also note that ROX-sampled has also been evaluated against XMark documents,[2] and has proven to be successful in picking a good execution order for the operators in the join graph. In particular, given as input the XQuery Q and its variant Q' presented in Section 5.4, ROX-sampled is capable of detecting the correlation and its changing effects between the different nodes in the query, and of exploiting it to determine a good execution order of the operators in the join graph. In fact, the plans chosen by ROX-sampled and ROX-full for Q and Q' are the same.

[2] `http://www.xml-benchmark.org/`

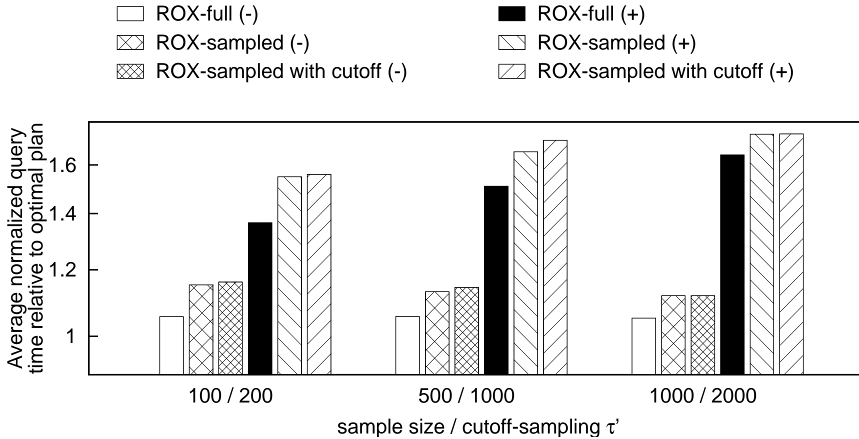


Figure 6.20 Average normalized execution time of the pure plan and full run plan generated by the ROX-full, ROX-sampled and ROX-sampled with cutoff for the 831 document combinations using different values of *sample size*. ROX-sampled with cutoff uses a cutoff limit τ' for the sampling operations performed during its execution phases, which is equal to twice the *sample size*. The normalization is relative to the fastest of the six considered execution plans.

6.4 Implementing ROX-sampled in Pipelined Database Systems

Now that we have explained the ROX-sampled approach, we briefly discuss the requirements to implement the ROX-sampled algorithm in database systems with a pipelined execution strategy. We focus on two aspects: constructing the sample and execution-sample tables associated with vertices in the join graph, and the techniques used to sample joins.

In the ROX-sampled algorithm, we have used index lookups to pick the initial samples of XML nodes corresponding to a given vertex. Another method that does not require the existence of indexes on every attribute is to have the sample tables pre-built and stored 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 stored. One difference between collecting statistics and storing samples for single attributes, is that correlations existing between two or more attributes can easily be derived by using the sampled data, a task that is not possible in the case of statistics, resulting in an attribute value independence being usually assumed. A good survey about techniques to select representative data samples from a table is given in [100].

Two types of edges exist in a join graph: XPath step edges and relational join edges. In the ROX-sampled prototype, XPath steps are sampled using

staircase join operators, while the sampling of relational-joins, which are restricted in the implementation to equi-joins, is performed using an index-based joins. Therefore, the implementation of ROX in a pipelined system requires, on the one hand, the support of a structural join that satisfies the zero-investment property, and, on the other hand, the existence of an index on one of the equi-joined attributes. The latter is not a realistic requirement and therefore we investigate some possible solutions.

Techniques that efficiently sample a relational join without the use of an index have been proposed in [29]; 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 to sample a relational join. But this means that hash tables must be built on both the input sample and the entire relation. The first operation is cheap; however, hashing an entire relation R is expensive, and does not conform to the zero-investment property. We argue that the cost of hashing is in fact amortized, since the hash table will be used by the subsequent sampling operations, and when executing the final plan with full tables. The latter implies that instead of building the hash table during the full execution of the chosen plan, the hashing operation is moved forward and rescheduled to occur when a join with the relation R is sampled for the first time. If the hash table is indeed used during the execution of the plan, then building the hash table before the sampling operation is similar to using a sampling operation with is compliant to the zero-investment property. Similarly, it is possible to sample a join with a sort-merge implementation instead of hash-based one. In that case, the relation R is sorted instead of hashed. This allows the use of a sort-merge join while executing the final plan, which might be a better choice in case the data is known to be already sorted. A sort-merge join also allows ROX to support the sampling and execution of relational joins with range comparison.

6.5 Conclusion

In this chapter, we have presented ROX-sampled a variant of ROX that is suitable for database systems with a pipelined execution strategy. Contrary to ROX-full, ROX-sampled uses only sampled data throughout the whole algorithm, hence consuming and generating a small number of tuples. The execution decisions made during every optimization phase of ROX-sampled are recorded, thus iteratively defining the final execution plan. When all edges are ordered, the final plan is then executed on full tables. A detailed description of the ROX-sampled algorithm has also been given. Experiments assessing the performance and robustness of ROX-sampled have been presented, and the requirements to implement ROX-sampled in a pipelined database systems were briefly studied.

In Section 6.1.1, while giving a global overview of the ROX-sampled approach, three questions were posed. We present the answers below.

Does the use of only small samples during both the optimization and execution steps jeopardize the robustness of the algorithm?
Although only data samples are used throughout the whole algorithm, the experiments have shown that the performance of ROX-sampled is comparable to ROX-full, especially, when using a larger *sample size*: the plans generated are on average 6% slower than their ROX-full counterpart. The latter shows that ROX-sampled is indeed more sensitive than ROX-full with respect to the size of the chosen samples. Moreover, setting the cutoff limit for the sampling operations performed during the execution phases of ROX-sampled to twice the *sample size* is enough to maintain the quality of generated plans.

Will the small generated intermediates be representative enough to detect data correlations?
Experiments have shown that the ROX-sampled approach is robust in the face of different types of correlations. It is capable of not only detecting but also exploiting the existing correlations to generate good execution plans.

ROX-sampled needs, in some situations, to perform redundant operations. Will this reduce the efficiency of ROX-sampled?
It was shown that the sampling overhead in ROX-sampled is kept limited. Despite all the re-execution and sampling operations, the overhead is on average only 6% higher than that of ROX-full.

Finally, we conclude by stressing that the main contribution of this chapter is the generalization of ROX to pipelined systems, allowing the large number of pipelined database engines to integrate the ROX idea into their optimization paradigm.

7 Conclusions

In this chapter, we present our answers to the research questions posed in Chapter 1 and briefly describe possible future research directions. We first quickly review the problem we aim to solve, and give a brief overview of our ROX solution.

Relational database systems have been introduced in the 70's, and since then the optimization of queries submitted to a database system has been extensively researched resulting in the proposal of a multitude of techniques. While sufficient for some applications, the widely used type of optimizers are not always robust, and in some cases pick execution plans that are far from optimal. The reasons behind the shortcomings of classical optimizers are the following: *(i)* they depend on statistics and a cost model which are often inaccurate, not up-to-date, and sometimes even absent, *(ii)* they fail to detect correlations which can unexpectedly make certain plans much cheaper than others, *(iii)* they cannot efficiently handle the large search space of big queries. The challenges faced by traditional relational optimizers and their impact on the quality of the chosen plans are aggravated in the context of XML and XQueries. This is due to the fact that in XML, statistics should capture, in addition to the value of the nodes, the structure of the document. Moreover, the search space of plans for an XQuery query is on average larger than that of relational queries. This is due to the higher number of joins in an XQuery plan resulting from the existence of many XPath steps in a typical query.

To overcome the above challenges, we propose an optimizer that satisfies the following properties: **autonomy** from statistics and cost model, **robustness** in always finding a good execution plan, and **efficiency** in exploring the search space. Our approach is to adopt an optimizer with a fundamentally different internal design which moves the optimization to run-time, and interleaves it with query execution. As such sampling techniques can be used to accurately estimate the cardinality and cost of operators without depending on any statistics and cost model. To detect correlation among the queried data, we introduce the *chain sampling* technique which we believe to be the first generic and robust method to deal

with any type of correlated data. We suggest to explore the search space by interleaving optimization and execution steps, defining the plan incrementally, *i.e.* the plan is built step by step. The exploration is performed efficiently through the chain sampling technique.

Our proposed optimizer is ROX. ROX is the first run-time optimizer for XQueries. It interleaves optimization and execution steps where each optimization phase initiates a sampling-based search for the *superior* sequence of operators. The subsequent execution step executes the chosen sequence and materializes the results, allowing the next optimization phase to benefit from the newly materialized intermediates, and the knowledge which can be extracted from it. ROX has been explained in detail in Chapter 4. It has been implemented on top of the MonetDB database system, and experiments have been conducted which have shown that ROX is indeed robust and efficient, and performs better than and relational compile-time optimizers.

7.1 Revisiting the Research Questions

In this section, we enumerate the research questions introduced in this thesis, and then present the answers given by ROX to these questions.

7.1.1 Main Research Question

The main research question which is the center of focus of this thesis is the following:
Main research question: *How to develop an XQuery optimizer that has the following properties:* ***autonomy***, ***robustness*** *in always finding a good execution plan, and* ***efficiency***.

The above main research question has been divided into the following three sub-questions:

- ***Research question 1:*** *How can an optimizer accurately estimate the cardinality and cost of operators without relying on any a priori collected statistics and cost model?*
- ***Research question 2:*** *How can the correlation existing between several attributes be detected and exploited?*
- ***Research question 3:*** *How can the proposed optimizer guarantee a good quality of decisions?*

We now present the answers given by ROX to the above questions, describing how the run-time optimizer satisfies the 3 properties mentioned in the main research question.

Autonomy

By deferring optimization to run-time, ROX eliminates any dependency on a priori collected statistics and a pre-built cost model. As such it does not suffer from the deficiencies of the current state-of-the-art in XQuery cost estimation, and their adverse effect on the decisions of optimizers. ROX makes autonomously informed optimization decisions by accurately observing, through the use of sampling techniques, the size and characteristics of (intermediate) data and the cost of operators (*Research question 1*).

Robustness

The robustness of ROX is the result of several aspects in its design. We present these next.

Accurate estimations (*Research question 1*): Through the use of sampling techniques, ROX not only eliminates any dependency on the existence of statistics and cost model, but also succeeds in making accurate estimations about the result size and cost of the joins in the graph. The cases in which the estimations deviate from the real values can be avoided through the use of better sampling techniques. Moreover, the alternation of optimization and execution steps in which results are fully materialized allows ROX to update the previously estimated result sizes using the newly materialized intermediates, hence always improving the accuracy of the derived estimations.

Detection of correlation (*Research question 2*): By updating the knowledge in the join graph after each execution phase, ROX can detect previously unnoticed correlations between the newly materialized data and the other data in the join graph. Moreover, exploring the join graph through the chain sampling process in which different sequences of join operators are consecutively sampled enables ROX to discover existing correlations among the joined XML nodes. We note that our chain-sampling technique provides the first generic and robust method to deal with any type of correlated data.

Quality of decisions (*Research question 3*): The ROX algorithm defines the execution plan of a given query iteratively: every optimization step decides which sequence of operators to execute next. This decision is made during the chain sampling process by either the SuperiorPath or the StoppingCondition functions which guarantee that the path returned for execution is a path estimated to be *superior* to all the explored paths in the join graph.

The above three characteristics of our run-time optimizer succeed in making ROX robust in constantly finding (near-)optimal plans and invariably avoiding the bad ones.

Efficiency

ROX explores the search space of a query by iterating optimization and execution phases, defining the execution plan in an augmentation manner. It uses the chain sampling technique to explore different path segments in the graph in search for a *superior* path. To ensure an efficient exploration, ROX uses a stopping condition which detects the existence of a superior path without exploring the whole join graph. Moreover, ROX uses a cutoff limit and small samples as input to the sampling operations of joins to keep the sampling cost under control. The conducted experiments have indeed shown that ROX is an efficient optimizer, it keeps the sampling overhead imposed by the run-time optimization limited. With a sample size equal to 100 tuples, the sampling overhead is on average around 27% of the full execution time.

In summary, ROX, the first proposed run-time optimizer for XQueries, is autonomous, efficient, and robust in finding good execution plans. It does not depend on any statistics and cost model, and accurately estimates the size of joins through sampling. It also gives a certain guarantee on the quality of the paths chosen for execution. Its chain sampling technique provides the first generic and robust method for detecting any type of correlations. ROX improves the state-of-the art in XQuery optimizers both in plan quality and execution time.

7.1.2 Research Question 4:

The fourth research question posed in Chapter 1 is the following:
Research question 4: *How can our proposed optimization technique be applied to different database system architectures (full materialization and pipelined execution strategies)?*

The original variant of ROX, named ROX-full, is proposed in the context of database systems that support full materialization of intermediates. Since most used database systems nowadays adopt a pipelined execution scheme, we designed ROX-sampled, a variant of ROX-full, that is suitable for the aforementioned database architecture. The two ROX variants are similar, but contrary to ROX-full, ROX-sampled uses only sampled data throughout the whole algorithm, hence consuming and generating a small number of tuples at every optimization and execution step.

Although only data samples are used throughout the whole ROX-sampled algorithm and the operations performed during its execution phases are cutoff-sampled, its performance is comparable to that of ROX-full especially when using a relatively larger *sample size* (equal to 1000 tuples). It has been shown that ROX-sampled is a robust optimizer which succeeds in choosing (near-)optimal plans, and is capable of detecting and exploiting different types of correlations. Despite the fact that ROX-

sampled performs considerably more sampling operations than ROX-full, its sampling overhead is kept limited to on average 34% of the full execution time, which is on average only 7% higher than ROX-full.

Summarizing, ROX-sampled is a generalization of the ROX approach, allowing the large number of existing pipelined database systems to integrate the ROX idea into their optimization paradigm. ROX-sampled has shown to be a robust and efficient optimizer with a performance closely comparable to that of ROX-full.

7.2 Additional Strong Aspects of ROX

In addition to the already mentioned characteristics of ROX, we list here other strong aspects of our run-time optimizer.

- **Beyond the current state-of-the-art:** 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*. ROX improves the state-of-the art in XQuery optimizers both in *plan quality as well as running time*.
- **Support of the entire XQuery language:** Although ROX focuses on optimizing the crucial order of the relational joins and XPath steps in a join graph, the possibility to handle multiple join graphs embedded in one execution plan allows ROX to *support the entire XQuery language*. We stress that the fragment of the XQuery language which can be mapped to an execution plan with a single join graph is more expressive than the twig queries widely considered in other previous work [26, 35, 129, 33, 65].
- **Seamless handling of XPath steps and relational joins:** By grouping the XPath steps and relational joins in one join graph structure, ROX is able to integrate XPath- and XQuery-specific optimization techniques in the well-known approach of re-ordering relational joins. In fact, as part of the *seamless optimization of the execution order of the two different types of operators: XPath steps and relational joins*, ROX also *breaks-up and stitches* complex path expressions, and *adaptively determines the execution direction* of a step (*i.e.* whether to execute the step with a forward or a backward axis).
- **Beyond XQuery:** Although the ROX approach is explained in the context of XQuery, we stress that the proposed optimization strategy is generic enough to be exported to other query languages, like SQL and SPARQL.[1]
- **Dynamic environments:** Given a specific query load, the performance of classical optimizers can be enhanced by determining and

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

building beforehand the appropriate statistics that accurately estimate the result size of the operators in any of the queries. For those queries that include attributes without statistics, the performance of the optimizer might degrade. ROX, on the other hand, is not "query specific". Due to its independence of the existence of statistics, ROX is capable of adapting to different query loads, hence performing well in dynamic environments where the workload is continuously changing.

7.3 Future Research Directions

We now finally sketch possible future work to extend and enhance the ROX approach.

7.3.1 Balancing Between the Optimization and Execution Times of a Query

Since ROX intertwines sampling-based query optimization with query evaluation, it becomes possible to strike a balance between these two query evaluation cost factors. ROX can estimate the potential execution cost of a given query by observing, through sampling, the execution time of the joins. Then, based on the estimated cost, it can adaptively determine the amount of time to invest on optimization. A potential approach to balance between the amount of time to spend on the optimization and the execution of a query has been described in Section 5.5.9.

7.3.2 Incorporating Execution Time in the Weight of Edges

The ROX algorithm described in this thesis only looks at the result sizes of the edges in the join graph to decide which sequence of operators to execute next. An alternative is to also measure the *execution time* of the sampled operators and to take it into account in the decision making process. To a limited extent, such functionality is already present in the current ROX prototype which, after deciding to execute an edge, samples the edge in the two possible execution directions trying all applicable physical operators to see which one is fastest. The execution time of an edge can be derived as part of the weight estimation process, and therefore does not lead to any extra sampling cost. In summary, a future adaptation of ROX may use the actual execution time of a sampling operation in the calculation of the weight of an edge, such that deciding which path segment to execute naturally takes into account more characteristics of operator execution.

7.3.3 Pipelining in ROX-full

Even though current RAM sizes and virtual memory techniques allow materializing strategies on many of today's "large" problems, MonetDB and other database systems that are based on full materialization share the risk of unnecessarily materializing large intermediate results. The ROX-full approach of fully materializing the execution result of edges faces the same risk. To avoid this potential problem, a possible extension to ROX-full could identify, while chain sampling, path segments that generate large intermediates, and execute such sub-chains in a pipelined fashion, thus improving scalability. Moreover, to achieve higher CPU efficiency, the vectorized iterator introduced in MonetDB/X100 [23] can be adopted instead of the traditional tuple-by-tuple pipelining mechanism.

7.3.4 Re-Optimization in ROX

Although much robuster than other optimizers, ROX also suffers from a small risk of picking for execution a bad sequence of operators. The latter is due to the adopted sampling techniques which in few cases wrongly estimate the result size of joins. To neutralize the effect of such erroneous estimations, ROX could, while executing the chosen sequence of operators, take note of the actual size of the generated intermediates, and compare it with the estimated values. If the difference between the two numbers is larger than a certain factor f, then execution stops and optimization is re-initiated. Query re-optimization has already been suggested in literature [79, 91]. Since ROX consists of an alternation between optimization and execution, such re-optimization techniques are much easier to incorporate in ROX than in classical database systems. The main question in the ROX scenario is how to determine the value of the factor f which dictates the acceptable gap between the estimations and the actual observed numbers. The work in [91] assigns a validity range for each enumerated plan, defining an upper and lower bound outside which the plan is considered sub-optimal. A similar approach might be employed here, in which the path segment chosen for execution is assigned an upper bound of tuples it is allowed to generate. If the number of tuples produced during the execution of the path exceeds the upper bound, then execution is halted and optimization is re-initiated.

Another question is whether to materialize or disregard the intermediate results generated during the execution of the bad path segment. If the intermediates are considerably large, it might be that joining it with other data will add more cost than re-executing the same path segment at a later stage. In case the intermediates are materialized, they can be used to update the knowledge in the join graph before re-starting the optimization phase.

7.3.5 Pushing Other Operators Inside the Join Graph

An interesting point to study is the possibility to push other types of operators like Sort, Distinct, and Group inside the join graph, and find efficient ways of integrating these operators into the runtime optimization and evaluation environment of ROX. This extension would move ROX beyond the optimization of the order of joins only.

7.3.6 Improvements to Estimation Techniques

We have already mentioned that the sampling techniques used in ROX to sample from tables and indexes are primitive. Therefore, there is room to adopt better sampling techniques which result in more representative sample sets.

Moreover, the implementation of the proposed cutoff sampling approach is front biased, which might lead to erroneous estimations of the result size of joins. We have in Section 5.5.8 described a better technique which can be used to improve the representativeness of the sampling results, and consequently the accuracy of estimations.

Finally, the estimation of $cost(p_j|p_i)$ is done through linear extrapolation, which in some cases might result in estimations that are from the real value. To derive a more accurate estimation of the cost, one of the techniques proposed in Section 4.3.6 could be adopted.

7.3.7 Other Possible Future Work

We briefly list here some additional future work.

1. Investigate the use of a dynamic sample size which is adaptively determined by ROX based on, among others, the size of the sampled table and the time already spent on optimization.
2. Investigate the possibility of filtering out, during the query evaluation, the duplicate data generated by joins, and re-introduce these duplicates when generating the final result of the join graph.
3. Test the ROX approach with a wider variety of data and queries, including purely relational use cases, to better confirm ROX's robustness and efficiency.
4. Conduct an experimental comparison between the theoretical and the implemented chain sampling approaches.
5. Implement ROX-sampled in a real pipelined database system and compare it to its own optimizer.
6. Investigate the possibility of adopting the ROX approach in non-relational XML database systems, like for instance Natix [44].[2]

[2] http://pi3.informatik.uni-mannheim.de/natix.html

Bibliography

[1] Extensible Markup Language (XML). Website. http://www.w3.org/XML/.

[2] FunctX XQuery Functions: Hundreds of useful examples. Website. http://www.xqueryfunctions.com/xq/.

[3] MonetDB database system with XQuery front-end. Website. http://monetdb.cwi.nl/XQuery/.

[4] XMark — An XML Benchmark Project. Website. http://www.xml-benchmark.org/.

[5] XML Path Language (XPath) 2.0. Website. http://www.w3.org/TR/xpath20/.

[6] XQuery 1.0: An XML Query Language. Website. http://www.w3.org/TR/xquery/.

[7] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: Run-Time Optimization of XQueries. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 615–626, 2009.

[8] R. Abdel Kader, P. A. Boncz, S. Manegold, and M. v. Keulen. ROX: The Robustness of a Run-Time XQuery Optimizer Against Correlated Data. In *Proceedings of the 26th International Conference on Data Engineering*, pages 1185–1188, 2010.

[9] R. Abdel Kader, M. v. Keulen, P. A. Boncz, and S. Manegold. Runtime Optimization for Pipelined Systems. In *Proceedings of the IV Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.

[10] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 591–600, 2001.

[11] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 181–192, 1999.

[12] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 141–152, 2002.

[13] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in Oracle Rdb. *The VLDB Journal*, pages 229–237, 1996.

[14] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[15] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 107–118, 2005.

[16] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. Simmen, M. Wang, and C. Zhang. Cost-Based Optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–319, 2006.

[17] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. In *In Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407, 1991.

[18] G. Bhargava, P. Goel, and B. Iyer. Hypergraph Based Reorderings of Outer Join Queries with Complex Predicates. *SIGMOD Record*, pages 304–315, 1995.

[19] P. Bizarro, N. Bruno, and D. J. DeWitt. Progressive Parametric Query Optimization. *IEEE Transaction on Knowledge and Data Engineering*, pages 582–594, 2009.

[20] 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 *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2006.

[21] P. A. Boncz, J. Flokstra, T. Grust, M. v. Keulen, S. Manegold, K. S. Mullender, J. Rittinger, and J. Teubner. MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 1190–1193, 2006.

[22] P. A. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives*, 2005.

[23] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[24] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2002.

[25] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a Multidimensional Workload-Aware Histogram. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 211–222, 2001.

[26] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[27] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, 1998.

[28] S. Chaudhuri. Query Optimizers: Time to Rethink the Contract? In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 961–968, 2009.

[29] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. *SIGMOD Record*, pages 263–274, 1999.

[30] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A Pay-As-You-Go Framework for Query Execution Feedback. *Proceedings of the VLDB Endowment*, pages 1141–1152, 2008.

[31] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Diagnosing Estimation Errors in Page Counts Using Execution Feedback. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1013–1022, 2008.

[32] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. *SIGMOD Record*, pages 161–172, 1994.

[33] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 283–294, 2006.

[34] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *Proceedings of the 17th International Conference on Data Engineering*, pages 595–604, 2001.

[35] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 237–248, 2003.

[36] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems*, pages 163–186, 1984.

[37] S. Cluet and G. Moerkotte. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Proceedings of the 5th International Conference on Database Theory*, pages 54–67, 1995.

[38] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 150–160, 1994.

[39] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350, 2001.

[40] D. DeHaan and F. W. Tompa. Optimal Top-Down Join Enumeration. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 785–796, 2007.

[41] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[42] S. Ewen, H. Kache, V. Markl, and V. Raman. Progressive Query Optimization for Federated Queries. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 847–864, 2006.

[43] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *Proceedings of the Database and Expert Systems Applications, 9th International Conference*, pages 726–735, 1998.

[44] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *The VLDB Journal*, pages 292–314, 2002.

[45] D. K. Fisher and S. Maneth. Structural Selectivity Estimation for XML Documents. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 626–635, 2007.

[46] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 181–191, 2002.

[47] C. Galindo-Legaria and A. Rosenthal. Outerjoin Simplification and Reordering for Query Optimization. *ACM Transactions on Database Systems*, pages 43–74, 1997.

[48] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, Randomized Join-Order Selection - Why Use Transformations? In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 85–95, 1994.

[49] C. A. Galindo-Legaria and A. Rosenthal. How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 402–409, 1992.

[50] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 228–238, San Francisco, CA, USA, 1998.

[51] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[52] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 436–445, 1997.

[53] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. *SIGMOD Record*, pages 160–172, 1987.

[54] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, 1993.

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

[56] T. Grust. Purely Relational FLWORs. In *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives*, 2005.

[57] T. Grust, M. v. Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 524–535, 2003.

[58] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, pages 91–131, 2004.

[59] T. Grust, M. Mayr, and J. Rittinger. XQuery Join Graph Isolation: Celebrating 30+ Years of XQuery Processing Technology. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1167–1170, 2009.

[60] T. Grust, M. Mayr, and J. Rittinger. Let SQL Drive the XQuery Workhorse (XQuery Join Graph Isolation). In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 147–158, 2010.

[61] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 252–263, 2004.

[62] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. *SIGMOD Record*, pages 377–388, 1989.

[63] P. J. Haas, F. Hueske, and V. Markl. Detecting Attribute Dependencies from Query Feedback. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 830–841, 2007.

[64] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. *Journal of Computer and System Sciences*, pages 550–569, 1996.

[65] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. How to Recognise Different Kinds of Tree Patterns From Quite a Long Way Away. In *Proceedings of the 2007 Programming Language Technologies for XML*, pages 14–24, 2007.

[66] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 167–178, 2002.

[67] A. Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 766–777, 2003.

[68] T. Ibaraki and T. Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems*, pages 482–502, 1984.

[69] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *Third Biennial Conference on Innovative Data Systems Research*, pages 68–78, 2007.

[70] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.

[71] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. *SIGMOD Record*, pages 268–277, 1991.

[72] Y. E. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. *SIGMOD Record*, pages 312–321, 1990.

[73] Y. E. Ioannidis and Y. C. Kang. Left-Deep vs. Bushy Trees: an Analysis of Strategy Spaces and its Implications for Query Optimization. *SIGMOD Record*, pages 168–177, 1991.

[74] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 103–114, 1992.

[75] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *The VLDB Journal*, pages 132–151, 1997.

[76] Y. E. Ioannidis and E. Wong. Query Optimization by Simulated Annealing. *SIGMOD Record*, pages 9–22, 1987.

[77] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4):274–291, 2002.

[78] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Revised Papers from the 8th International Workshop on Database Programming Languages*, pages 149–164, 2002.

[79] N. Kabra and D. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*, 1998.

[80] C.-C. Kanne, M. Brantner, and G. Moerkotte. Cost-Sensitive Reordering of Navigational Primitives. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 742–753, 2005.

[81] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 133–144, 2002.

[82] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140, 2002.

[83] M. L. Kersten and S. Manegold. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[84] D. Kossmann and K. Stocker. Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. *ACM Transactions on Database Systems*, pages 43–82, 2000.

[85] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 128–137, 1986.

[86] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing Large Join Queries Using A Graph-Based Approach. *IEEE Transactions on Knowledge and Data Engineering*, pages 298–315, 2001.

[87] M. K. Lee, J. C. Freytag, and G. M. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 218–229, 1988.

[88] H. Li, M. L. Lee, W. Hsu, and G. Cong. An Estimation System for XPath Expressions. In *Proceedings of the 22nd International Conference on Data Engineering*, page 54, 2006.

[89] Q. Li, M. Shao, V. Markl, K. S. Beyer, L. S. Colby, and G. M. Lohman. Adaptively Reordering Joins during Query Execution. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 26–35, 2007.

[90] G. M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 18–27, 1988.

[91] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing Through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004.

[92] N. May, S. Helmer, C.-C. Kanne, and G. Moerkotte. XQuery Processing in Natix with an Emphasis on Join Ordering. In *Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives*, pages 49–54, 2004.

[93] S. Mayer, T. Grust, M. v. Keulen, and J. Teubner. An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 1305–1308, 2004.

[94] J. Mchugh and J. Widom. Optimizing Branching Path Expressions. Technical report, Stanford University, 1999.

[95] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 315–326, 1999.

[96] C. Mishra and N. Koudas. Join Reordering by Join Simulation. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 493–504, 2009.

[97] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 930–941, 2006.

[98] G. Moerkotte and T. Neumann. Dynamic Programming Strikes Back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 539–552, 2008.

[99] T. Neumann. Query Simplification: Graceful Degradation for Join-Order Optimization. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 403–414, 2009.

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

[101] K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 314–325, 1990.

[102] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 39–48, 1992.

[103] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph-Structured XML Databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 358–369, 2002.

[104] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph-Structured XML Databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Ddata*, pages 358–369, 2002.

[105] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 466–477, 2002.

[106] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2004.

[107] V. G. V. P. Prasad. Parametric Query Optimization: A Geometric Approach. Master's thesis, Indian Institute of Technology, Kanpur, 1999.

[108] V. Raman, A. Deshpande, and J. M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proceedings of the 19th International Conference on Data Engineering*, pages 353–364, 2003.

[109] J. Rao, B. G. Lindsay, G. M. Lohman, H. Pirahesh, and D. E. Simmen. Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering. In *Proceedings of the 17th International Conference on Data Engineering*, pages 585–594, 2001.

[110] S. V. U. M. Rao. Parametric Query Optimization: A Non-Geometric Approach. Master's thesis, Indian Institute of Technology, Kanpur, 1999.

[111] N. Reddy and J. R. Haritsa. Analyzing Plan Diagrams of Database Query Optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1228–1239, 2005.

[112] A. Rosenthal and C. Galindo-Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. *SIGMOD Record*, pages 291–299, 1990.

[113] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[114] S. Seshradi. *Probabilistic Methods in Query Processing*. PhD thesis, University of Wisconsin, 1992.

[115] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, 1999.

[116] L. Sidirourgos and P. Boncz. Generic and Updatable XML Value Indices Covering Equality and Range Lookups. Technical Report INS-E0802, CWI.

[117] L. Sidirourgos and P. Boncz. Generic and Updatable XML Value Indices Covering Equality and Range Lookups. In *Proceedings of the 2009 EDBT/ICDT Workshops*, pages 65–73, 2009.

[118] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *Proceedings of the 22nd International Conference on Data Engineering*, page 39, 2006.

[119] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, pages 191–208, 1997.

[120] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28, 2001.

[121] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions Database System*, pages 189–222, 1976.

[122] A. Swami. Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques. *SIGMOD Record*, pages 367–376, 1989.

[123] A. Swami and A. Gupta. Optimization of Large Join Queries. *SIGMOD Record*, pages 8–17, 1988.

[124] M. Templeton, H. Henley, E. Maros, and D. J. V. Buer. InterViso: Dealing with the Complexity of Federated Database Access. *The VLDB Journal*, pages 287–318, 1995.

[125] J. Teubner, T. Grust, S. Maneth, and S. Sakr. Dependable Cardinality Forecasts for XQuery. *Proceedings of the VLDB Endowment*, pages 463–477, 2008.

[126] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 35–46, 1996.

[127] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 240–251, 2004.

[128] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proceedings of the 8th International Conference on Extending Database Technology*, pages 590–608, 2002.

[129] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the 19th International Conference on Data Engineering*, pages 443–454, 2003.

[130] V. Zadorozhny, L. Raschid, M. E. Vidal, T. Urhan, and L. Bright. Efficient Evaluation of Queries in a Mediator for WebSources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 85–96, 2002.

[131] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 289–300, 2005.

Siks dissertations

1998

1998-1	Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2	Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information
1998-3	Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
1998-4	Dennis Breuker (UM) Memory versus Search in Games
1998-5	E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999

1999-1	Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling ofQuality Change of Agricultural Products
1999-2	Rob Potharst (EUR) Classification using decision trees and neural nets
1999-3	Don Beal (UM) The Nature of Minimax Search
1999-4	Jacques Penders (UM) The practical Art of Moving Physical Objects
1999-5	Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6	Niek J.E. Wijngaards (VU) Re-design of compositional systems
1999-7	David Spelt (UT) Verification support for object database design
1999-8	Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1	Frank Niessink (VU) Perspectives on Improving Software Maintenance
2000-2	Koen Holtman (TUE) Prototyping of CMS Storage Management
2000-3	Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4	Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5	Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.
2000-6	Rogier van Eijk (UU) Programming Languages for Agent Communication
2000-7	Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management
2000-8	Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks
2000-9	Florian Waas (CWI) Principles of Probabilistic Query Optimization
2000-10	Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture
2000-11	Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001

2001-1	Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
2001-2	Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
2001-3	Maarten van Someren (UvA) Learning as problem solving
2001-4	Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5	Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6	Martijn van Welie (VU) Task-based User Interface Design
2001-7	Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8	Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9	Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10	Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11	Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01	Nico Lassing (VU) Architecture-Level Modifiability Analysis
2002-02	Roelof van Zwol (UT) Modelling and searching web-based document collections
2002-03	Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
2002-04	Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05	Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents
2002-06	Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain
2002-07	Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08	Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09	Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems
2002-10	Brian Sheppard (UM) Towards Perfect Play of Scrabble
2002-11	Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12	Albrecht Schmidt (Uva) Processing XML in Database Systems
2002-13	Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications
2002-14	Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15	Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16	Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications
2002-17	Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01	Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments
2003-02	Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems
2003-03	Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04	Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology
2003-05	Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach
2003-06	Boris van Schooten (UT) Development and specification of virtual environments
2003-07	Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks
2003-08	Yongping Ran (UM) Repair Based Scheduling
2003-09	Rens Kortmann (UM) The resolution of visually guided behaviour
2003-10	Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11	Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12	Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval
2003-13	Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models
2003-14	Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15	Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems
2003-16	Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17	David Jansen (UT)

	Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18	Levente Kocsis (UM) Learning Search Decisions

2004

2004-01	Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02	Lai Xu (UvT) Monitoring Multi-party Contracts for E-business
2004-03	Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04	Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures
2004-05	Viara Popova (EUR) Knowledge discovery and monotonicity
2004-06	Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques
2004-07	Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08	Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise
2004-09	Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning
2004-10	Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects
2004-11	Michel Klein (VU) Change Management for Distributed Ontologies
2004-12	The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
2004-13	Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
2004-14	Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15	Arno Knobbe (UU) Multi-Relational Data Mining
2004-16	Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
2004-17	Mark Winands (UM) Informed Search in Complex Games
2004-18	Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
2004-19	Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2004-20	Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005

2005-01	Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
2005-02	Erik van der Werf (UM)) AI techniques for the game of Go
2005-03	Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language
2005-04	Nirvana Meratnia (UT) Towards Database Support for Moving Object data
2005-05	Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06	Pieter Spronck (UM)

	Adaptive Game AI
2005-07	Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08	Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09	Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
2005-10	Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
2005-11	Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
2005-12	Csaba Boer (EUR) Distributed Simulation in Industry
2005-13	Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14	Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15	Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
2005-16	Joris Graaumans (UU) Usability of XML Query Languages
2005-17	Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
2005-18	Danielle Sent (UU) Test-selection strategies for probabilistic networks
2005-19	Michel van Dartel (UM) Situated Representation
2005-20	Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives
2005-21	Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01	Samuil Angelov (TUE) Foundations of B2B Electronic Contracting
2006-02	Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations
2006-03	Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems
2006-04	Marta Sabou (VU) Building Web Service Ontologies
2006-05	Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines
2006-06	Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling
2006-07	Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering
2006-08	Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-09	Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion
2006-10	Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems
2006-11	Joeri van Ruth (UT) Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA)
Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)

Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramírez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA)

	Topic Driven Access to Scientific Handbooks
2008-14	Arthur van Bunningen (UT)
	Context-Aware Querying; Better Answers with Less Effort
2008-15	Martijn van Otterlo (UT)
	The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.
2008-16	Henriette van Vugt (VU)
	Embodied agents from a user's perspective
2008-17	Martin Op 't Land (TUD)
	Applying Architecture and Ontology to the Splitting and Allying of Enterprises
2008-18	Guido de Croon (UM)
	Adaptive Active Vision
2008-19	Henning Rode (UT)
	From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
2008-20	Rex Arendsen (UVA)
	Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.
2008-21	Krisztian Balog (UVA)
	People Search in the Enterprise
2008-22	Henk Koning (UU)
	Communication of IT-Architecture
2008-23	Stefan Visscher (UU)
	Bayesian network models for the management of ventilator-associated pneumonia
2008-24	Zharko Aleksovski (VU)
	Using background knowledge in ontology matching
2008-25	Geert Jonker (UU)
	Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
2008-26	Marijn Huijbregts (UT)
	Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
2008-27	Hubert Vogten (OU)
	Design and Implementation Strategies for IMS Learning Design
2008-28	Ildiko Flesch (RUN)
	On the Use of Independence Relations in Bayesian Networks
2008-29	Dennis Reidsma (UT)
	Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
2008-30	Wouter van Atteveldt (VU)
	Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
2008-31	Loes Braun (UM)
	Pro-Active Medical Information Retrieval
2008-32	Trung H. Bui (UT)
	Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
2008-33	Frank Terpstra (UVA)
	Scientific Workflow Design; theoretical and practical issues
2008-34	Jeroen de Knijf (UU)
	Studies in Frequent Tree Mining
2008-35	Ben Torben Nielsen (UvT)
	Dendritic morphologies: function shapes structure

2009

2009-01	Rasa Jurgelenaite (RUN)
	Symmetric Causal Independence Models
2009-02	Willem Robert van Hage (VU)
	Evaluating Ontology-Alignment Techniques
2009-03	Hans Stol (UvT)
	A Framework for Evidence-based Policy Making Using IT
2009-04	Josephine Nabukenya (RUN)

	Improving the Quality of Organisational Policy Making using Collaboration Engineering
2009-05	Sietse Overbeek (RUN)
	Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
2009-06	Muhammad Subianto (UU)
	Understanding Classification
2009-07	Ronald Poppe (UT)
	Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08	Volker Nannen (VU)
	Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09	Benjamin Kanagwa (RUN)
	Design, Discovery and Construction of Service-oriented Systems
2009-10	Jan Wielemaker (UVA)
	Logic programming for knowledge-intensive interactive applications
2009-11	Alexander Boer (UVA)
	Legal Theory, Sources of Law & the Semantic Web
2009-12	Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
	Operating Guidelines for Services
2009-13	Steven de Jong (UM)
	Fairness in Multi-Agent Systems
2009-14	Maksym Korotkiy (VU)
	From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15	Rinke Hoekstra (UVA)
	Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16	Fritz Reul (UvT)
	New Architectures in Computer Chess
2009-17	Laurens van der Maaten (UvT)
	Feature Extraction from Visual Data
2009-18	Fabian Groffen (CWI)
	Armada, An Evolving Database System
2009-19	Valentin Robu (CWI)
	Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20	Bob van der Vecht (UU)
	Adjustable Autonomy: Controling Influences on Decision Making
2009-21	Stijn Vanderlooy (UM)
	Ranking and Reliable Classification
2009-22	Pavel Serdyukov (UT)
	Search For Expertise: Going beyond direct evidence
2009-23	Peter Hofgesang (VU)
	Modelling Web Usage in a Changing Environment
2009-24	Annerieke Heuvelink (VUA)
	Cognitive Models for Training Simulations
2009-25	Alex van Ballegooij (CWI)
	"RAM: Array Database Management through Relational Mapping"
2009-26	Fernando Koch (UU)
	An Agent-Based Model for the Development of Intelligent Mobile Services
2009-27	Christian Glahn (OU)
	Contextual Support of social Engagement and Reflection on the Web
2009-28	Sander Evers (UT)
	Sensor Data Management with Probabilistic Models
2009-29	Stanislav Pokraev (UT)
	Model-Driven Semantic Integration of Service-Oriented Applications
2009-30	Marcin Zukowski (CWI)
	Balancing vectorized query execution with bandwidth-optimized storage
2009-31	Sofiya Katrenko (UVA)
	A Closer Look at Learning Relations from Text
2009-32	Rik Farenhorst (VU) and Remco de Boer (VU)
	Architectural Knowledge Management: Supporting Architects and Auditors
2009-33	Khiet Truong (UT)
	How Does Real Affect Affect Affect Recognition In Speech?
2009-34	Inge van de Weerd (UU)

Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL)
Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN)
Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN)
Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU)
Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin)
Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT)
Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT)
Digital Analysis of Paintings

2009-42 Toine Bogers (UvT)
Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT)
Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT)
Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU)
Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA)
Querying XML: Benchmarks and Recursion

2010

2010-01 Matthijs van Leeuwen (UU)
Patterns that Matter

2010-02 Ingo Wassink (UT)
Work flows in Life Science

2010-03 Joost Geurts (CWI)
A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT)
Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT)
Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT)
Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT)
A Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL)
Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL)
A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL)
Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD)
The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU)
Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN)
High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU)
Automated Web Service Reconfiguration

2010-15	Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models
2010-16	Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice
2010-17	Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications
2010-18	Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19	Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems
2010-20	Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship in Emergent Narrative
2010-21	Harold van Heerde (UT) Privacy-aware data management by means of data degradation
2010-22	Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
2010-23	Bas Steunebrink (UU) The Logical Structure of Emotions
2010-24	Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies
2010-25	Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
2010-26	Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
2010-27	Marten Voulon (UL) Automatisch contracteren
2010-28	Arne Koopman (UU) Characteristic Relational Patterns
2010-29	Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels
2010-30	Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
2010-31	Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web
2010-32	Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-33	Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval
2010-34	Teduh Dirgahayu (UT) Interaction Design in Service Compositions
2010-35	Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval
2010-36	Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification
2010-37	Niels Lohmann (TUE) Correctness of services and their composition
2010-38	Dirk Fahland (TUE) From Scenarios to components
2010-39	Ghazanfar Farooq Siddiqui (VU) Integrative modeling of emotions in virtual agents
2010-40	Mark van Assem (VU) Converting and Integrating Vocabularies for the Semantic Web
2010-41	Guillaume Chaslot (UM) Monte-Carlo Tree Search
2010-42	Sybren de Kinderen (VU) Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach
2010-43	Peter van Kranenburg (UU)

	A Computational Approach to Content-Based Retrieval of Folk Song Melodies
2010-44	Pieter Bellekens (TUE)
	An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain
2010-45	Vasilios Andrikopoulos (UvT)
	A theory and model for the evolution of software services
2010-46	Vincent Pijpers (VU)
	e3alignment: Exploring Inter-Organizational Business-ICT Alignment
2010-47	Chen Li (UT)
	Mining Process Model Variants: Challenges, Techniques, Examples
2010-48	Milan Lovric (EUR)
	Behavioral Finance and Agent-Based Artificial Markets
2010-49	Jahn-Takeshi Saito (UM)
	Solving difficult game positions
2010-50	Bouke Huurnink (UVA)
	Search in Audiovisual Broadcast Archives
2010-51	Alia Khairia Amin (CWI)
	Understanding and supporting information seeking tasks in multiple sources
2010-52	Peter-Paul van Maanen (VU)
	Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention
2010-53	Edgar Meij (UVA)
	Combining Concepts and Language Models for Information Access

Summary

Query optimization is the most important and complex phase of answering a user query. While sufficient for some applications, the widely used type of relational optimizers are not always robust, picking execution plans that are far from optimal. This is due to several reasons. First, they depend on statistics and a cost model which are often inaccurate, and sometimes even absent. Second, they fail to detect correlations which can unexpectedly make certain plans considerably cheaper than others. Finally, they cannot efficiently handle the large search space of big queries.

The challenges faced by traditional relational optimizers and their impact on the quality of the chosen plans are aggravated in the context of XML and XQueries. This is due to the fact that in XML, it is harder to collect and maintain representative statistics since they have to capture more information about the document. Moreover, the search space of plans for an XQuery query is on average larger than that of relational queries, due to the higher number of joins resulting from the existence of many XPath steps in a typical XQuery.

To overcome the above challenges, we propose ROX, a Run-time Optimizer for XQueries. ROX is **autonomous**, *i.e.* it does not depend on any statistics and cost models, **robust** in always finding a good execution plan while detecting and benefiting from correlations, and **efficient** in exploring the search space of plans. We show, through experiments, that ROX is indeed robust and efficient, and performs better than relational compile-time optimizers. ROX adopts a **fundamentally different internal design** which moves the optimization to run-time, and interleaves it with query execution. The search space is efficiently explored by alternating optimization and execution phases, defining the plan incrementally. Every execution step executes a set of operators and materializes the results, allowing the next optimization phase to benefit from the knowledge extracted from the newly materialized intermediates. Sampling techniques are used to accurately estimate the cardinality and cost of operators. **To detect correlations**, we introduce the *chain sampling* technique, the first generic and robust method to deal with any type of correlated data. We also extend the ROX idea to pipelined architectures to allow most of the existing database systems to benefit from our research.

Samenvatting

Query-optimalisatie is de meest belangrijke en gecompliceerde fase bij het beantwoorden van een query. Ook al zijn de standaard, veelgebruikte relationele optimalisatiemethoden goed genoeg voor sommige toepassingen, ze zijn niet altijd robuust; soms worden executieschema's gekozen die verre van optimaal zijn. Dit heeft meerdere oorzaken. Ten eerste zijn de optimizers afhankelijk van statistieken en een kostenmodel die vaak onnauwkeurig, en soms zelfs volledig afwezig zijn. Ten tweede falen ze in het detecteren van correlatie in de gegevens, terwijl deze correlatie ervoor kan zorgen dat bepaalde schema's, soms onverwacht, aanzienlijk efficiënter zijn dan andere.

De uitdaging waar traditionele optimizers—met name wat betreft hun impact op de kwaliteit van de gekozen schema's—mee te maken hebben, worden verder uitvergroot in de context van XML en XQueries. Dit komt vanwege het feit dat met XML het moeilijker is om representatieve statistieken te verzamelen en te beheren, en omdat de statistieken meer informatie over het document zelf moeten vastleggen. Bovendien is de zoekruimte bij het vinden van schema's voor een XQuery-query gemiddeld gezien groter dan die voor relationele queries. Dit wordt veroorzaakt door het grote aantal joins die voortkomen uit de vele XPath-stappen in een doorsnee XQuery.

Om bovenstaande uitdagingen het hoofd te bieden stellen wij ROX voor, wat staat voor Run-time Optimizer voor XQueries. ROX is *autonoom*, dat wil zeggen, het is niet afhankelijk van welke statistieken en kostenmodellen dan ook. Het is *robuust* met betrekking tot het altijd kunnen vinden van een goed executieschema, waarbij het correlaties detecteert en ervan profiteert, en het is *efficiënt* in het doorzoeken van de zoekruimte van mogelijke executieschema's. Door middel van experimenten laten we zien dat ROX inderdaad robuust en efficiënt is en beter presteert dan relationele compile-time optimizers. ROX maakt gebruik van fundamenteel ander ontwerp en andere technieken, waarbij de optimalisatie verplaatst wordt naar de uitvoeringsfase van een query. De optimalisatie en uitvoering van een query worden om-en-om uitgevoerd. De zoekruimte wordt efficiënt

verkend bij dit afwisselend uitvoeren en optimaliseren, waardoor het executieschema incrimenteel wordt bepaald. Bij elke uitvoeringsstap wordt een verzameling operatoren uitgevoerd waarvan het resultaat wordt bewaard. De volgende optimalisatiestap kan profiteren van kennis die uit dit voorgaande tussenresultaat afgeleid kan worden. Steekproeftechnieken worden gebruikt om nauwkeurig de kardinaliteit en de kosten van de operatoren te kunnen inschatten. Om correlatie te kunnen detecteren introduceren we een techniek baseerd op een *keten* van steekproeven, de eerste generieke en robuuste methode die om kan gaan met alle typen correlaties die voor kunnen komen in gegevens. Daarnaast breiden we ROX-aanpak uit voor toepassing in databases met een *pipelined* architectuur, wat het mogelijk maakt dat de meeste bestaande databasesystemen kunnen profiteren van ons onderzoek.