Cost-Based XQuery Optimization in Native XML Database Systems

Concepts, Implementation, and Empirical Evaluation

Beim Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
eingereichte Dissertation

von

Diplom-Informatiker
Andreas Matthias Weiner

Dekan des Fachbereichs
Prof. Dr. Arnd Poetzsch-Heffter

Datum der Einreichung
31. Mai 2011

Abstract

Over the past three decades, the history of query processing in database management systems has shown that cost-based query optimization is an effective approach for finding sufficient low-level evaluation strategies for queries written in high-level declarative query languages like *SQL*.

In recent years, the *eXtensible Markup Language (XML)* (Bray et al., 2008) has been established as the de-facto standard for exchanging semi-structured data between individuals, business partners, and various organizations. Today's *native XML database management systems (XDBMSs)* provide a stable infrastructure for efficiently storing, indexing, and querying small-to-large XML documents.

Boag et al. (2007) introduced *XQuery* as a semi-declarative programming language that is now considered to be the language of choice to query XML documents.

Nowadays, in an XDBMS, we can dispose of a potpourri of various join operators (structural joins and value-based joins) and a still growing set of indexes as low-level building blocks for query processing. For the evaluation of an XQuery expression in an XDBMS, numerous semantically equivalent combinations of these operations are possible. Choosing the most efficient one out of a tremendously large set of combinations is crucial for effective query processing in throughput-oriented systems. To achieve this goal, the cost of each building block is modeled as the sum of IO cost and CPU cost. Using these cost formulæ, a query optimizer can choose the cheapest one out of several alternative building blocks and, finally, combine them in an optimal way.

In this thesis, we assess how and whether concepts and techniques of relational cost-based query optimization can be reused in the context of XDBMSs to optimize XQuery expressions. Furthermore, we show which new techniques make cost-based optimization even more effective in such systems.

In summary, this thesis contributes to the following research topics:

- **Transformation Rules** To enable a query optimizer to generate semantically equivalent plans, various *transformation rules* are introduced for structural joins, valued-based joins, and index structures.

- **Cost Model** Based on the analysis of the low-level building blocks used for query evaluation, we develop a *cost model* for estimating the CPU and IO costs.

- **Cardinality Estimation** To allow the plan generator to make use of the cost model, reliable estimates for XQuery expressions must be furnished. In this work, we contribute to the dependable estimation of structural and value-based XQuery expressions.

- **Generic Query Optimization Framework** By describing and implementing an extensible *query optimization framework*, a testbed for empirically evaluating all concepts developed in this thesis is provided.

- **Empirical Evaluation** We empirically assess the correctness of our cost model. By employing and comparing several instances of our query optimization framework, we (1) derive a minimal set of rewrite rules that allow for stable cost-based query optimization and (2) provide several *design recommendations* (e. g., when it is useful to generate indexes on particular paths) serving as simple heuristics for simplifying database administrators' lives.

Conventions

URL	URLs in footnotes, text, or bibliography
Text	Emphasized text
Code	XQuery code if presented standalone
Code	Inlined XQuery language constructs
IDENTIFIER	Rule identifier (transformation or cardinality inference)
Identifier	Abstract plan type or identifier of rewrite patterns
function(...)	Function identifiers in algorithms
$=$	Comparison operator in rules and algorithms
$\longleftarrow$	Value assignments in transformation rules (Chapter 6)
$=^!$	Assignment of inferred cardinality value in Chapter 7
$\sqsubseteq^!$	Inferred domain inclusion relationship in Chapter 7

Nomenclature

BP	Base Profile
CAS	Content-and-Structure Index
DAG	Directed Acyclic Graph
EXsum	Element-Centered XML Summarization
HTJ	Holistic Twig Join
IP	Intermediate Profile
PAL	Physical Algebra
PAP	Primary Access Path
PCR	Path-Class Reference
PPO	Path Processing Operator
PS	Path Synopsis
QEP	Query Execution Plan
RDBMS	Relational Database Management System
SAP	Secondary Access Path
SJ	Structural Join
TAP	Tertiary Access Path
XDBMS	XML Database Management System
XQGM	XML Query Graph Model
XTC	XML Transaction Coordinator
XTCcmp	XTC XQuery-to-XQGM compiler
XUG	XTC Universal GUI

Contents

List of Figures

List of Tables

I

Introduction

1 Motivation

"We should not look back unless it is to derive useful lessons from past errors, and for the purpose of profiting by dearly bought experience."

(George Washington)

In 1998, the World Wide Web Consortium (W3C) gave birth to the *Extensible Markup Language* (XML) by publishing the first *W3C Recommendation* dealing with this novel meta language (Bray et al., 1998). Being now more than 10 years publicly available, it has become the de-facto standard for exchanging and representing semi-structured data in numerous application areas. Therefore, it is not astonishing that there exists, for almost every business scenario, at least one standardized XML format, for example, FIXML[1] for financial applications—that allows for collaboration within and across enterprises.

1.1 Managing XML

Today, efficient management of XML documents is of utmost importance. Even though XML started as an exchange format, enterprises would like—or even have to—permanently store XML documents, for example, due to legal issues. The management of such documents raises new challenges for transaction isolation and query processing. For example, in financial application logging scenarios, cooperative and concurrent actions of hundreds of users must be supported (Härder et al., 2010).

For almost two score years, *Relational Database Management Systems* (RDMBSs) have been a reliable means for efficiently handling very large databases. Therefore, it seems to be quite natural to use them for coping with XML documents. Normally, XML documents are tree-structured graphs, hence, storing them using an RDBMS raises several problems: (1) Storing XML documents in a relational storage (flat and tuple-based) requires to split—here, the terminus technicus is "to shred"—the XML tree into probably numerous tables. (2) Shredding makes efficient transaction isolation laborious and avoids direct access to it using an XML query language (Haustein and

[1] For more information, see: `http://www.fixprotocol.org/specifications`

Härder, 2007). To avoid expensive transitions between the relational and the XML data model, native *XML Database Management Systems* (XDBMSs) provide an alternative approach that allows to store XML documents as they are. By preserving their structure, fine-granular transaction isolation and efficient XML query processing are possible. Furthermore, it is promising that even major database vendors such as IBM or Oracle rely nowadays on a native XML storage mechanism, even though they originally started with hybrid or shredding techniques.

1.2 Querying XML

In accordance with the rapid proliferation of XML, users have been demanding for a tailor-made query language. In the past, we have seen several language proposals such as *XPath 1.0, Quilt, XQL,* or *XML-QL* that finally led to the development of *XQuery 1.0* (Boag et al., 2007; Lehner and Schöning, 2004). XQuery is much more than a query language—it is a Turing-complete, semi-declarative programming language. As SQL is the mother tongue of every relational database system, XML database systems must at least fluently speak XQuery, because both languages are considered de-facto standards in their fields.

For the evaluation of XQuery expressions in native XDBMSs, we can dispose of a potpourri of join operators (structural joins and value-based joins) and a still growing set of indexes as low-level building blocks for query processing (Mathis, 2009). Using these physical operators, numerous semantically equivalent plans can be derived. Choosing the most efficient one out of a tremendously large set of combinations is crucial for effective query processing in throughput-oriented systems.

1.3 Cost-Based XQuery Optimization

In the mid-1970s, *System R*—the fist prototype of a relational database management system—was developed by Astrahan et al. (1976). Even in this early stage, this system incorporated a cost-based query optimizer (Selinger et al., 1979). Database research history teaches us that RDBMSs became extremely successful, because of their unique way for handling queries: They provide a declarative query language (SQL), which supports the user in describing *what* he is searching for, instead of *how* to get it. Combining this language with a cost-based optimization infrastructure allows to formulate and eval-

uate complex queries in an efficient way; even if the user is not a database expert.

Even though XQuery—taken as a whole—is not completely declarative, the language fragment that is primarily used in database systems, is mainly declarative. Hence, cost-based optimization is also a possible way to handle these queries. In an interview with the *Communications of the ACM*, cost-based query optimization pioneer Patricia G. Selinger states on XQuery processing in database systems:

> *"[...] I think it is absolutely essential to continue on the path of automatic query optimization rather than put programmers back into the game of understanding exact data structures and doing the navigation in the application program manually"* (Patricia G. Selinger, as quoted in Hamilton, 2008).

Her statement is promising and especially true, if we recall that the search space for XQuery execution plans is even larger than in the relational case (Section 1.2). Furthermore, rules-of-thumb-based optimization is overburdened, if an unexpected situation occurs.

1.4 Research Objectives

This thesis has four main objectives and accompanied research questions:

- **Analyzing** Which concepts of relational query optimization can be reused in the context of XDBMSs? Are there new techniques that are novel to XML query optimization and do not have a direct counterpart in relational query optimization?

- **Understanding** What is the impact of the classical and novel concepts of query optimization on the overall query optimization process in general and on specific query optimizers in particular?

- **Modeling** How can we model the system behavior in terms of CPU cycles and IO costs to reflect the actual processing costs of a query?

- **Evaluating** Does our cost model, which serves as a set of hypotheses on the system behavior, reflect the reality? How high or low is the quality of various optimizers equipped with this cost model? How does a particular optimizer configuration behave in different hardware setups?

1.5 Overview

This work consists of four parts:

- **Part I** discusses the related work on relational and XML query optimization. Moreover, it provides an introduction to the *XML Transaction Coordinator* (XTC), our prototype of a native XDBMS, that furnishes the infrastructure for our query optimizer. Finally, it sketches the logical XQuery algebra called *XML Query Graph Model* (XQGM) that serves as input for query optimization.

- **Part II** encompasses the main theoretical contribution of this thesis. First, it introduces selection push-up rules that allow to delay expensive accesses to *fn:text()* functions as long as possible. Moreover, it discusses a plan graph abstraction that is the main data structure being manipulated during cost-based query optimization. Next, we discuss the novel set of query transformation rules that allow to derive semantically equivalent plans. For effective cost estimation, we need reliable cardinality information. This information is gathered using a set of inference rules. Thereafter, we discuss the cost model that helps to assign to each plan graph, which was derived by applying the transformation rules, a cost factor. Finally, we describe our plan generation approach.

- **Part III** provides insight into the implementation of the cost-based query optimization framework and contains the empirical evaluation of our optimization approach.

- Finally, **Part IV** concludes this thesis and points out future research questions.

2 Preliminaries

"Because if you have a strong foundation like we have, then you can build or rebuild anything on it. But if you've got a weak foundation you can't build anything."

(Jack Scalia)

In this chapter, we give a brief overview of important query processing techniques, which are necessary to understand the optimization approach developed in this thesis. Section 2.1 introduces query optimization in database systems. It describes the Query Evaluation Process and the most important components of query optimizers (search strategies, query rewrite rules, and cost formulæ). Next, Section 2.2 shows which new challenges are raised by native XML query optimization. Furthermore, it provides an overview of the various path processing operators (different types of join operators) and the plethora of access paths being available to an optimizer's plan generator. In Section 2.3, we look at the query optimization frameworks of important relational and XML database systems. Thereafter, we analyze and compare these systems with the cost-based query optimizer of XTC. Finally, Section 2.4 concludes this chapter with a short summary.

2.1 Principles of Query Optimization

The query processor is an integral part of every modern database system whose quality directly influences the performance of the whole system (Jarke and Koch, 1984). If we have a look at the layered architecture for database systems, which was proposed by Härder and Reuter (1983), we can see that the query processor bridges the gap between the top-most layers (L 4 and L 5). Here, it mediates between a declarative query language (SQL or XQuery) with its access-path-independent data model and record-oriented access to specific access paths (Härder, 2005).

In conformity with Graefe (1993), we can split query processing into two subtasks: *query optimization* and *query execution*. During query optimization, a declarative query is translated into a tree (or graph) consisting of several

physical plan operators[1]. Thereafter, it is run by the execution engine. In general, finding the optimal plan for a given query is computationally intractable (Jarke and Koch, 1984). Incomplete and imprecise statistics make the situation worse. Finding the optimal join order is one of the fundamental optimization problems in database systems. Ibaraki and Kameda (1984) have shown that this problem is NP-hard. Hence, the term "query optimization" is a bit too promising, because most optimization strategies, which are in most cases cost-based heuristics, tend to restrict their job to reduce costs significantly instead of trying to find globally optimal solutions (Jarke and Koch, 1984). From the point of view of industry expert Pat Selinger, customers are much more satisfied with reliably good plans for a broad range of queries, instead of finding optimal solutions for corner cases (Hamilton, 2008). Pursuant to private communications of the author with senior researchers like Goetz Graefe and Johann-Christoph Freytag, this can be boiled down to the following dictum:

> *"We do not insist on finding the optimal plan anymore. Instead, we just want to omit bad plans."* (private communication).

The notion of query optimization can be further refined: Freytag (1989) identifies two main aspects of query optimization: *query rewrite* and *query translation*. During query rewrite, the optimizer modifies—but keeps intact—the non-procedural description of the query. For example, it simplifies the query or removes redundant parts, for example, by normalizing predicates into conjunctive or disjunctive normal form. On the other hand, query translation employs a search strategy (borrowed from artificial intelligence programming) to generate a plethora of physical alternatives for a query, for example, by applying query transformations such as join commutativity or join associativity. Finally, it maps the most promising plan onto a physical plan, which we refer to as *query execution plan* (QEP) (Jarke and Koch, 1984; Graefe, 1993).

A full-fledged query processing infrastructure uses numerous operators and mechanisms to execute complex queries. In the query processing world, it is quite common to refer to these primitives as *physical algebra* operators (Graefe, 1993). Conceptually, the physical algebra is strongly related to its counterpart—the logical algebra. The logical algebra is dependent on a specific data model (e. g., the relational model for SQL or the XML Query Graph Model for XQuery) and defines which queries are valid in it. Query rewrite modifies logical algebra expressions, therefore, it is often referred to by the more general term *algebraic optimization*.

On the contrary, the physical algebra is completely system-specific. Even though two given systems may implement the same logical algebra, their

[1] In accordance with many publications on query processing, we refer to this graph as plan.

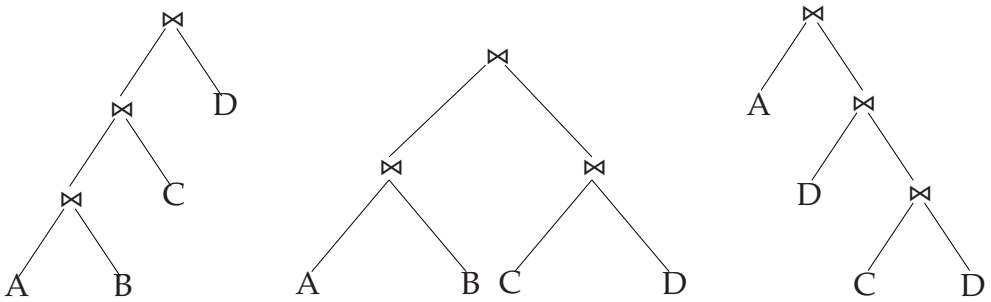


Figure 2.1: Archetypical shapes of query graphs (after Graefe, 1993)

physical algebra may differ significantly. Cost formulæ used during cost estimation are always associated with physical algebra operators, because only a physical algebra operator has a corresponding algorithm in the system. Consequently, to perform cost-based query optimization, a logical plan must be mapped first onto a physical plan, which embodies all physical properties needed to estimated its IO and CPU costs.

Graefe (1993) lists four different types of mappings from logical to physical algebra operators: (1) A single physical operator can implement a cascade of logical operators. For example, a join operator may additionally support the evaluation of selection predicates and projection. (2) A physical operator can only evaluate one part of a logical operator. For example, a duplicate elimination algorithm alone is not sufficient for implementing a relational projection operator. (3) A physical operator might not have a counterpart in the logical algebra. For example, a sort operator does not exist in the relational calculus, because it is unordered. (4) Finally, there are properties that exist in the logical algebra, but not in the physical one. For example, even though joins are commutative in the relational algebra, the nested-loops join algorithm does not handle its inputs equally (Graefe, 1993).

The interface of every physical operator has three operations that allow for a communication between operators: *open()*, *next()*, and *close()*. The *open()* operation initializes the physical operator, for example, by allocating memory for an in-memory sort operator. The *next()* operation returns the results of the physical operator. Finally, *close()* helps to perform clean-up operations, for example, memory deallocation in a hash join operator (Graefe, 1993).

As we have mentioned before, in database systems, queries are normally expressed as trees. We can distinguish between three archetypical shapes of query graphs: *left-deep*, *bushy*, and *right-deep* query graphs. Figure 2.1 shows the three different shapes for four relations A, B, C, and D (Graefe, 1993).

The class of bushy trees encompasses left-deep and right-deep plans. Which types of query graphs are supported by a plan generator determines how large or small the search space is. The larger the search space, the higher the chance that it contains an efficient plan. On the other hand, increasing the plan generator's freedom of action can raise dramatically higher costs for search-space exploration. The designer of a query optimizer must always be aware of this stress ratio. For example, the query optimizer of System R—the first prototype of a relational database system—only supported the creation of left-deep query plans (Astrahan et al., 1976; Selinger et al., 1979).

2.1.1 Query Evaluation Process

As you are now familiar with the fundamental notions of query processing, we can move on to a description of a complete query processing pipeline. In the style of Mitschang (1995), we identify three major steps for an overall query evaluation process: *analysis*, *optimization*, and *code generation*. Figure 2.2 shows the different steps in chronological order.

Analysis

The analysis stage receives a query in a textual representation. At the beginning, this representation is parsed (*syntactic analysis*) and mapped onto an *abstract syntax tree* (AST). During the *semantic analysis* step, the optimizer checks whether the user, which posed the query, has all access privileges needed for evaluating this query and whether there exist violations of integrity constraints, for example, queried relations that do not exist in the database. During *normalization*, selection predicates are transformed into disjunctive or conjunctive normal form (Jarke and Koch, 1984). *Simplification* aims at reducing the complexity of the query by finding redundant predicates. Furthermore, this step helps to identify contradictory predicates that allow the query optimizer to immediately stop optimization, if the query will not return any result. Finally, the query is mapped onto a logical algebra expression (query graph[2]) that serves as input for the optimization stage.

Optimization

The optimization stage performs *query rewrite* and *query transformation*. During query rewrite, several algebraic equivalence rules are applied (*algebraic*

[2] A prominent internal representation for query graphs is, for example, the Query Graph Model (Pirahesh et al., 1992).

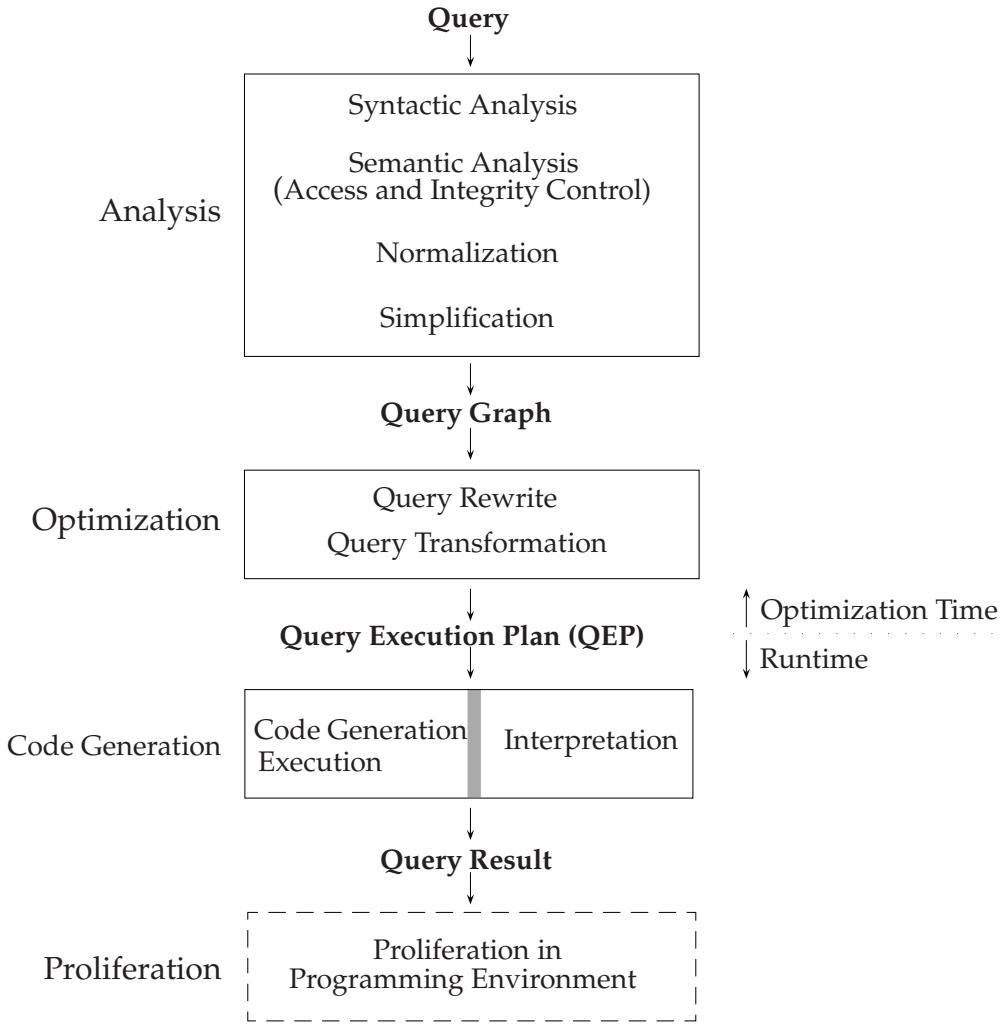


Figure 2.2: Query evaluation process (after Mitschang, 1995)

optimization) to the logical algebra expression (query graph). The main goal of these heuristic rules is the restructuring of the query graph in such a way that:

1. query transformation can be performed more easily. For example, Mathis (2009) introduces several rewrite rules for unnesting XQuery expressions to achieve a more convenient evaluation of them using specialized *n*-ary join operators (so-called twig joins).

2. the restructured version can be evaluated more efficiently, even if no further optimizations would be applied. For example, in the relational

world, *predicate push-down* is a prominent heuristics that assumes that evaluating predicates as early as possible (during retrieving tuples from an access path) helps to speed-up query evaluation (Härder and Rahm, 2001).

So far, all tasks (the whole analysis stage as well as query rewrite) that we discussed, served solely for preparing the query graph for the most challenging job of a query optimizer—*query transformation*.

In Figure 2.3, you can see the query transformation cycle. The search strategy (Section 2.1.2) receives the rewritten query graph and passes it on to the plan generator that creates several alternative plans using a set of transformation rules, for example, by exchanging the inputs of a join operator (join commutativity) or reordering join operators (join associativity). Further alternatives are created by exchanging the implementation of the query graph's subtrees (e. g., a relational join operator could be implemented using a nested-loops join, a hash join, or a sort-merge join). After enumerating alternative plans ($p_1 \ldots, p_n$), they are handed over to the cost estimator. The cost estimator employs a cost model (a set of formulæ describing the costs for low-level operations such as scans or join evaluation). The cost estimator is associated with the statistics component that offers access to value distributions and attribute-value (relational model) or element (XML world) cardinalities to estimate the expected cost[3] of an alternative plan. After assigning each alternative a cost, the pruning component cuts off all plans apart from the one with the lowest cost. Next, the optimizer makes a binary decision whether the optimization goal is already satisfied or not. If the answer is negative, the aforementioned process continues—now, with the currently cheapest plan as input—until no further plans can be generated or another termination criteria is met (e. g., time restriction or no cost reduction after n consecutive optimization cycles).

Conventionally, plan generation is not performed on the query graph itself. Instead, alternative plans are captured by an internal data structure called *search tree* (Freytag, 1989). The final step of query optimization is now mapping the optimal plan onto a QEP. This task can be easily performed by a left-most depth-first traversal of the optimal plan and a 1:1 translation of it to a graph of physical algebra operators. Finally, the QEP is passed on to the execution engine.

[3] The measure of cost is strongly dependent on the optimization goal. For example, if we want to achieve maximum throughput, CPU and IO costs must be minimized. On the other hand, if we want to minimize energy consumption, operators that consume the lowest amount of energy should be preferred.

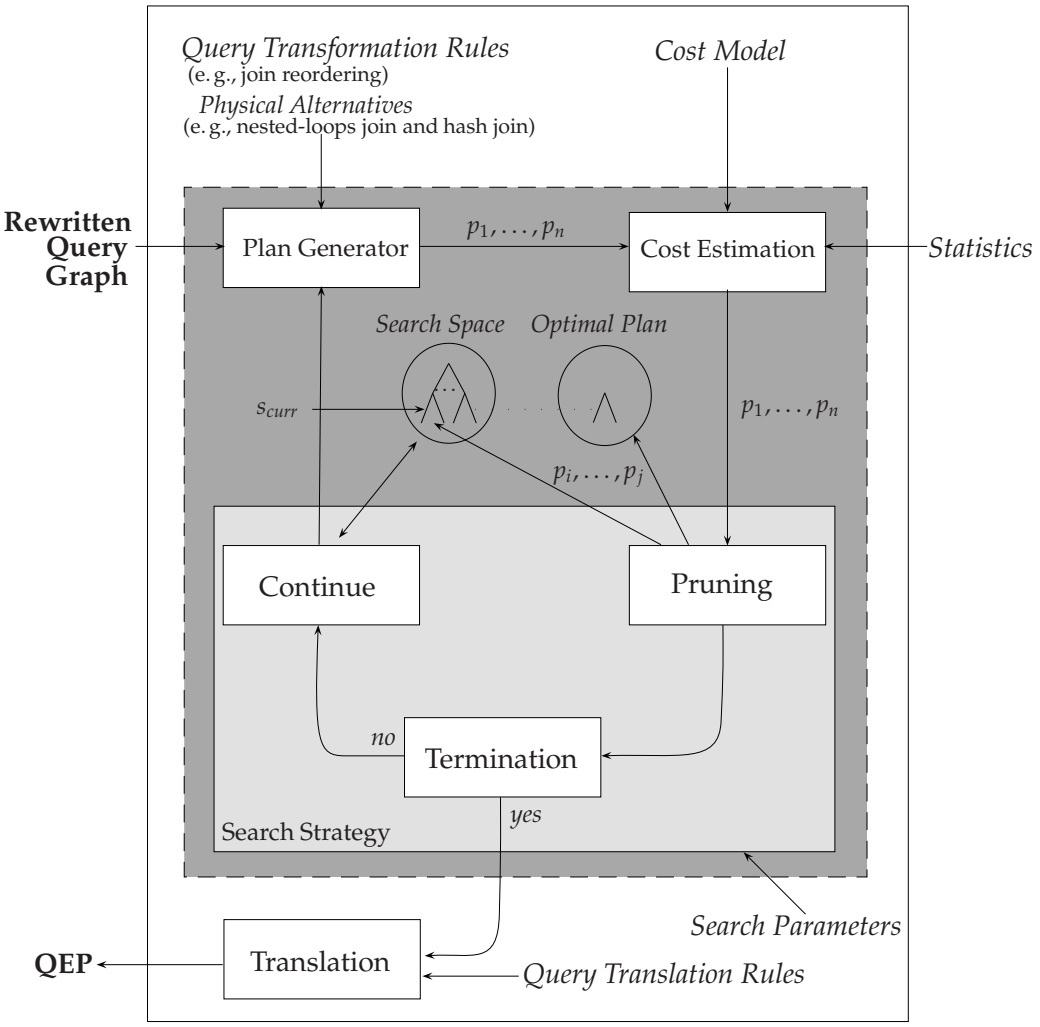


Figure 2.3: Query transformation cycle (adapted from Härder and Rahm, 2001)

Query Execution

The query execution engine receives a QEP and has two options for executing it: *code generation* or *direct interpretation*. Code generation creates a module out of the QEP before executing it. This option is especially meant for queries that occur repetitively in the system, because we can spare the expensive task of query optimization and just execute the module over and over again[4]. Especially for ad-hoc queries, direct interpretation is the method of choice,

[4]Please note, this is only possible if there were no updates on the database or any changes in the statistics. Otherwise, there might exist a QEP that is more efficient than the one compiled into the module.

because generating a module—which is probably never used again—is too expensive.

Proliferation

The final step of query processing is proliferation. Here, the query result is exposed to a programming environment or can serve as input for a nested query. In native XML database systems, this step is costly, because the QEP mostly produces sequences of node identifiers and does not output XML subtrees. During proliferation, which we call in the context of XML databases *materialization*, these identifiers must be resolved by accessing the queried XML document (see Section 2.2).

2.1.2 Strategies for Search-Space Exploration

In Section 2.1.1, we argued that the search strategy plays a major role in query optimization, that is, it provides the "heartbeat" for the query optimizer, because it dictates how the search space is explored for alternative plans. Moreover, it determines whether a sufficient solution has been found. In general, we can distinguish between two classes of strategies: *exhaustive search* and *probabilistic search*.

Exhaustive Search

Exhaustive strategies perform a bottom-up full enumeration of the search space and prune expensive subtrees as soon as possible. Therefore, it is guaranteed that they find the optimal solution. Nevertheless, though the search space grows quadratically with the number of joins to be reordered, in practice, this approach is only applicable if a limited number of joins (10–15) is involved (according to Ioannidis, 1997).

To give you an impression how exhaustive search works, we have a brief look at the search strategy of System R (see Section 1.3). Its dynamic programming algorithm generates a search tree using a bottom-up approach for query graphs consisting of access paths as leaf nodes and join operators as inner nodes. The algorithm starts at the leaf nodes of a query graph. For every access path that is associated with a relation, it generates a possible access plan. For each leaf node, only the cheapest access path is retained. Next, the algorithm proceeds to the next level in the query graph and generates all

interesting join orders[5] using the cheapest access paths from the first step. Next, all interesting orders are generated for two consecutive joins and so on, until the root of the query graph is reached, whereupon in every step only the cheapest join order is kept.

Probabilistic Approaches

If very large join trees (more than 15 join operators) shall be optimized, we have to give up exhaustive search. Instead, it is quite common—as in artificial intelligence programming, too—to use probabilistic methods (Russell and Norvig, 2003) for top-down optimization. Probabilistic approaches do not guarantee to always find the optimal solution. Nevertheless, because join reordering is NP-hard, it is our only chance to get at least near-optimal plans for large join graphs. Every solution to a combinatorial optimization problem can be expressed as a state in a solution space, for example, a graph where each vertex represents a valid solution. Every state is annotated with a cost factor that is estimated using the cost model. To achieve their goal, finding the state with the globally minimal cost, probabilistic (also called randomized) algorithms take *random walks* through the search space. Each walk is formed by a sequence of *moves* from one state to another one. After each walk, the costs are recalculated. We call a state s a neighbor of state s', if and only if s' can be reached from s via a single move. We distinguish between *up-hill moves* and *down-hill moves*. A state transition is called an up-hill move (down-hill move), if the goal state has higher (lower) costs than the original state. We say a state is a *local minimum*, if there is no neighbor state that can be directly reached via a down-hill move. Instead, if a state has no neighbor that can be reached using an up-hill move, we call it a *plateau*. Finally, a state whose cost is lower than the cost of any other state is referred to as *global minimum* (Ioannidis and Kang, 1990).

In the literature, there exist many probabilistic search algorithms (compare Russell and Norvig, 2003). In the database context, the most relevant strategies are: Iterative Improvement, Simulated Annealing, and 2-Phase Optimization (Ioannidis, 1997).

Ioannidis and Kang (1990) introduce *Iterative Improvement* (Algorithm 2.1). They assume an initial random state S_∞ with maximum cost ∞. The function cost(S) returns the cost of state S. Using the function neighbor(S), we get all neighbor states of S generated by applying transformation rules. The

[5] The concept of interesting join order plays an important role in the System R optimizer, because it restricts it to only generate left-deep query plans without introducing additional Cartesian products (Selinger et al., 1979).

```
1  S_min = S_∞;
2  while ¬(stopping condition) do
3  |  S = random state;
4  |  while ¬(local_minimum(S)) do
5  |  |  S' = random state in neighbors(S);
6  |  |  if cost(S') < cost(S) then
7  |  |  |  S = S';
8  |  |  end
9  |  end
10 |  if cost(S) < cost(S_min) then
11 |  |  S_min = S;
12 |  end
13 end
14 return S_min;
```

Algorithm 2.1: Iterative Improvement (Ioannidis and Kang, 1990)

inner loop of Algorithm 2.1 (lines 4–9) performs down-hill moves as long as the current cost ($S_{\min}$) is reduced and no local minimum is found. Local optimization continues until the termination criteria is satisfied, for example, consecutive local optimizations do not lead to a further cost reduction. Finally, the local minimum with the lowest cost is returned as best plan. According to Ioannidis and Kang (1990), the probability that the algorithm also finds the global minimum converges to 1.0, though, the quality of the result is strongly dependent on the cost model.

In contrast to Iterative Improvement, where the search algorithm accepted only down-hill movements, *Simulated Annealing* (Ioannidis and Wong, 1987) additionally permits up-hill moves with a certain probability to avoid getting stuck in local cost minima (Ioannidis and Wong, 1987; Ioannidis and Kang, 1990). The inner loop (Algorithm 2.2, lines 5–17) consists of several *stages*. Every stage is executed using an arbitrarily but consistently chosen parameter T (temperature). The temperature determines the probability, which is calculated using the formula $e^{-(\Delta C/T)}$, that up-hill moves are accepted. Here, ΔC is the cost difference between the original state and its neighbor. Every stage ends, if an equilibrium[6] is reached. Next, the temperature value is reduced using reduce(T), for example, by 5%. Thereafter, the search continues with the next stage. Over time, the temperature decreases monotonically until it is equal to 0 (frozen).

[6] For example, Ioannidis and Wong (1987) define the equilibrium as a fixed number of re-executions without cost reduction, which are proportional to the total number of joins involved.

```
1  S = S_0;
2  T = T_0;
3  S_min = S;
4  while ¬(frozen) do
5  |   while ¬(equilibrium) do
6  |   |   S' = random state in neighbors(S);
7  |   |   ΔC = cost(S') − cost(S);
8  |   |   if ΔC ≤ 0 then
9  |   |   |   S = S';
10 |   |   end
11 |   |   if ΔC > 0 then
12 |   |   |   S = S' with probability e^−(ΔC/T);
13 |   |   end
14 |   |   if cost(S) < cost(S_min) then
15 |   |   |   S_min = S;
16 |   |   end
17 |   end
18 |   T = reduce(T);
19 end
20 return S_min;
```

Algorithm 2.2: Simulated Annealing (Ioannidis and Wong, 1987)

Ioannidis and Kang (1990) showed that the probability of finding the global minimum using Simulated Annealing also converges to 1.0. To allow the algorithm to terminate in finite time, the parameters must be chosen wisely—a task that turns out to be error-prone in practice.

Finally, *2-Phase Optimization* (Ioannidis and Kang, 1990) combines Iterative Improvement and Simulated Annealing. In an initial step, a locally optimal solution S_{II} is determined using Iterative Improvement. In the second step, S_{II} serves as input for Simulated Annealing and replaces the previously chosen initial state S_0. Furthermore, the temperature T is initialized with a much smaller value than before—$T = 0.1 \cdot \text{cost}(S_{\text{II}})$—, to restrict Simulated Annealing's movements to the neighborhood of S_{II}. Ioannidis and Kang (1990) have shown that this approach can provide better performance and quality compared to exclusively using Iterative Improvement or Simulated Annealing.

2.1.3 Cardinality and Cost Estimation

In Section 2.1.1, we said that cost is the main criteria for restricting the search space for query optimization. Hence, the optimizer's ability to propose good plans is strongly dependent on the quality of the cost estimates provided by

the cost model. For every operator in a query graph, we are mainly interested in finding answers to the following questions:

1. Given a number of input tuples for the operator, how many output tuples can we expect? (*cardinality estimation*)
2. How much cost is raised for processing the inputs? (*cost estimation*)

A database management system maintains statistical information for each relation (RDBMS) or document (native XDBMS). Using such statistical summaries, for example, histograms (Chaudhuri, 1998), and simplifying assumptions, for example, a uniform distribution of values or statistical independence of predicate values in conjunctive clauses, the query optimizer can provide satisfying answers to the first question.

To answer the second question, the query optimizer uses the statistical information and the *cost model*, which is consisting of a set of formulæ describing the execution cost of each physical operator with respect to its inputs[7].

Statistical Profiles

According to Mannino et al. (1988), *statistical profiles* are compact data structures for managing relevant quantitative descriptors (e. g., standard deviation, minimum, or maximum of an numeric attribute value) in a database. We can distinguish between two types of profiles: *base profiles (BP)* and *intermediate profiles (IP)* (Mannino et al., 1988).

Base profiles are accurate and associated with "real" objects that physically exist, such as relations (RDBMS) or documents (XDBMS). On the other hand, intermediate profiles contain estimated information for intermediate objects and do not physically exist, for example, tuple streams created by joining two relations (RDBMS) or several element node streams (XDBMS).

Every leaf node in a query graph (e. g., an Access operator) has its own BP that reflects its statistical properties according to the metadata catalog of the database system, for example, the total number of pages that are consumed or the total number of distinct attribute values.

Intermediate profiles are associated with the inner nodes of query graphs. They provide estimated information about the output of intermediate operators that do not necessarily receive their inputs directly from access paths, for example, for a join operator, they provide the estimated cardinality of the

[7] Please keep in mind that cardinality estimates are always invariant for semantically equivalent plans. This statement does not hold for cost estimates, because cost is solely dependent on the physical characteristics of an operator that typically changes from one physical alternative to another one (Chaudhuri, 1998).

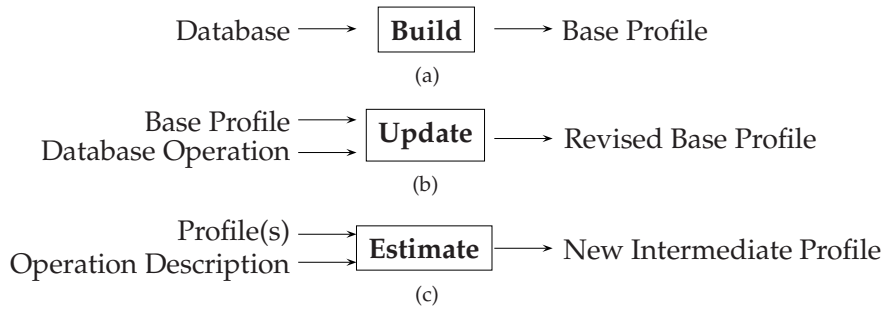


Figure 2.4: Statistical profile operations (after Mannino et al., 1988)

join result, whose calculation is based on the information furnished by (base or intermediate) profiles associated with its input operators.

Figure 2.4 shows the three basic operations that can be applied to base and intermediate profiles: *Build*, *Update*, and *Estimate* (Mannino et al., 1988). The Build operation, which is triggered by the database administrator's call to a statistics collection tool, creates BPs based on a lookup to the database catalog, sampling, or histograms. If the database changes over time, the Update command recalculates property values of BPs, for example, the average value, minimum, or maximum of numeric attributes. Such updates are performed automatically—but only for the most primitive properties. For more complex properties, the Build operation must be run again. Finally, the query optimizer uses the Estimate operation to derive IPs, which are needed for cardinality and cost estimation.

Cardinality Estimation

In database systems, the cardinality of intermediate results is primarily determined by the "selectiveness" of predicates, that is, how many tuples of the input stream satisfy the predicate. In turn, by estimating the selectivity of a predicate, we can estimate the expected cardinality of the intermediate result:

Let $s(p) = |\sigma_p(R)|/|R|$ be the selectivity of predicate p, where $|R|$ is the cardinality of the base relation R and $|\sigma_p(R)|$ is the cardinality of the intermediate result after applying p on it. Hence, we can calculate the cardinality of the intermediate result by: $|\sigma_p(R)| = s(p) \cdot |R|$.

Exact cardinality estimation is a complex task. Since the beginning of cost-based query optimizers (see Selinger et al., 1979), their cardinality estimation frameworks have been strongly relying on simplifying assumptions, for example, a *uniform distribution* of attribute values or the *statistical independence*

of attribute values (Selinger et al., 1979; Mannino et al., 1988). Based on these assumptions, Selinger et al. (1979) define a set of simple formulæ that can be used for selectivity estimation. If there is no statistical information, they use a set of default values as selectivity estimates. For example, for an equality predicate, a selectivity of $1/10$ is assumed.

Honestly, the simplifying assumptions are not always met in reality[8]. For example, if we consider an employee table with an *age* attribute, its values are not uniformly distributed, because the vast majority of employees are midagers (e. g., between 28 and 45) and only a few are young (< 20) or mature (> 60). To overcome this deficit, more accurate value distributions can be gained by building *histograms*[9] for relevant attributes (compare Mannino et al., 1988; Chaudhuri, 1998).

Especially the statistical independence of attribute values is error-prone. Let us have a look at the following predicate, which shall be evaluated on an employee table: $(age \leq 25) \wedge (work\ experience > 10)$. The attribute values are not statistically independent, because only few young employees can have such a long work experience. To model the value distribution of statistically dependent attributes, multi-dimensional histograms can be used (e. g., Muralikrishna and DeWitt, 1988; Poosala and Ioannidis, 1997).

Cost Model

For every physical operator in a database system, there exists a cost formula that describes its execution cost for a given input. Altogether, the set of cost formulæ is called the *cost model*. In throughput-oriented systems, execution costs are primarily determined by a weighted sum of IO and CPU cost, where the weight w allows to adapt the estimation to CPU-bound or IO-bound systems:

$$\text{Cost}_{\text{total}} = \text{Cost}_{\text{IO}} + w \cdot \text{Cost}_{\text{CPU}}$$

Here, the IO cost is assumed to be proportional to the total number of page fetches that must be performed to load all relevant pages into the database buffer. In contrast, the CPU cost is measured as the total number of calls to the physical operator's *next()* function.

Statistical profiles are only one ingredient to allow for cost estimation. More precisely, it is impossible without additional information on the characteris-

[8] Recently, new approaches for improving statistical estimation accuracy can be observed (compare Chaudhuri, 2009; Beyer et al., 2009).

[9] A histogram divides the domain of an attribute value into a constant number of buckets. Only within buckets, a uniform distribution of values is assumed (Ioannidis, 1997). There exists a plethora of different histogram types whose taxonomy is given by Poosala et al. (1996).

Query 2.1 A simple XQuery expression

```
let $auction := doc("auction.xml")
return for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

tics of physical operators. For example, some operators need sorted input streams (e. g., sort-merge join) that must be built at additional cost or a hash join operator can only be used if the join predicate involves only equality predicates. Graefe and DeWitt (1987) call such characteristics *physical properties*. We call the universe of all physical properties for a given physical operator a *physical profile*, whereas statistical profiles can be considered *logical profiles* in this context.

2.2 Native XML Query Processing

Today, XQuery (Boag et al., 2007) is the predominant query language in native XML database systems. XQuery is a hybrid of a declarative query language, describing *what* should be searched for, and a functional programming language, specifying *how* the result is retrieved. Even if we only consider XQuery as a stand-alone language, it is fairly complex to optimize. Traditionally, XQuery is normalized into the XQuery Core Language (Draper et al., 2007), a minimal subset of XQuery that specifies its formal semantics. Even though we gain a proper definition of its semantics, some important information for query optimization (e. g., path expressions) are becoming intransparent. For example, let us consider Query 2.1 that selects the name of all *person* nodes having an *id* attribute whose value is equal to *"person0"*, where you can easily see the specification of the path to *person* nodes.

After normalizing and slightly simplifying this query, we get the expression depicted in Query 2.2. Now, it is not so easy anymore to identify the path expression. In the context of query optimization in native XDBMSs, where we want to exploit efficient access paths (e. g., path indexes), this is a big problem, because it is in general not trivial to reconstruct the actual path. Moreover, XQuery relies on a node-at-a-time processing paradigm (nested *for*-loops), whereas query processing in database systems is traditionally set-at-a-time processing.

To overcome both deficits, there exist several approaches for transforming XQuery expressions into a representation where (1) the query optimizer can effortless detect path expressions by applying unnesting rules that annihilate

Query 2.2 Normalized version of Query 2.1

```
for $b in fn:distinct-doc-order(
 for $fs:dot in doc("auction.xml")
  return fn:distinct-doc-order(
   for $fs:dot in child::site
    return fn:distinct-doc-order(
     for $fs:dot in child::people
      return for $fs:dot in child::person
     where fn:data(attribute::id) = person0
     return $fs:dot
)))
return fn:distinct-doc-order(
 for $fs:dot in $b
  return fn:distinct-doc-order(
   for $fs:dot in child::name
   return child::text()
 ))
```

path nestings introduced by the normalization of XQuery and (2) set-at-a-time processing is supported for most parts of an XQuery expression (Mathis, 2009). Nevertheless, being "forced" to destroy important information in the first place and costly reconstructing it in the second step, seems to be absolutely cumbersome.

2.2.1 The Challenges of XML Query Optimization

Even though many architectural aspects of native XDBMSs are inherited from their relational predecessors (e. g., its five-layered architecture), there are novel challenges that make XML query processing in general and XQuery optimization in particular an even more interesting and challenging task.

Schema Evolution

According to Loeser (2008), flexibility, especially in terms of rapid schema evolution, is one of the main sales arguments for introducing XML databases to enterprise information management. Hence, we can expect that the schemas of XML documents will change frequently. Schema evolution may occur within a document (Balmin et al., 2006) or on a document-to-document ba-

sis. In the worst case, there exists no schema at all. In this situation, an approximation of the schema can be provided by summarization techniques like *Data Guides* (Goldman and Widom, 1997). Having no schema information or only a rough approximation of it, makes query processing more difficult. For example, for element cardinality estimation, a schema making use of the XML Schema (Fallside and Walmsley, 2004), language primitives *minOccurs* and *maxOccurs* could provide exact lower and upper bounds of element occurrences on a specific path. If there is no schema, the optimizer must rely on estimated information that is sometimes error-prone.

Heterogeneity

In the relational world, due to the simplicity of the data model with its homogeneous rows, value-based cardinality estimation works fine. As XML explicitly supports heterogeneity, two documents—let us refer to them as d_1 and d_2, respectively—complying with a given schema do not necessarily share the same structure, e. g., an element e can occur in d_1 several times, but never in d_2. Furthermore, XML does not support explicit *NULL* values for elements (Balmin et al., 2006).

In addition to querying XML documents, XQuery allows to process and generate arbitrary nestings of sequences using its *let* and *for* language features. Especially for *let*, where the length of the bound sequence may vary in every iteration over a related and *for*-quantified sequence, cardinality estimation is hindered.

Hierarchical Data Model

XML is based on a hierarchical data model, where structural relationships, for example, *child* and *descendant*,—in addition to classic value-based relationships—play an important role; especially during query evaluation. Therefore, an XML query optimizer has to deal with value-based joins as well as with structural joins and must arrange them in an at least near-optimal way. Ibaraki and Kameda (1984) have shown that finding the optimal join order for value-based joins in relational database systems is NP-hard. Now, by additionally taking structural joins into consideration, the query optimizer must inspect an even larger search space.

Query processors in relational database systems and in XML database systems make generous use of indexes for speeding-up query evaluation. The hierarchical data model of XML allows to define several kinds of indexes on element names, attribute values, complete paths in the document, or even

on the content residing on specific paths. The index selection problem—finding an optimal set of indexes that provides maximum speed-up for query processing—is an NP-complete problem, too (Piatetsky-Shapiro, 1983). Due to the tremendous diversity of XML index types, this problem becomes even more challenging in XML database systems.

Physical Operators—A Plethora of Possibilities

In addition to value-based predicates, XQuery (and its subset XPath) introduce structural predicates. Each XPath path expression is a sequence of structural predicates. For example, the path expression *a/b/c* qualifies all *c* nodes that are on the path *a/b*. In XDBMSs, structural relationships are evaluated using *Path Processing Operators* (PPOs). The simplest type of a PPO is a *Navigation* operator that evaluates the structural predicate similar to a relational nested-loops join. More advanced PPOs belong to one of the following operator classes: *Structural Joins* (SJs) (e. g., Al-Khalifa et al., 2002) and *Holistic Twig Joins* (HTJs) (e. g., Bruno et al., 2002).

Structural joins decompose each path expression into n binary relationships, evaluate each of them separately, and finally "stitch" their results together. On the other hand, HTJs are capable of evaluating path expressions (and even more complex tree patterns) using a single n-way join operator. Both classes of operators must provide their results in document order and sometimes need to perform duplicate elimination which additionally increases the complexity of these operators. Compared to classical value-based join operators, like nested-loops join or sort-merge join, estimating the CPU costs of SJ or HTJ operators is hard and needs much more effort and empirical analysis.

Efficient XML query processing is impossible without tailor-made indexes. XML provides many opportunities for indexing: (1) *Content indexes* provide access to *text()* nodes being leaf nodes of an XML document. Moreover, they provide efficient access to attribute values. In most cases, content indexes are implemented as classical B*-trees. A more general type of content index is the *full-text index* that provides access to the atomization of complete subtrees. (2) *Element indexes* (e. g., Bruno et al., 2002; Chien et al., 2002; Rao and Moon, 2003; Jiang et al., 2003) allow to index all element nodes of an XML document. Element indexes can be used by SJs and HTJs as efficient access paths. (3) *Path indexes* (e. g., McHugh and Widom, 1999; Beyer et al., 2006; Mathis, 2009) can be considered as materialized views on certain paths of XML documents. (4) Finally, *Content-and-structure indexes* (CAS) (e. g., Rizzolo and Mendelzon, 2001; Wang et al., 2003; Kaushik et al., 2004; Mathis et al., 2009) allow hybrid indexing of content and structure at the same time.

The plethora of access paths increases the search space for a cost-based optimizer tremendously, because they are not only providing input streams for join operators, e. g., by using element indexes, but may also completely replace them by path indexes or CAS indexes. Consequently, such advanced indexes are competing with SJs and HTJs for a first-class citizenship, because they can replace parts of—or even complete—cascades of join operators.

2.2.2 Historical Note

Having a look at the long history of relational database systems and the rather short history of XML database systems reveals an interesting common ground. Query processing in both types of systems started using a bottom-up approach (e. g., Jarke and Koch, 1984; Al-Khalifa et al., 2002; Bruno et al., 2002): there was a strong emphasis on efficient evaluation algorithms and processing strategies for interesting queries rather than on providing a complete picture of query processing. Quickly, bottom-up approaches met their limits in both system classes. As a result, a top-down approach consisting of four stages emerged for relational database systems (Jarke and Koch, 1984) that can also serve as trail blazer for XML query processing:

1. Development of an internal query representation that allows to easily express all relevant language constructs
2. Introduction of logical query transformations that allow to standardize, simplify, and prepare the internal representation for efficient evaluation
3. Generation of different implementation alternatives for the internal representation
4. Estimation of the costs of every implementation alternative and selection of the most promising one

In the context of native XDBMS, the first and most parts of the second step have been addressed in previous works (e. g., Mathis, 2009). The work at hand improves the second step and contributes novel concepts for the third and fourth step to provide a complete and efficient cost-based query optimization infrastructure for XQuery.

2.3 Related Work

This section discusses important concepts contributed by seminal research projects on cost-based query optimization. First, in Section 2.3.1, we have a

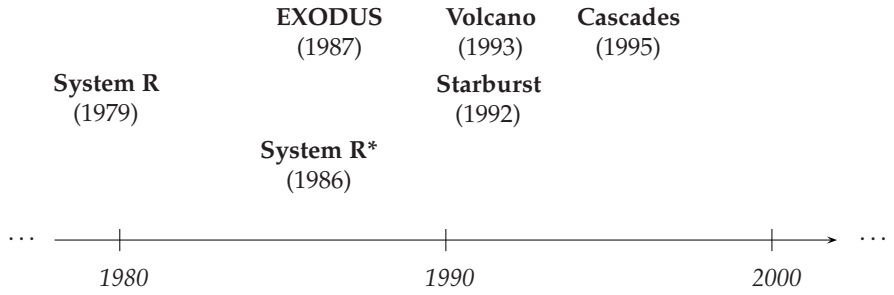


Figure 2.5: Evolution of query optimizers

look at important relational query optimizers. Thereafter, we show in Section 2.3.2, which progress has been made so far in optimizing XML queries. Finally, in Section 2.3.3, we compare our query optimization technique with previous XML query optimization approaches.

2.3.1 Query Optimization in Relational Database Systems

Figure 2.5 shows how the different research prototypes emerged over time. Most interestingly, research on modern query optimizers in database systems did not start in academia. Essentially, it started in industry. More precisely, the success story began at IBM San Jose Research (nowadays, it is called IBM Almaden Research Center) with the *System R* project—the progenitor of relational database systems (McJones, 2011). Additionally, IBM started *System R** as well as the very influential *Starburst* project. The second branch of prototypes (*EXODUS*, *Volcano*, and *Cascades*) was advanced by contributions of famous academic researchers like Michael J. Carey, Goetz Graefe, and David J. DeWitt[10].

System R

Selinger et al. (1979) described the first cost-based query optimizer, which was part of *System R*—the prototype of the first relational database system. The optimizer was capable of optimizing simple and linear SPJ (select, project, and join) queries. The authors introduced a simple cost model based on weighted IO and CPU costs and used statistics on the number of data pages consumed

[10] Even though this section will discuss the most important concepts of relational query optimization, providing the complete picture of 30+ years of research is out of scope of this work. You will find more detailed information in important surveys (e. g., Jarke and Koch, 1984; Chaudhuri, 1998; Graefe, 1993; Hameurlain and Morvan, 2009; Moerkotte, 2009).

by relations to bind the cost model's variables to concrete values. Their dynamic programming algorithm initially selects optimal operator fittings for access paths. Thereafter, an optimal join order is determined based on a local optimality assumption. To early prune the search space, not all possible enumerations are taken into account. Instead, they only focus on *interesting join orders*, that is, orders that do not require additional introductions of Cartesian products.

System R*

The *System R** project (Lohman et al., 1985; Mackert and Lohman, 1986; Kacimi and Neumann, 2009) focussed on the development of a distributed query processing infrastructure. Here, relations are scattered over several sites. System R* was capable of handling distributed queries—queries that involve tables from different sites. The site where the query is issued is called *master site*. All query optimization tasks are also distributed over the affected sites. The master site is responsible for all inter-site decisions (e. g., it selects the optimal join site and chooses the best shipping method), whereas inner-site optimization (e. g., join reordering and access path selection) is performed by local optimizer instances. Even though the System R* optimizer is mostly based on the cost-based optimizer of System R, there are several differences and enhancements: In System R*, the CPU costs are modeled more precisely than in System R. Instead of just approximating the CPU costs by counting the total number of calls to the record-level storage, the total number of instructions is used as a more precise measure. Furthermore, the cost model consists now of four components. Besides CPU and IO costs, System R* considers communication costs (total number of bytes to be transfered and total number of messages), too. Moreover, the cost model allows to access each cost component individually to allow for more fine-granular decisions.

Starburst

Haas et al. (1989) elevated query processing to the next level of abstraction. The primary goal of the *Starburst* project was to easily allow to extend the system by adding new language constructs, data management capabilities, and language processing features.

One of the pillars of the Starburst prototype was the novel query language processor called *Corona*. As an internal query representation for Corona, the authors introduced the so-called *Query Graph Model* (QGM) (Pirahesh et al., 1992)—an extended relational algebra with a strong emphasis on structural

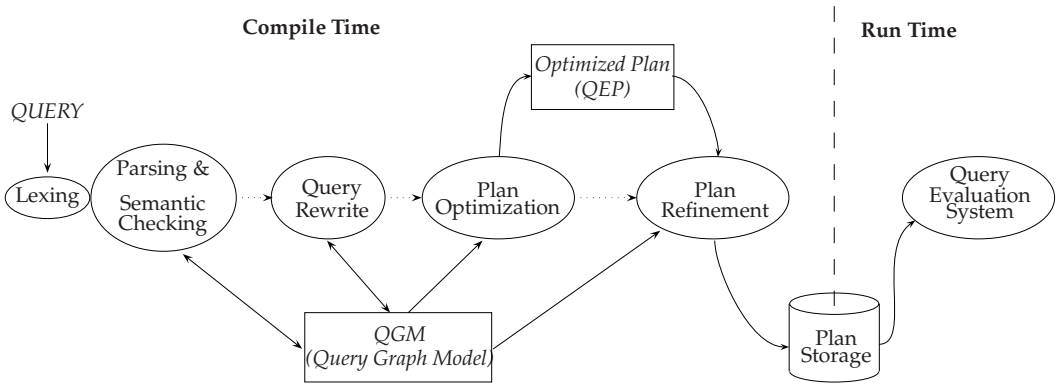


Figure 2.6: Query evaluation in Starburst (according to Haas et al., 1989)

relationships between language constructs. QGM allows to express queries as high-level operations on tables. The most important operation in QGM is called *SELECT*, which allows to express selection predicates, projections, and joins. Using QGM, new language features can be seamlessly integrated[11].

In the context of System R and System R*, query optimization primarily meant join reordering and access path selection (non-algebraic optimization). In Starburst, *query rewrite* as a way of optimizing queries at the logical (algebraic) level was taken into consideration, too. Starburst supports three classes of query rewrites for QGM: migration of predicates, projection push-down, and merging of operators (Haas et al., 1989). Every query rewrite rule is specified as a transformation rule (described using the programming language C) consisting of a *condition* (qualifying a QGM graph) and an *action* part (specifying how to perform the modification). Using Starburst's flexible rule engine, a set of rewrite rules can be applied to QGM. According to Pirahesh et al. (1992), even advanced features like the optimization of recursive queries and magic set transformations are supported. Figure 2.6 shows the Starburst query evaluation process and illustrates how query rewrite seamlessly integrates into the overall query evaluation process.

Starburst also uses a rule-based approach for plan generation. Thereby, it employs grammar-like production rules (Lohman, 1988) that are called *strategy alternative rules* (STARs). The terminals of these rules are formed by so-called *low-level plan operators* (LOLEPOPs)[12]. By redesigning the plan generator, more flexible search-space exploration became possible. For example, the

[11] Most notably, IBM managed to integrated XQuery—a completely different language—into QGM, even after being more than 20 years available to the market (Beyer et al., 2005).

[12] In Section 2.1, we referred to LOLEPOPs by the more general term physical operators.

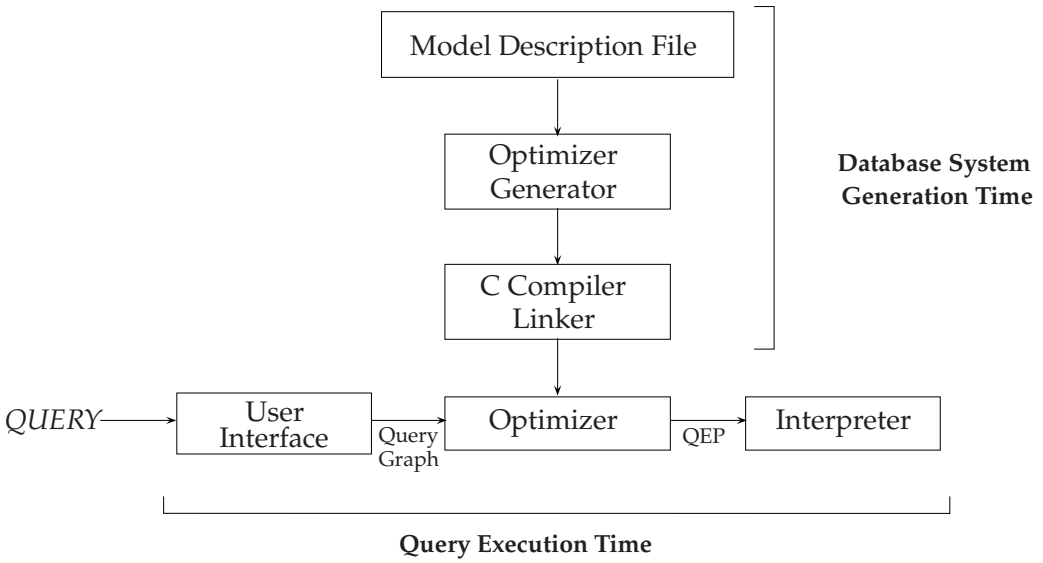


Figure 2.7: Query optimization in EXODUS (after Graefe and DeWitt, 1987)

plan enumerator provided tuning knobs for modifying several parameters for restricting or enlarging the search space by prohibiting or allowing bushy trees, respectively.

EXODUS

As indicated by its name, the goal of the *EXODUS* project[13] was to overcome the traditional thinking of monolithic systems in database research. EXODUS provides a generic framework for generating modular and application-specific database systems. Hence, EXODUS provides a complete data management infrastructure that is not tied to a specific data model.

For extensible query evaluation, EXODUS provides a rule-based optimizer generator (Graefe, 1987; Graefe and DeWitt, 1987) that helps to assemble and compile query optimizers for arbitrary algebraic query languages. Figure 2.7 sketches the EXODUS optimizer generation approach. The optimizer generator receives a *model description file*, which describes the supported logical and physical operators as well as sets of transformation and implementation rules. Based on this description and further parameters (e. g., the actual search

[13] Probably, for paying attention to the *Zeitgeist* of the late 80s, where object-oriented database systems were en vogue, EXODUS may also be an abbreviation for **EX**tensible **O**bject-Oriented **D**atabase **S**ystem (Carey et al., 1990).

strategy), the *optimizer generator* creates code written in the C programming language. Finally, the C compiler produces an executable optimizer.

Volcano

Graefe and McKenna (1993) promoted the *Volcano* project as an advancement of EXODUS that followed its optimizer generation paradigm (Figure 2.7). Besides several improvements in the search engine (plan generation using backward chaining) allowing for more efficient plan generation, their concept provides a clear separation between the logical and the physical algebra—as we have already discussed in Section 2.1. Furthermore, they introduce the distinction between logical properties (e. g., cardinality) that can be derived from the logical algebra and physical properties (e. g., sort order) that are operator-specific. Whenever the plan generator creates a new (intermediate) state, a *physical property vector* is attached that encapsulates all physical properties. Volcano uses so-called *enforcers*, that is, operators that are only present in the physical algebra—but unavailable in the logical algebra—, to provide semantically correct QEPs, for example, by enforcing the plan generator to sort the inputs of a sort-merge join operator.

Graefe (1990) introduces so-called *exchange modules* that serve as a capsule around physical operators allowing to parallelize their execution without changing their implementation (operator-level parallelism)[14]. Every exchange module provides an exchange iterator whose interface is compliant with the open-next-close protocol (see Section 2.1).

Exchange modules allow for realizing pipeline parallelism (vertical parallelism) and intra-operator parallelism (horizontal parallelism). The exchange operator acts as façade for the subtree below it, for which it creates a new process. By following the classical producer-consumer pattern, the exchange operator's iterator running in a parent process consumes tuples via interprocess communication produced by the producer (child) process, which, in turn, concurrently executes the subtree below the exchange operator.

Figure 2.8 shows a simple query graph with two join operators that is enriched with exchange modules (XCHG operator). Here, the scans are performed in parallel (pipelining). If we assume that the top-most join operator is implemented as sort-merge join, the merging of the different runs can be performed in parallel, too (intra-operator parallelism).

[14] By using exchange modules, the paradigm of demand-driven (lazy) execution is given up in favor of dataflow-driven (eager) execution.

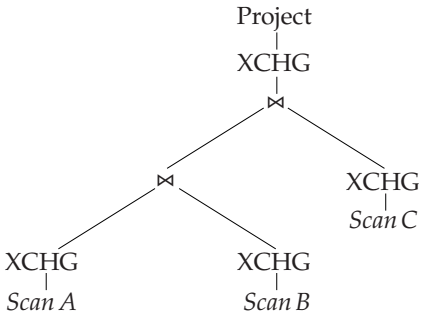


Figure 2.8: Query graph with exchange modules (adapted from Graefe, 1990)

Cascades

The experiences made in the EXODUS and the Volcano projects converged into the *Cascades* framework[15] (Graefe, 1995). Besides a more robust and object-oriented redesign of the framework architecture, the generator paradigm of its predecessors was abolished, because it proved to be too cumbersome. As a consequence, a query optimizer is now created by implementing abstract base classes rather than compiling a model description file into programming language code.

2.3.2 Query Optimization in XML Database Systems

In this section, we introduce several XML database systems (providing native or relational XML storage) and discuss their query processing capabilities. Figure 2.9 on page 32 illustrates the temporal emergence of the different systems, which we will discuss consecutively.

Lore

In 1997, the *Lore* (**L**ightweight **O**bject **RE**pository for Semistructured Data) prototype (McHugh et al., 1997), which was developed at Stanford University, made its way to become the ancestor of XML database management systems, even though it did not use XML as a data model; but, instead, used the *Object Exchange Model* (OEM), which is though closely related to XML.

Abiteboul et al. (1997) introduced *Lorel* (Lore Language) as a declarative query language for Lore. For the efficient evaluation of Lorel expressions, the classic work of McHugh and Widom (1999) describes cost-based query optimization in Lore. Their optimizer relies on a simple cost model that

[15] Cascades also provided the basis for the query optimizer of Microsoft's SQL Server (Graefe, 1996).

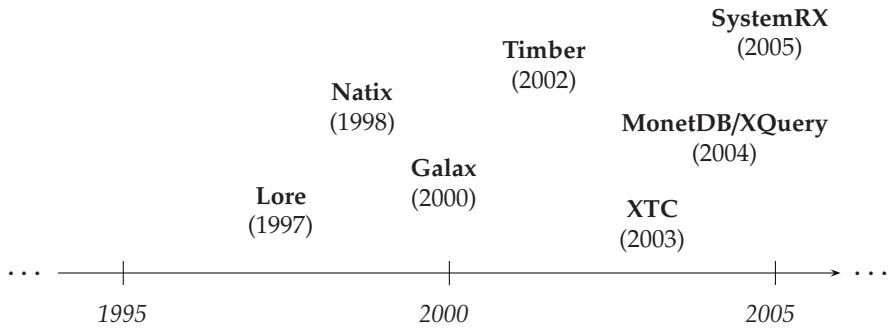


Figure 2.9: The short history of XML databases

considers IO and CPU costs for the cost estimation of alternative plans. Besides statistics provided by the system catalog, statistical information about all possible subpaths up to a fixed length k are collected as well.

Even though the Lore query processor was only capable of a navigational evaluation of path expressions, it already supported a rich set of index structures (McHugh and Widom, 1999). The *value index* (Vindex) allows to retrieve content values. Using the *label index* (Lindex), all parents of a node having a specified label can be found. The *edge index* (Bindex) helps to query all parent-child pairs with a given label. Finally, a *path index* (Pindex) allows to select all nodes on a specific path in the document. For this task, it employs a *DataGuide*, a structural summary of an OEM document, providing dynamic schema information (Goldman and Widom, 1997).

Natix

Natix (Fiebig et al., 2002) is one of the first XML database systems that allowed to store XML documents natively. This means that the hierarchical structure of XML documents as well as the order among nodes is preserved. In contrast to relational storage, where XML documents must be shredded into several tables, native XML query processing allows for a more efficient and convenient way for querying XML documents[16]. Natix provides support for the evaluation of XQuery expressions on XML documents. As an internal query representation, Natix uses the tuple-based algebra *NAL* (Natix Algebra) (Brantner et al., 2005)—which also supports nested tuples—that strongly influenced the data model of XTC's logical algebra called the *XML Query Graph Model* (XQGM) (Mathis, 2009).

[16] This approach has been strongly adopted by many XML database research projects and is the method of choice in today's commercial database systems, e. g., IBM DB2 or Oracle.

Query optimization in Natix primarily means algebraic optimization. According to May (2007), Natix incorporates a simple cost-based query optimizer that performs (structural) join reordering and access path selection, but does not take advanced indexes (e. g., CAS indexes) or *n*-way join operators (e. g., twig joins) into consideration. Furthermore, the details about cardinality estimation as well as statistics collection are unclear.

TIMBER

Jagadish et al. (2002) sketch the architecture of *TIMBER*, a native XML database system whose query processor relies on a tree-based algebra called *TAX* (tree algebra for XML). In contrast to the Natix approach, TIMBER strongly focuses on set-at-a-time processing of XPath/XQuery expressions.

In the context of TIMBER, one of the most important operators for efficient XML query processing was developed—the SJ operator *StackTree* (Al-Khalifa et al., 2002) that evaluates a structural predicate (e. g., *descendant* relationship) on its ordered input sequences[17]. Moreover, Wu et al. (2003) proposed five novel dynamic programming algorithms for SJ reordering. Their approach is orthogonal to our work, that is, it can be employed to choose the best join order in scenarios where the optimizer can only rely on SJs. Compared to our work, they use only a very simple cost model for driving the join-reordering process and do not consider the combination of SJs and HTJs as well as different index-based (and more advanced) access operators.

Galax

Galax is an open-source project that aims to provide a full implementation of the XQuery 1.0 standard (Fernández et al., 2003) and relies on a file-based storage mechanism. Its evaluation strategy for XQuery expressions directly derives from the guidelines provided by the Formal Semantics of XQuery 1.0 (Draper et al., 2007). Hence, they do not support set-at-a-time processing (e. g., by evaluating path expressions using SJs), but, instead, use node-at-a-time processing that is driven by the nested *for*-loops introduced during query normalization[18].

Query optimization in Galax (Ré et al., 2006) is plain algebraic optimization. Hence, it does not use a cost-based query optimizer. In fact, the authors provide rules for mapping XQuery Core Language constructs to their logical

[17] As we will see soon, SJs are used to evaluate XPath path expressions.

[18] Evaluating XQuery expressions using nested *for*-loops is similar to an evaluation of relational queries using nested-loops joins.

algebra and introduce several optimization rules. Most notably, Michiels et al. (2007) develop techniques allowing to detect tree patterns in XQuery expressions—a task often neglected by researchers providing operators for the evaluation of such patterns (e. g., Bruno et al., 2002).

MonetDB/XQuery

In contrast to the native XML database systems, which we discussed before, MonetDB/XQuery (Boncz et al., 2006) takes a completely different road to XML data management by exploiting mature relational data management technologies for efficient XQuery evaluation. MonetDB/XQuery is a hybrid of the Pathfinder XQuery compiler (Boncz et al., 2005) and the relational database system MonetDB (Boncz, 2002). Pathfinder pre-optimizes and compiles XQuery into an extended relational algebra. In MonetDB, this extended relational algebra is called *Monet Interpreter Language* (MIL) that provides the so-called *Staircase Join* operator (Grust et al., 2003). The Staircase Join allows to efficiently evaluate XPath path expressions—in a similar way as SJs in native XDBMSs do—using a relational query processing infrastructure.

Cardinality estimation in the context of XQuery is difficult, as we have pointed out before. Teubner et al. (2008) present an interesting approach for reliable XQuery cardinality estimation. As we will see later, their approach strongly influenced the advanced cardinality estimation capabilities of XTC's cost-based query optimizer.

System RX/DB2 pureXML

IBM started its endeavors in XML database management by developing a new system prototype called *System RX* (Beyer et al., 2005). System RX is a hybrid of a traditional relational database system and a tailor-made native XML storage with novel indexes and an extension to SQL supporting XQuery expressions, too. According to Beyer et al. (2005), they also use an extension of the seminal Query Graph Model allowing to represent SQL and XQuery in a single model. Moreover, System RX's cost-based query optimizer follows the traditional approach of System R.

Nevertheless, the publicly available resources discussing *DB2 pureXML*—the commercial database product that is based on System RX—do not reveal many details (compare Balmin et al., 2006; Beyer et al., 2006). Though, there are several differences between XTC's XML Query Graph Model and the extension by IBM. For example, in System RX, XPath expressions are not normalized. Instead, they are treated as a whole. This probably provides for a

more convenient usage of path indexes. Their physical algebra provides three novel operators: (1) *XML Scan* (XSCAN), which is similar to a traditional table scan in the relational world, (2) *XML Index Scan* (XISCAN), comparable to a relational index scan, and (3) *XANDOR* allowing to perform the set operations `AND` and `OR` on XML index streams. For the evaluation of path expressions, the *XML Navigation* (XNAV) operator is provided.

Other Approaches to XML Query Processing

Tamino (Schöning, 2001) is a commercial native XML database management system that supports XQuery 1.0 and more advanced features like updates and schema-awareness. According to private communication with two of Tamino's architects during a visit in our research group, their system follows a heuristics-based query optimization approach and does not provide a cost model. The internal query representation as well as query rewriting (especially query unnesting) are strongly influenced by Natix.

Saxon (Kay, 2008) is a stand-alone XQuery/XSLT processor that is available in two different versions. Saxon-B is a complete open-source implementation of XQuery 1.0 (Boag et al., 2007). In addition to that, the commercial product Saxon-SA adds, amongst others, schema-awareness and the support of the XQuery Update Facility (Chamberlin et al., 2009). In Saxon, path expressions are not unnested. Hence, they must be evaluated using navigational primitives. Saxon does not provide a tailored storage engine that would allow to store documents or support to materialize indexes[19].

Even though we aim at providing a complete picture of different approaches to XML query processing, the relatively short history of XML data management has revealed an overwhelmingly large number of systems (e. g., Naughton et al., 2001; Bohannon et al., 2002; Meier, 2002; Meng et al., 2003; Fomichev et al., 2006) that cannot be discussed in detail here, either because they are outdated or do not substantially overlap with the approach followed in this work.

2.3.3 Comparison

So far, we only had a look at the history of query processing in relational and XML database systems. In this section, we compared the query processing capabilities of the *XML Transaction Coordinator* (XTC), which is our prototype of a native XML database system, with the approaches taken by its competitors.

[19] Actually, if Saxon wants to exploit an index for query evaluation, it must be created at runtime, resulting in severe performance drawbacks in many situations.

System / *Property*	Lore	Natix	TIMBER	Galax	MonetDB/ XQuery	System RX/ DB2 pureXML	Tamino	Saxon	XTC[a]
Storage mechanism	native (DBMS)	native (DBMS)	native (DBMS)	native (file system)	relational (DBMS)	native (DBMS)	native (DBMS)	native (file system)	native (DBMS)
Query language	Lorel	XQuery	XQuery	XQuery	XQuery	XQuery	XQuery	XQuery	XQuery
Logical algebra	OQL-based (graph-based)	NAL (tuple-based)	TAX (tree-based)	XQuery Algebra (seq.-based)	Rel. algebra[b] (seq.-based)	QGM (tuple-based)	NAL (tuple-based)	XQuery Core (seq.-based)	XQGM (tuple-based)
Cost-based optimizer	✓	✓	✓		✓	✓			✓
Cardinality estimation	✓ (Lorel)	✓ (XPath)	✓ (XPath)		✓ (XQuery)	✓ (XPath)			✓ (XQuery)
Flexible search strategies									✓
Rule-based plan generation						✓			✓
Seamless query visualization					(✓) (partially)	(✓) (only QEP)			✓
Path processing operators	Navigation	Navigation, Struct. Join	Navigation, Struct. Join	Navigation, Struct. Join, Twig Join	Navigation, Staircase Join	Navigation, Twig Join	Navigation	Navigation	Navigation, Struct. Join, Twig Join
Supports indexing of[c]	Content, elements, paths	Full-text, elements	Content, elements		[d]	Content, elements, paths, CAS[e], Full-text	Content, elements, Full-text		Content, elements, paths, CAS
Parallel query execution		✓ (pipelining)	✓ (pipelining)	✓ (pipelining)	✓ (pipelining)	✓ (pipelining)	(✓)[f]	✓ (pipelining)	✓ (pipelining)

Table 2.1: Comparison of XML query processing infrastructures

[a] The features of XTC reported here, already include the contributions of this thesis, for example, a cost-based query optimizer.
[b] To be precise, MonetDB/XQuery relies on an extended relational algebra.
[c] Here, we only consider materialized indexes and omit main-memory indexes.
[d] We are not aware of any XML-specific indexing capabilities.
[e] Hybrid indexing of content-and-structure (CAS)
[f] Even though the literature does not provide any information on the parallelization opportunities, we assume that Tamino, being a commercial database system, supports it.

Table 2.1 shows the results of the comparison. At first sight, XTC shares many features with its competitors. Nevertheless, XTC's query optimization framework has several features that makes it second to none: (1) XTC provides an extensible framework for the integration of arbitrary search strategies. At the moment, XTC provides six search strategies out-of-the box (e. g., bottom-up optimization and Simulated Annealing). (2) A substantial repertoire of PPOs and index operators provide plenty of opportunities for plan enumeration. (3) XTC provides a visualization component (Weiner et al., 2010) that allows to track the complete query evaluation process from the beginning to the end. Using this tool, every modification of the query graph (during query rewrite) and every possible QEP derived during plan enumeration can be visualized. (4) XTC is the only native XDBMS that supports XQuery cardinality estimation. (5) Fine-granular transaction isolation is key for efficient query processing in transaction-oriented application scenarios.

2.4 Summary

In this chapter, we gave a short introduction to query processing in database systems. In Section 2.1, we had a look at the principles of query optimization. Even though these concepts have been developed in the context of relational database systems, we will see later that they remain valid for native XDBMSs. Section 2.2 discussed the novel challenges of XML query optimization. Moreover, we introduced several types of physical join operators (e. g., structural joins and holistic twig joins) and index structures (e. g., element index) that are necessary for efficient XML query processing. Finally, Section 2.3 first discussed the intersections with related relational database systems as well as with XML database systems. Second, we compared XTC, our prototype of a native XDBMS that serves as testbed for our query optimization approach, with its competitors and revealed the novel features developed in this work.

Consecutively, Chapter 3 discusses the architecture of XTC in general and its index structures in particular. Beyond that, we will have a look at XTC's logical algebra—the XML Query Graph Model (XQGM).

3 Towards Cost-Based XQuery Optimization

"To get through the hardest journey,
we need take only one step at a time,
but we must keep on stepping"

(Chinese proverb)

In this chapter, we introduce the internals of the *XML Transaction Coordinator* (XTC) that are relevant for understanding our query optimization approach.

In Section 3.1, we give a brief overview of XTC's system architecture. We discuss why the DeweyID concept—as a node labeling scheme—is absolutely necessary for efficient query processing. Moreover, we introduce the access paths that the query optimizer can use for query evaluation. Next, we will have a look at the *Path Processing Operators* that are capable of efficiently evaluating (cascades of) structural predicates. Consecutively, we sketch the major components of the *XML Query Graph Model* (XQGM) that serves as our logical XQuery algebra. Thereafter, we deal with XTC's effective cardinality estimation framework EXsum. Afterwards, we give an overview of the query processing pipeline in XTC. In Section 3.2, we discuss the strengths and weaknesses of XTC's current query processor and reveal the roadmap to the cost-based XQuery optimizer being developed in this work. Subsequently, a review of related work is given in Section 3.3. In the end, Section 3.4 concludes this chapter with a short summary and an outlook on Part II.

3.1 System Architecture

Originally, the *XML Transaction Coordinator* (XTC)[1] was developed as a testbed for comparing different lock protocols for XML transaction processing (Haustein, 2006). Soon, it matured to a full-fledged native XML database system supporting, amongst others, ACID transactions and a large fragment of the XQuery language (Härder et al., 2010).

Figure 3.1 on page 40 shows the architecture of XTC (Haustein and Härder, 2007), which follows the classical five-layered architecture introduced by

[1] More information about the project can be found online at: `http://www.project-xtc.de`.

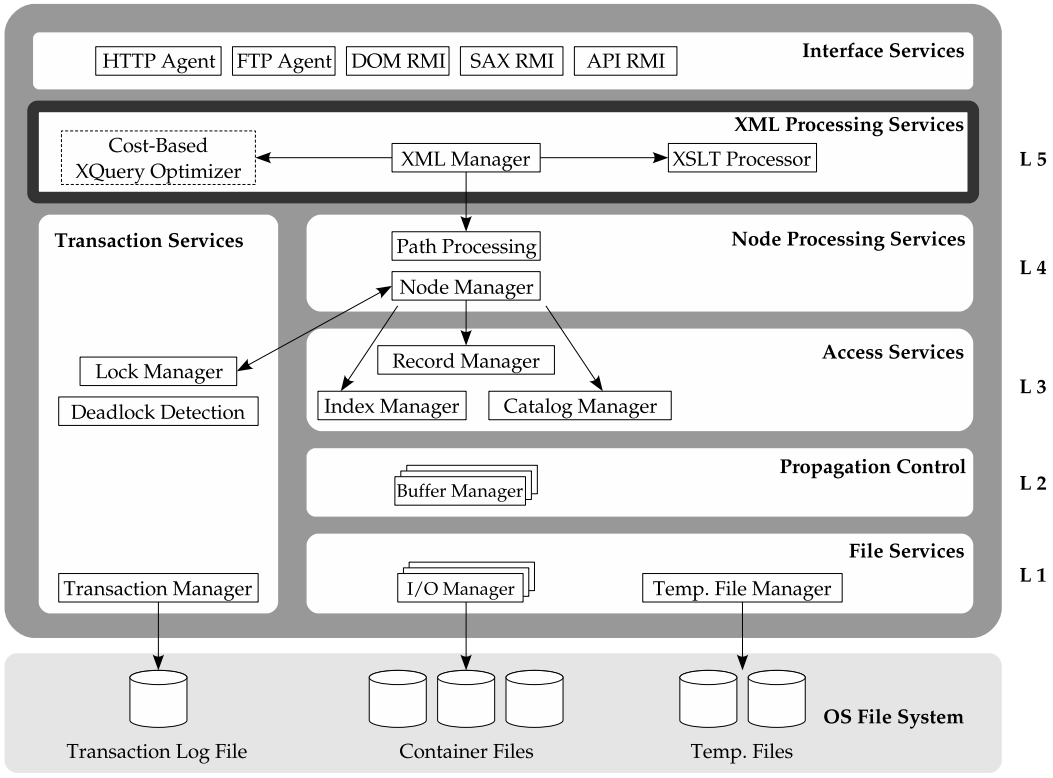


Figure 3.1: Architecture of XTC (after Härder et al., 2010)

Härder and Reuter (1983). The lowest levels are formed by the *File Services* and the *Propagation Control*. The File Services allow for block-oriented access to the external memory. Atop of it, the Propagation Control provides a *Buffer Manager*, which supports various replacement strategies, that offers a page-oriented interface to upper layers. The *Transaction Services* are mandatory for ACID transactions in XTC. Using the *Transaction Manager*, which is responsible, amongst others, for the physiological logging of database operations, permits to reconstruct a consistent system state after a system crash. The ACID paradigm properties *atomicity* and *durability* can be guaranteed by levels 1 and 2, whereas *consistency* and *isolation* can only be realized in higher levels. In level 3 (*Access Services*), where record-oriented operations are supported, for example, loading of records or reconstruction of subtrees, various index structures are situated (e. g., the document index serves as default access path). Moreover, level 3 is responsible for the management of the metadata (*Catalog Manager*). For node-oriented operations, XTC employs the *Node Processing Services* as a means for realizing the DOM axes operations

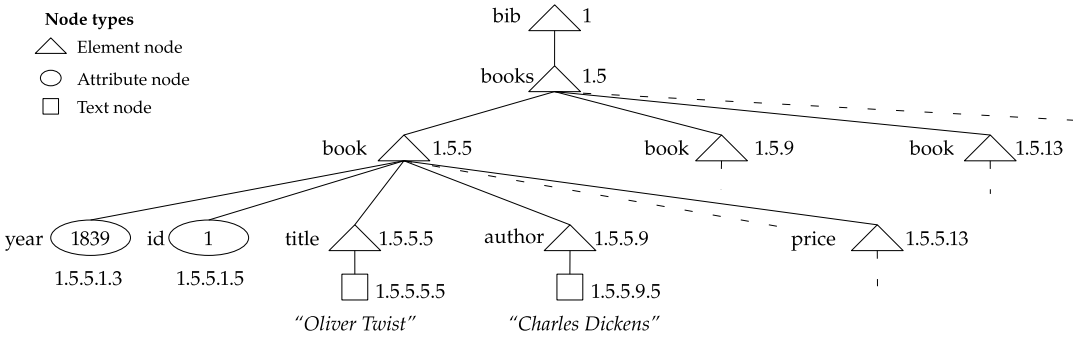


Figure 3.2: XML document labeled with DeweyIDs

(e. g., fetching the first child of a given context node). Here, the *Node Manager* interacts with the *Lock Manager*, which is responsible for transaction isolation, to guarantee a logical single-user mode while concurrently processing several transactions. In the context of this work, layer 5 (*XML Processing Services*) is of utmost importance. Here, the *XML Manager* furnishes different interfaces for accessing and manipulating XML data, for example, Sax, DOM, and XQuery. Most importantly, the XML Manager interacts with the cost-based XQuery optimizer to derive an efficient QEP for an XQuery expression. Finally, the *Interface Services* provide a multi-lingual API for interacting with documents stored in the XTC server.

3.1.1 Node Labeling

In XML documents, we can identify two types of relationships between language constructs: *value-based relationships* and *structural relationships*. Value-based relationships are well-known from the relational world, for example, relationships between foreign and primary keys[2]. In the context of XML, we are additionally confronted with structural relationships. Let us consider the XML document shown in Figure 3.2, which is illustrated as a tree of element, attribute, and text nodes. Here, the most obvious structural relationships are *parent* and *child*. More precisely—the XPath query language (Boag et al., 2007), which is a sublanguage of XQuery and the predominant language for qualifying specific nodes in an XML document—distinguishes between 13 different structural relationships (so-called *axes*).

The naïve approach for evaluating a structural relationship, for example, the descendant axis, would require a traversal of the document. As we will

[2] In XML documents, value-based relationships may be expressed using the ID/IDREF construct.

see soon, real-world XML queries make excessive use of axis relationships to identify nodes related to a specific context node in the document. Hence, the drawback of the naïve approach is obvious: it requires numerous and expensive accesses to the document. If we could decide this problem based on a static property and without any document accesses, we could immensely speed-up query processing. The precondition for achieving this goal is a *node labeling scheme*. Such a scheme assigns to each node of an XML document a unique identifier in left-most depth-first traversal order. Prominent representatives of node labeling schemes are *prefix-based schemes* (Härder et al., 2007).

Intrinsically, prefix-based node labels encode the complete path from the document root to the associated node. Hence, by simply comparing the labels of two XML nodes, we can decide whether they are structurally related to each other with respect to a specific axis. In XTC, we use the so-called *DeweyID* concept that relies on the famous prefix-based and hierarchical node labeling scheme *ORDPATH* (O'Neil et al., 2004). In contrast to the ORDPATH approach, XTC's DeweyID concept has a more refined DeweyID order mapping and supports a mechanism allowing to reserve some gaps for overflow handling (Härder et al., 2007).

For example, let us reconsider the XML document shown in Figure 3.2. Each DeweyID consists of several divisions that are separated by the character ".". The root of the document is always labeled with DeweyID 1. Using a left-most depth-first traversal of the document, each child node extends its parent node's DeweyID by a new division, where each division value increases from left to right. Initially, only odd division values are assigned. Even division values might be inserted later on, if an overflow occurs. For determining the tree level of a given node, we just count the total number of odd divisions of its DeweyID. For example, the DeweyID 1.5.5 indicates that the associated book node is on level 3. Moreover, we can infer that 1.5.5 is an ancestor of 1.5.5.5.5, because both labels share the same prefix. But, in turn, 1.5.5.5.5 is not a child of 1.5.5, because their levels differ by more than one, though they share a common prefix. Actually, DeweyIDs allow to decide all relevant axis relationships used in XQuery[3]. This property makes them essential for efficient query processing in XTC[4].

[3] A detailed discussion of the benefits and drawbacks of different node labeling schemes, in general, and the DeweyID concept, in particular, is out of scope of this work. You will find more information about the efficient implementation of the concept and its benefits for query processing as well as fine-granular transaction isolation in the comprehensive article by Härder et al. (2007).

[4] It is not an exaggeration to claim that without such a labeling scheme, no efficient native XDBMS can be built at all.

3.1.2 Path Processing Operators

In Section 2.2.1 on page 24, we discussed the challenges for XML query processing. Probably the most important challenge for query processing is efficient evaluation of structural relationships. XTC provides three types of *Path Processing Operators* (PPOs) for the evaluation of structural predicates: the navigational operator *NavTree*, the SJ operator *Extended StackTree*, and the HTJ operator *Extended TwigOpt* (Mathis, 2009)[5].

The NavTree operator returns for a single node (context node) or for a sequence of nodes, all nodes that satisfy an XPath axis step—which may be filtered by an additional name test. Rather than evaluating structural predicates in a nested-loops style (NavTree), StackTree evaluates a structural predicate similar to a relational sort-merge join. In contrast to the classical StackTree operator, Extended StackTree allows to evaluate semi joins and outer joins. As we will see soon, cascades of Extended StackTree operators can be optimized using classical relational optimization techniques.

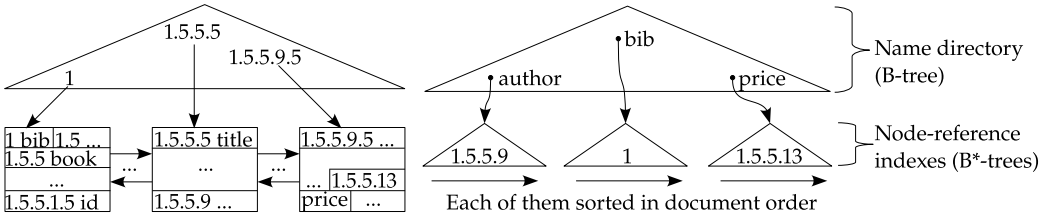
Finally, the Extended TwigOpt operator is an n-way join operator that evaluates so-called *twig query patterns* in a holistic manner. Hence, there is no need to evaluate each binary relationship separately; as it would be the case for a cascade of SJs. Even though Bruno et al. (2002) once promoted HTJs as superior over SJs, we will see later that in realistic XQuery evaluation scenarios, the opposite is true[6]. In fact, HTJs do not provide much opportunities for query optimization, because they are more or less a "black box" for the query optimizer and solely allow for cost-based access path selection. In contrast, SJ operators can be optimized using classical join reordering techniques and support query evaluation using pipeline parallelism.

3.1.3 Access Paths

As we have already mentioned in Section 2.2.1 on page 24, there exists a plethora of different XML indexes for accelerating query evaluation. In the context of XTC, we distinguish between three classes of access paths: primary, secondary, and tertiary access paths.

[5] As their names indicate, both operators are extensions of previously introduced operators. The classical StackTree operator was introduced by Al-Khalifa et al. (2002), whereas Fontoura et al. (2005) described the original TwigOpt operator.

[6] In our opinion, HTJs are only superior to cascades of SJs, if obscure patterns shall be evaluated, which are making almost exclusive use of the "//" axis and whose semantics is therefore becoming fuzzy. Such patterns may occur in information retrieval scenarios, but are rarely seen in classical database queries.



(a) Document index (b) Element index

Figure 3.3: Primary and secondary access path

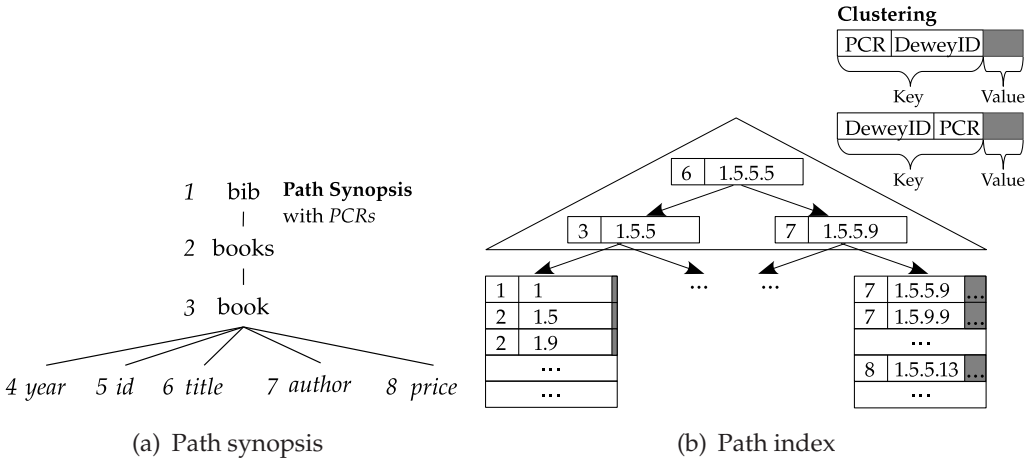
Primary Access Paths

The *primary access path* (PAP) of XTC is called *document index*. This data structure is available per default and indexes an XML document using its unique DeweyIDs as keys. For example, Figure 3.3(a) depicts the document index for the XML document illustrated in Figure 3.2 on page 41. The document index is implemented as a classical B*-tree. For navigational query evaluation (with respect to a specific context node), this data structure helps to identify all relevant nodes by their DeweyIDs, for example, finding all descendant nodes of a given context node can be easily solved by a range scan over the leaf nodes. Nevertheless, it is quite insufficient for finding all nodes having the same name. For example, finding all *book* element nodes would require to scan all leave pages of the index, even though only a small fragment of them actually contains *book* records.

Secondary Access Paths

To overcome the deficits of the document index, XTC provides an *element index* as *secondary access path* (SAP). This index is implemented as a two-stage index consisting of a *name directory* (B-tree) and a set of *node-reference indexes* (B*-tree), as illustrated in Figure 3.3(b). The name directory uses all unique element names of the document as keys. Each key points to a node-reference index that contains all DeweyIDs in document order, which share the same element name.

As we will see soon, the element index is crucial for efficient query evaluation. Structural joins and holistic twig joins rely on it as an efficient means for providing streams of DeweyIDs having the same element name. This operation is particularly important for evaluating the omnipresent path expressions.



(a) Path synopsis (b) Path index

Figure 3.4: Path synopsis and path index

Tertiary Access Paths

In XTC, we distinguish between path indexes and content-and-structure indexes. Advanced indexing in XTC is supported by *tertiary access paths* (TAPs), which rely on the *path synopsis* (PS). The PS is a small, memory-residing data structure, which is an extension of the seminal DataGuide (Goldman and Widom, 1997), serving as a structural summary or a dynamic document schema. Moreover, the PS provides valuable statistical information for cardinality estimation. Figure 3.4(a) shows the PS for the XML document depicted in Figure 3.2. Each unique path in the PS is annotated with a so-called *path-class reference* (PCR) number (Mathis et al., 2009).

Path Index Figure 3.4(b) shows an instance of XTC's *path index*. The path index allows to index complete paths (including potential leaf node values). It uses a combination of PCR and DeweyID as key and allows to cluster its records by PCRs or by DeweyIDs, respectively. If a path ends on a content node, for example, for PCR 7, its content value is also stored in the corresponding record. For example, if we would like to find all nodes that satisfy the path expression *//bib//book* we just need to scan the path index for records having PCR 3.

Content-and-Structure Index XTC's *content-and-structure index* (CAS index) allows to index content and structural information at once. Figure 3.5 shows a CAS index. Here, the content value acts as key in the B*-tree and

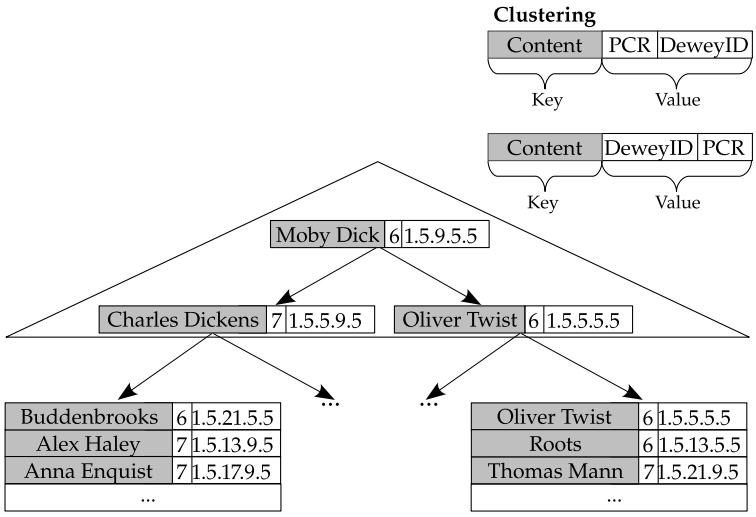


Figure 3.5: Content-and-structure index

a combination of PCR and DeweyID, or vice versa, serves as index value, depending on the chosen clustering.

For example, using the CAS index depicted in Figure 3.5, we would efficiently find an answer to the question: *Whose author name is in lexicographical order lower than "C"*. By identifying the qualifying nodes, we get the corresponding DeweyIDs and the PCR numbers allowing to reconstruct the complete path from the node to the document root. Otherwise, answering this question without an CAS index would force us to first retrieve all nodes satisfying *bib/books/book/author* using cascades of SJs or an HTJ and then filter out all *author* nodes whose content is larger or equal than "C"[7].

3.1.4 The XML Query Graph Model

The basis for every query optimization is a suitable logical algebra serving as internal query representation. Mathis (2009) introduced the *XML Query Graph Model* (XQGM) as XTC's logical algebra. In this section, we give a brief introduction to the basic concepts of XQGM that are necessary to understand our query optimization approach.

[7] You can imagine that this approach raises much more IO and CPU cost compared to an access to the CAS index, where, in contrast, access and processing costs are reduced to a minimum.

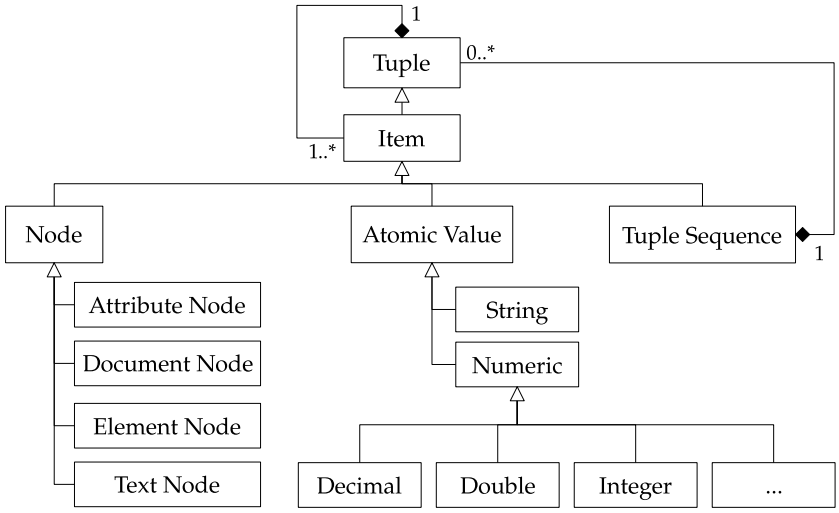


Figure 3.6: The XQGM data model (after Mathis, 2009)

The XQGM Data Model

The XQGM data model is based on the *XQuery Data Model* (XDM) (Fernández et al., 2007), which provides *atomic values*, *items*, and *sequences* as model primitives. The XQGM Data Model complements these primitives by *tuples*. More precisely, every XQGM Data Model object is a tuple (Mathis, 2009). Figure 3.6 shows a UML diagram of the different data model primitives. An item is either (1) a node, (2) an atomic value, or (3) an ordered list of tuples which we denote as *tuple sequence*. Consequently, tuple sequences allow for expressing nested tuples. We call a tuple sequence that contains exactly one tuple a *singleton tuple sequence* and a tuple that is formed by a single item is referred to as *singleton tuple*[8]. We say that a singleton is always equal to its unique element. Moreover, a tuple sequence is an XQuery sequence, if and only if it contains only singleton tuples or is equal to an empty sequence (Mathis, 2009).

XQGM Example

Figure 3.7 on page 48 shows an XQGM instance for Query 3.1, which returns the author names of books whose price is larger than 1.99.

In XQGM, operators are not connected with each other directly. Instead, an operator contains so-called *tuple variables* that handle the tuple sequences emitted by its input operators. Tuple variables can have three different types

[8]For the rest of this work, whenever the context is unambiguous, we just refer to them as singleton.

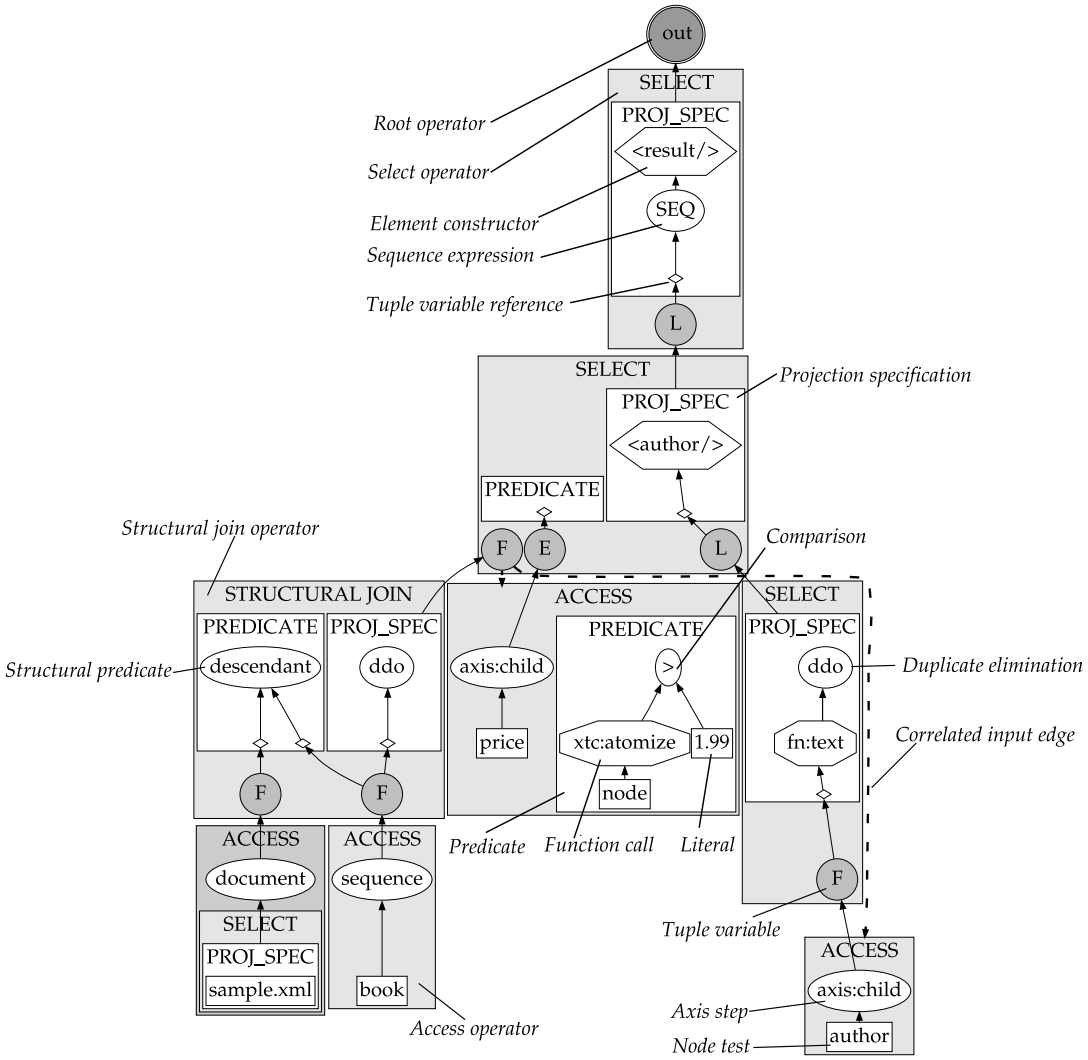


Figure 3.7: Sample XQGM instance

Query 3.1 Simple XQuery expression with value-based and structural predicates

```
<result> {
 for $b in doc("sample.xml")/descendant::book
 where $b/price > "1.99"
 return
  <author>{ $b/author/text() }</author>
} </result>
```

of quantifiers: *for* (F), *let* (L), and *exists* (E), whereupon the first and the second quantifier have a similar semantics as the correspondent XQuery constructs: A *for*-quantified tuple variable iterates tuple-wise over the tuple sequence received from its connected child operator. In contrast, *let*-quantified tuple variables expose their input only once as a complete tuple sequence with respect to a specific context node. Finally, *exists*-quantified tuple variables are only used in combination with predicates to check for the existence of a non-empty tuple sequence.

Now let us dive deeper into the XQGM semantics and discuss the evaluation of our sample query: Query evaluation starts at the left-most subtree, whose root is the SJ operator. The SJ operator evaluates the structural predicate (descendant axis) between the virtual document root and all *book* elements. Actually, SJ is a semi join operator, because only the tuple variable that is connected to the Access operator providing *book* elements is, in turn, connected to the projection specification. The parent Select operator receives a tuple sequence of all *book* elements and iterates over it: First, it passes the current evaluation context to the Access operator connected to it via the *exists*-quantified tuple variable[9]. The Access operator selects a sequence of all *price* element nodes that are children of the current (context) *book* element and returns only those items that satisfy the predicate. Next, the Select operator passes the current context to the right-most Access operator, which returns all *author* nodes that are connected to the current context node via the child axis. For each match, the parent Select operator evaluates the *fn:text()* function. Now, the Select operator, which is at the heart of the query graph, can produce the output: It creates a new XML element *<authors>seq</authors>*, if the sequence bound to the *exists*-quantified tuple variable is not empty, where *seq* is the tuple sequence received by the *let*-quantified tuple variable. Finally, the

[9] In the graphical XQGM representation, the context passing is illustrated as a dashed line from the "sending" tuple variable to the receiving operator.

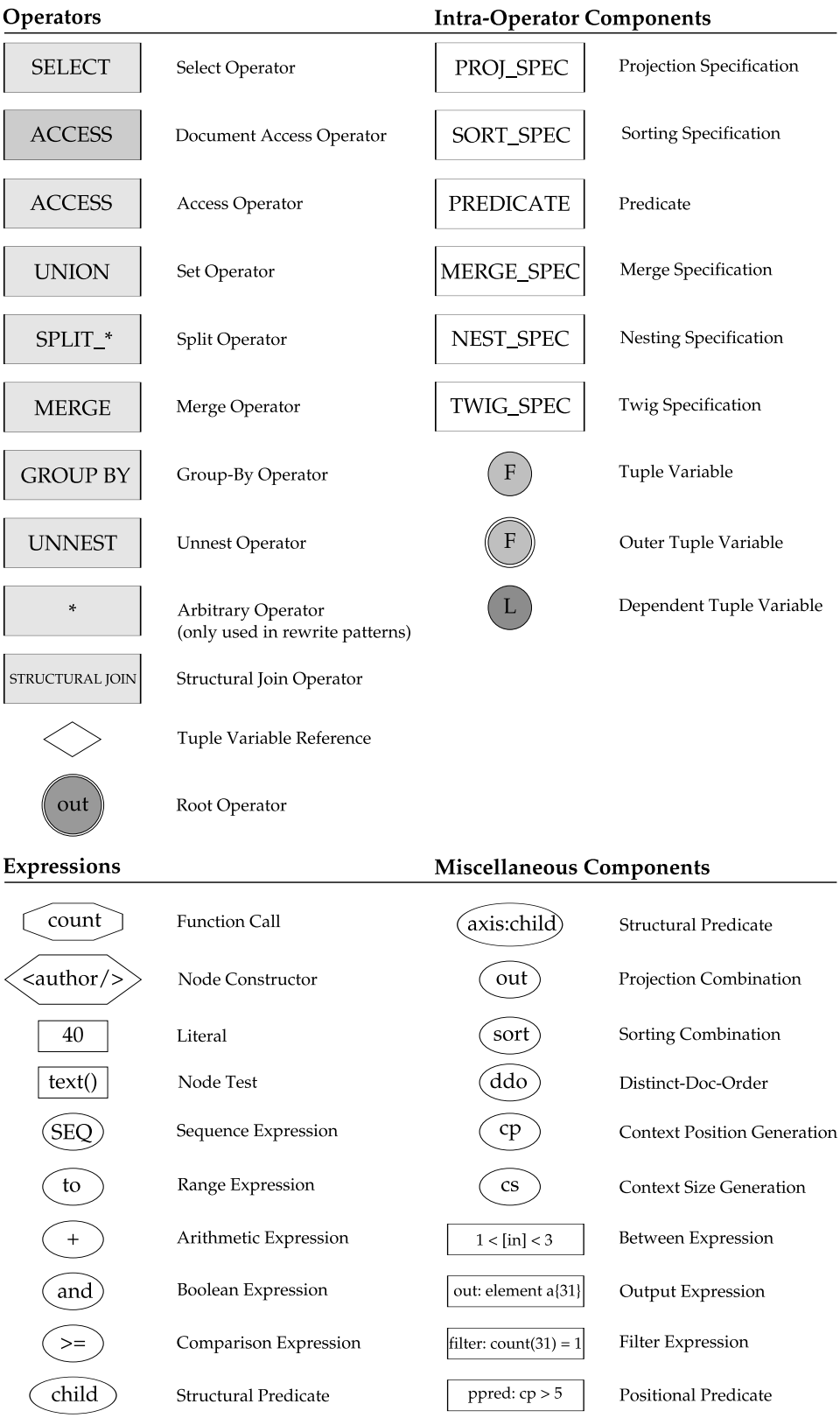


Figure 3.8: Overview of the XQGM components (adapted from Mathis, 2009)

top-most Select operator wraps its input in an opening <*result*> and a closing </*result*> tag.

XQGM Overview

For the rest of this work, we will need some basic knowledge on the different logical operators provided by XQGM. Hence, this section gives a brief overview over all operators. Figure 3.8 illustrates the integral parts of XQGM:

- *Access* operators are leaf nodes in an XQGM instance and provide access to XML nodes.
- The *Select* operator either simply evaluates predicates on a tuple sequence or mimics the *for* and *let* XQuery language constructs.
- The *Set* operator performs union, intersection, and difference on multiple (union and intersection) and two (difference) input tuple sequences.
- The *Split* operator allows to send its output to multiple receiving operators.
- Whenever there is a Split operator in an XQGM instance, the *Merge* operator serves as its counterpart for reuniting the data flow to a single tuple sequence and has the same semantics as an *n*-way outer join operator.
- As discussed earlier, the XQGM data model supports nested tuples. Grouping an input tuple sequence according to a specific item position can be performed by the *GroupBy* operator.
- Using the *Unnest* operator, which is the reverse operator of GroupBy, we can expand a nested tuple sequence with respect to a given item position.
- Query rewrite using *query unnesting* (Mathis, 2009) iteratively replaces Select operators, which are evaluating nested subexpressions and which are only evaluating structural predicates, by a *Structural Join* operator. Consequently, the node-at-a-time evaluation paradigm dictated by XQuery is replaced by a set-at-a-time approach being inherent to structural joins[10].

[10] Amongst others, query unnesting is mandatory for actually using structural joins for the evaluation of arbitrary XQuery expressions. You will find more information about this complex query rewrite in Mathis (2009).

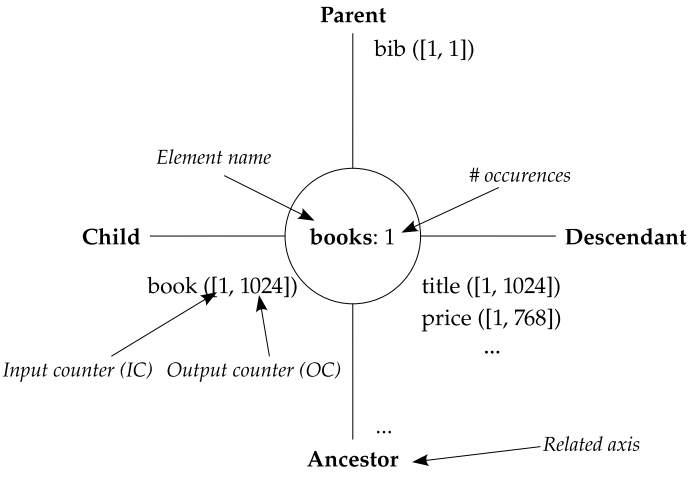


Figure 3.9: Sample EXsum summary node

3.1.5 Rudimentary Cardinality Estimation

In Section 2.1.3 on page 17, we argued that two components are absolutely necessary for cost-based query optimization: (1) effective cardinality estimation and (2) cost estimation. Aguiar Moraes Filho (2010) focused on the first problem and brought rudimentary support for XPath cardinality estimation to XTC.

Amongst others, Aguiar Moraes Filho (2010) provided a cardinality estimation approach called *EXsum* (Element-centered XML Summarization). EXsum captures all structural relationships between XML element nodes. This approach can be used to estimate the cardinality of simple XPath expressions. In contrast to previous techniques (e. g., Aboulnaga et al., 2001; Wang et al., 2004; Zhang et al., 2006), EXsum goes beyond cardinality estimation for widely-used XPath axes (e. g., child and descendant) and now supports, in principle, *all* XPath axes. Moreover, it allows to estimate the value distribution of content nodes.

EXsum maintains for every element name in an XML document an *ASPE* (Axes Summary Per Element) node. Figure 3.9 shows an ASPE node for the element *books*, which occurs in the XML document depicted in Figure 3.2. Here, the total number of occurrences is recorded (# occurrences) and an *Input Counter* (IC) as well as an *Output Counter* (OC) keep track of the cardinalities of those elements that are structurally related to the context node via a forward axis or a reverse axis, respectively. Let us assume that our XML document encompasses information about 1,024 books. Moreover, let us assume that

every *book* has a *title* child node, but only 75% of all books have a *price* tag. According to the PS depicted in Figure 3.4(a), *books* is the parent of *book* nodes and *title* nodes are descendants of *books* nodes. The illustration of the corresponding ASPE node reflects this information. The IC for *book* elements indicates that *book* nodes have exactly one parent: *books*. Moreover, *books* has 1,024 *book* nodes as children.

To estimate the cardinality of a single XPath axis step, EXsum probes the ASPE node of the context node and returns the exact cardinality of the structurally-related node. For a multi-step path expression, EXsum retrieves the corresponding ASPE nodes and interpolates the intermediate values to get a more accurate estimation[11].

3.2 Discussion and Roadmap

So far, this chapter briefly introduced different aspects of XTC. Now, we will analyze the deficits of the current approach and provide a roadmap for this thesis describing the steps that will be taken to finally establish cost-based query optimization in XTC.

3.2.1 Twig Discovery Considered Harmful

Mathis (2009) provides the groundwork for query optimization in XTC by introducing new storage and index structures, a logical query algebra (XQGM), several query simplification rules, for example, query unnesting, and a set of physical algebra operators. However, this approach only supports heuristics-based query processing, that is, XQGM-to-QEP mappings are hard-wired, but can be changed manually by an administrator. The coupling between XQGM and the corresponding QEP is rather tight, making it difficult to integrate arbitrary bottom-up or top-down plan generators, which are necessary for implementing flexible query optimizers.

Twig discovery is advocated as a technique that already detects opportunities for applying physical HTJs during logical query rewrite (Mathis, 2009). In our opinion, this approach is a perfect match for the (computer) science dictum:

> "*premature optimization is the root of all evil*" (Knuth, 1974, p. 268)

At first glance, this idea seems to be attractive, because it promises to dramatically reduces the number of operators involved in XQGM instances and

[11] There exist several corner cases that must be taken care of during cardinality estimation. Discussing theses situations is out of the scope of this work. More detailed information can be found in Aguiar Moraes Filho (2010).

consequently restricts the number of intermediate results. Unfortunately, this approach takes away almost all room for cost-based optimization, because:

1. it makes decoupling during cost-based optimization expensive, if the plan generator decides that query execution using optimally-reordered SJs is actually faster.

2. now, no join reordering can be performed anymore. Thus, the search space is reduced and might no longer contain the optimal plan. Moreover, the plan generator's job is limited to finding the optimal set of access paths.

3. cascades of SJs can be parallelized more easily using pipelining than HTJs.

The HTJ community (headmost, Bruno et al., 2002) always claimed that HTJs are also superior to SJs in terms of performance. In fact, the empirical results provided by Mathis (2009), which, amongst others, trade HTJs off against SJs in real XQuery processing scenarios, show that this claim is not true. Rather, it seems that HTJs are only superior to SJs, if an XML document is queried for obscure twig patterns, for example, patterns that only involve the descendant axis and whose actual semantics is—at least—questionable (also have a scrutiny at the empirical evaluation in Bruno et al., 2002).

Moreover, we experienced that the decision for or against HTJs is not as performance-critical as expected. For example, if we consider the XMark benchmark queries, which encompass fairly complex XQuery expressions, the performance of SJs and HTJs is mainly thwarted by blocking operators like GroupBy, Unnest, and Merge that actually determine the "heartbeat" of the execution.

Having these facts in mind, our query optimizer still considers HTJs as a possible alternative, but does not overestimate its importance for efficient execution. Thanks to our rule-based XQuery-to-XQGM compiler *XTCcmp* (Mathis et al., 2008), we can easily switch off the twig discovery rule. We will see later that the query optimizer's ability of making the most out of the various SAPs and TAPs has much greater influence on the performance than deciding the cumbersome SJ-versus-HTJ debate.

3.2.2 Cardinality Estimation for Path Expressions

XTC's current cardinality estimation framework EXsum mainly targets at cardinality estimation for XPath expressions. For cost-based XQuery optimization, this is insufficient, because it does not help to estimate the cardinalities

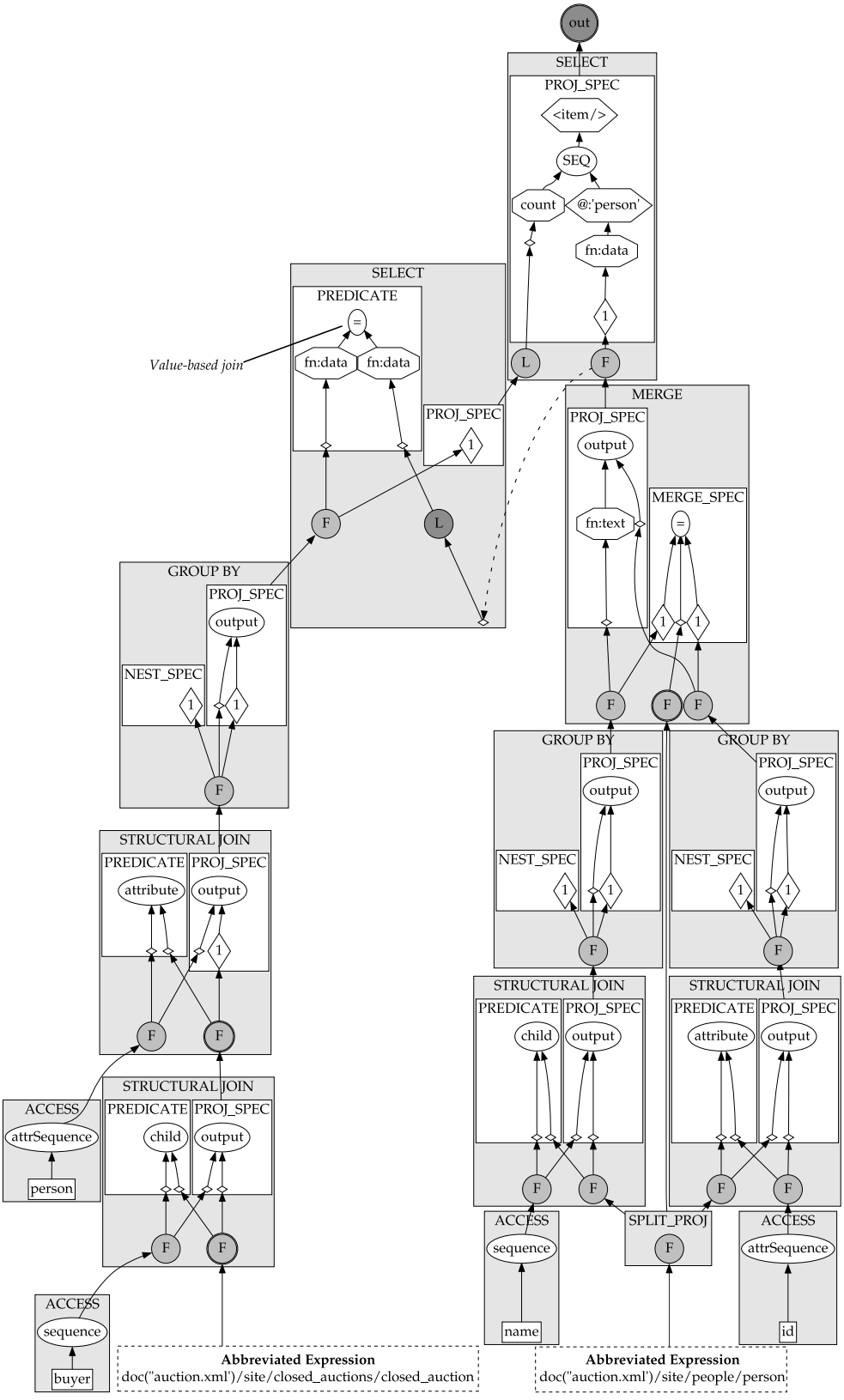


Figure 3.10: XQGM instance for XMark benchmark query Q8

Query 3.2 XMark benchmark query Q 8 (Schmidt et al., 2002)

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

for complex *for* and *let* bindings. To illustrate the fundamental problem, let us have a look at an example.

Figure 3.10 on page 55 shows the corresponding XQGM instance for Query 3.2. This query involves a value-based join (expressed as a predicate in a Select operator). As in the relational case, value-based joins can be evaluated using well-known physical operators: nested-loops join, sort-merge join, and hash join. For equality predicates, the hash join is most often a good choice. As the evaluation of a hash join can only be efficient, if its hash table is built for the smaller input rather than for the larger one, cardinality estimates are absolutely necessary for determining the correct proportions.

Let us test the capability of EXsum to furnish sufficient cardinality estimates: In this example, EXsum can only provide cardinality estimates for the abbreviated path expressions (sketched as boxes with dashed borders) that are normally expressed as cascades of structural semi join operators in XQGM. Such cascades emit only tuple sequences that are singleton tuples. For all other operators, the current approach is not applicable. For example, the left-most SJ operator evaluates an outer join and the SJs on the right side evaluate full joins. Both types of joins emit tuple sequences that contain binary tuples—one item for each join partner.

Moreover, EXsum has no means for estimating the cardinalities of GroupBy operators, which contain nested tuples. Consequently, we cannot provide correct cardinality estimates for the value-based join and probably hash the larger input instead of the smaller one, which may cause a severe performance loss.

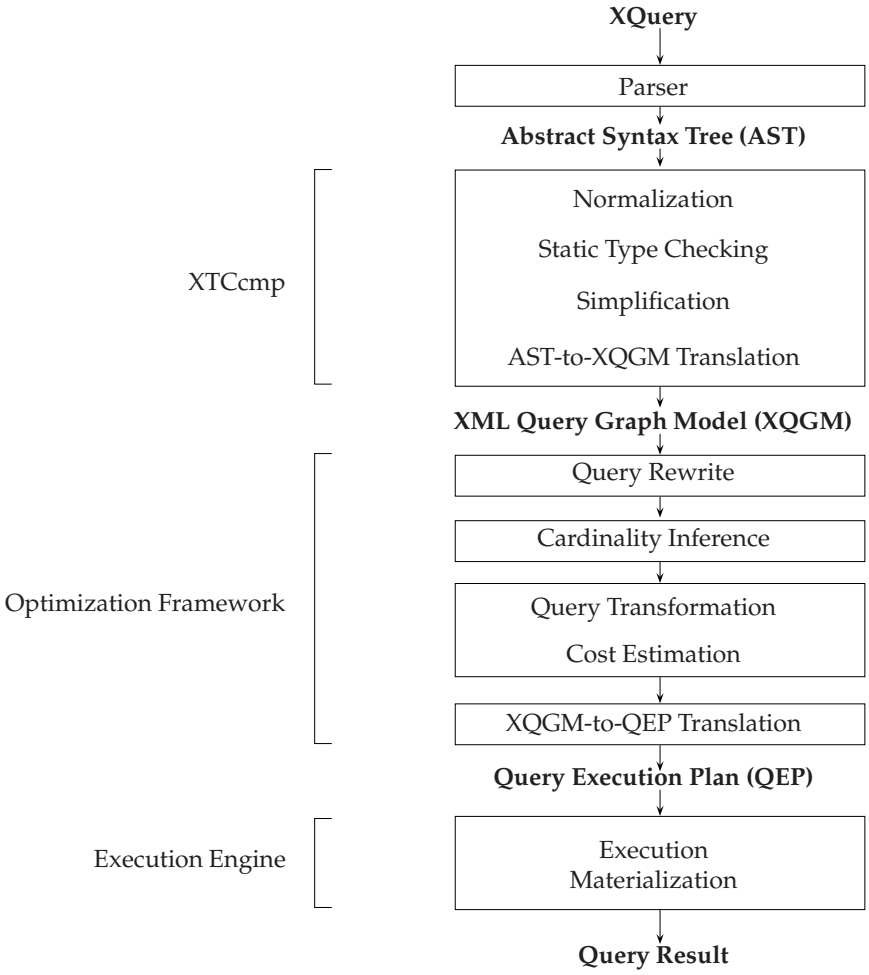


Figure 3.11: The XTC query optimization process

3.2.3 Roadmap

After our discussion of the deficits of the current XTC server with respect to its query processing and query optimization capabilities, we will now outline the roadmap for our query optimizer.

Figure 3.11 shows the complete query evaluation process of XTC's cost-based optimizer. We retain the *XTCcmp component* for compiling XQuery expressions into XQGM and use almost all query rewrite techniques—except twig discovery—proposed by Mathis (2009). The *Optimization Framework* (Part II) is one of the major contributions of this work that performs cost-based query optimization by generating and evaluating semantically-equivalent plans. Moreover, this component uses an advanced cardinality estimation

framework that pays attention to the XQGM data model. Finally, XTC's *Execution Engine* is reused to run the optimal QEP.

In Chapter 4, we will discuss logical query rewrite rules that are applied to an XQGM instance before optimization. Chapter 5 describes the formal representation of plans in our optimization framework. Next, logical query transformations, for example, structural join associativity or join fusion, are introduced in Chapter 6. These transformations allow to derive semantically equivalent plans for a single expression and empower the plan generator to inspect the search space. Chapter 7 deals with the advanced cardinality estimation framework of XTC. Chapter 8 provides the cost formulæ forming the query optimizer's cost model. Finally, Chapter 9 discusses how the concepts developed in Chapters 4–8 are used to perform cost-based query optimization.

After providing the theoretical background for query optimization in Part II, we elaborate on the implementation details and provide an in-detail empirical evaluation of our approach in Part III. In the end, Part IV concludes this work and provide an outlook on potential future work.

3.3 Related Work

As we have already mentioned before, Haustein (2006) provided the initial XTC prototype and introduced important concepts such as the taDOM family of lock protocols that allows for fine-granular transaction isolation based on the Dewey labeling scheme. Additionally, the early XTC prototype provided the document index (PAP) as well as the element index (SAP) (Haustein and Härder, 2007).

The comprehensive work of Mathis (2009) introduces several new aspects to XTC. As we have discussed before, mapping a large fragment of the XQuery language onto XQGM and providing heuristics for query optimization (e. g., query unnesting and twig discovery) is one of the integral parts of this work. Moreover, the thesis specifies two TAPs: path indexes and content-and-structure indexes that allow to speed-up query evaluation significantly. Finally, the work defines a fixed mapping of XQGM instances onto a set of physical algebra operators and provides an empirical evaluation of the effectiveness of the query rewrite rules with respect to query evaluation performance. Unfortunately, this work does not discuss cost-based query optimization.

Aguiar Moraes Filho (2010) contributes techniques for cardinality estimation of simple path expressions that rely on the path synopsis. The most advanced data structure (EXsum) can even handle value distributions. The

experimental evaluation compares the different approaches with each other by taking space consumption, estimation quality, and performance into consideration. This approach is only a small steps towards XQuery-level cardinality estimation that is quintessentially necessary for our cost-based optimizer.

3.4 Summary

In this chapter, we had a look at the most important concepts of XTC. In Section 3.1, we introduced the architecture of XTC and presented some "ingredients" needed for query optimization: Path Processing Operators for efficiently evaluating structural predicates, various access paths, and the logical XQuery algebra XQGM (XML Query Graph Model). Moreover we looked at the rather limited technique for cardinality estimation. Section 3.2 discussed the shortcomings of XTC's current query engine and provided a roadmap for the techniques that will be developed in this thesis to establish a cost-based XQuery optimizer. Finally, a short overview of related works is given in Section 3.3.

This section concludes Part I of this thesis. Consecutively, in Part II, we will introduce the theoretical background for our cost-based XQuery optimizer.

II

Optimization Framework

4 Query Rewrite

"Things alter for the worse spontaneously, if they be not altered for the better designedly."

(Francis Bacon)

In this chapter, we introduce two types of cost-aware query rewrites that can be applied even before the classical query optimization phase starts.

In Section 4.1, we will see how we can push-up expensive *fn:text()* accesses in XQGM instances and, therefore, delay their evaluation as long as possible. Moreover, Section 4.2 provides a termination criteria for query unnesting that helps to avoid the counterproductive application of them.

Finally, Section 4.3 summarizes this chapter and provides an outlook on the next chapter.

4.1 Push-Ups of fn:text() Accesses

In the relational world, selection push-down is a simple—but effective—heuristics to restrict the inputs for join operators by pushing down selection predicates to their access operators.

Query 4.1 Slightly modified XMark benchmark query Q1 (Schmidt et al., 2002)

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person
where $b/@id = "person0"
return $b/name/text()
```

In native XDBMSs, the evaluation of value-based predicates in general and of XQGM *fn:text()* functions in particular is very costly, because they require additional accesses to the document. Therefore, we want to avoid the evaluation of this function as long as we can be sure that it is evaluated only for those tuples that are actually part of the final query result and that are not filtered out

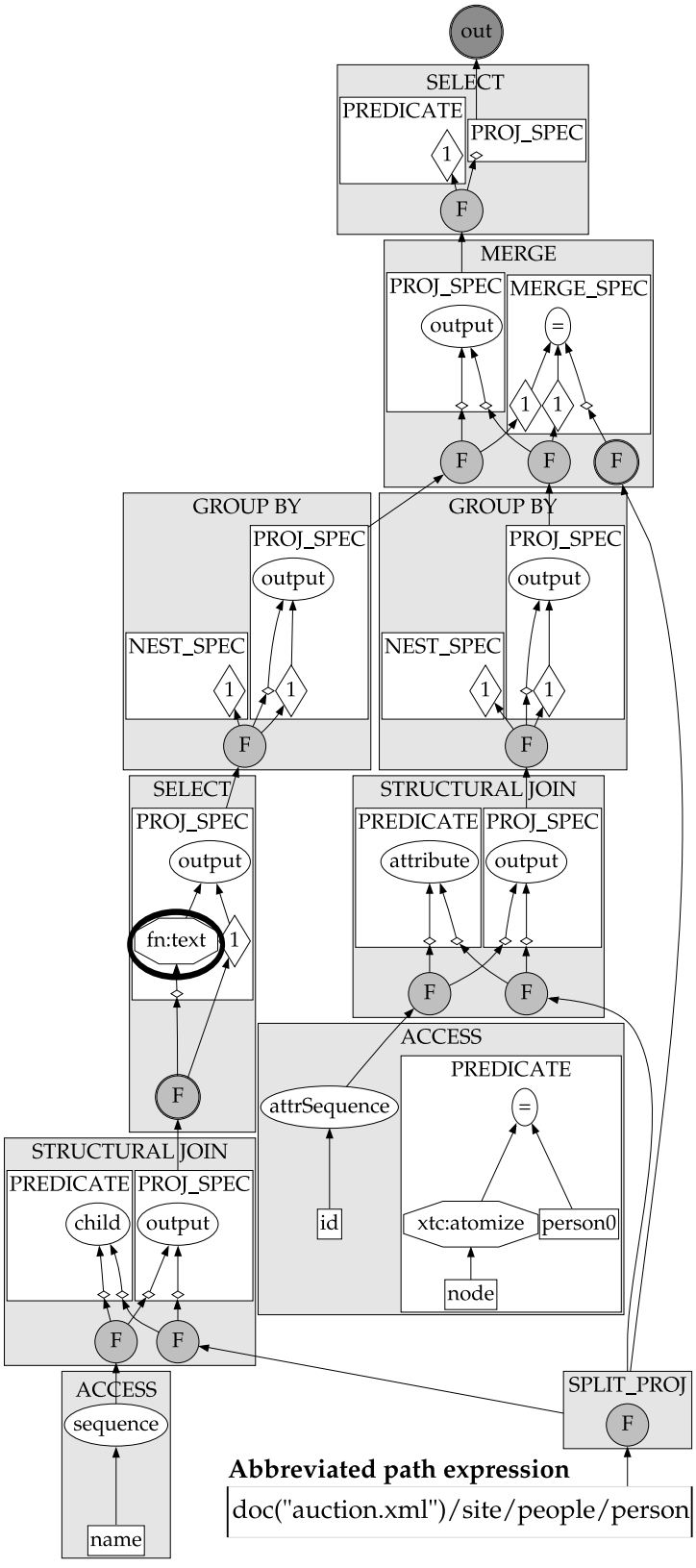


Figure 4.1: Modified XMark benchmark query Q1—without push-up

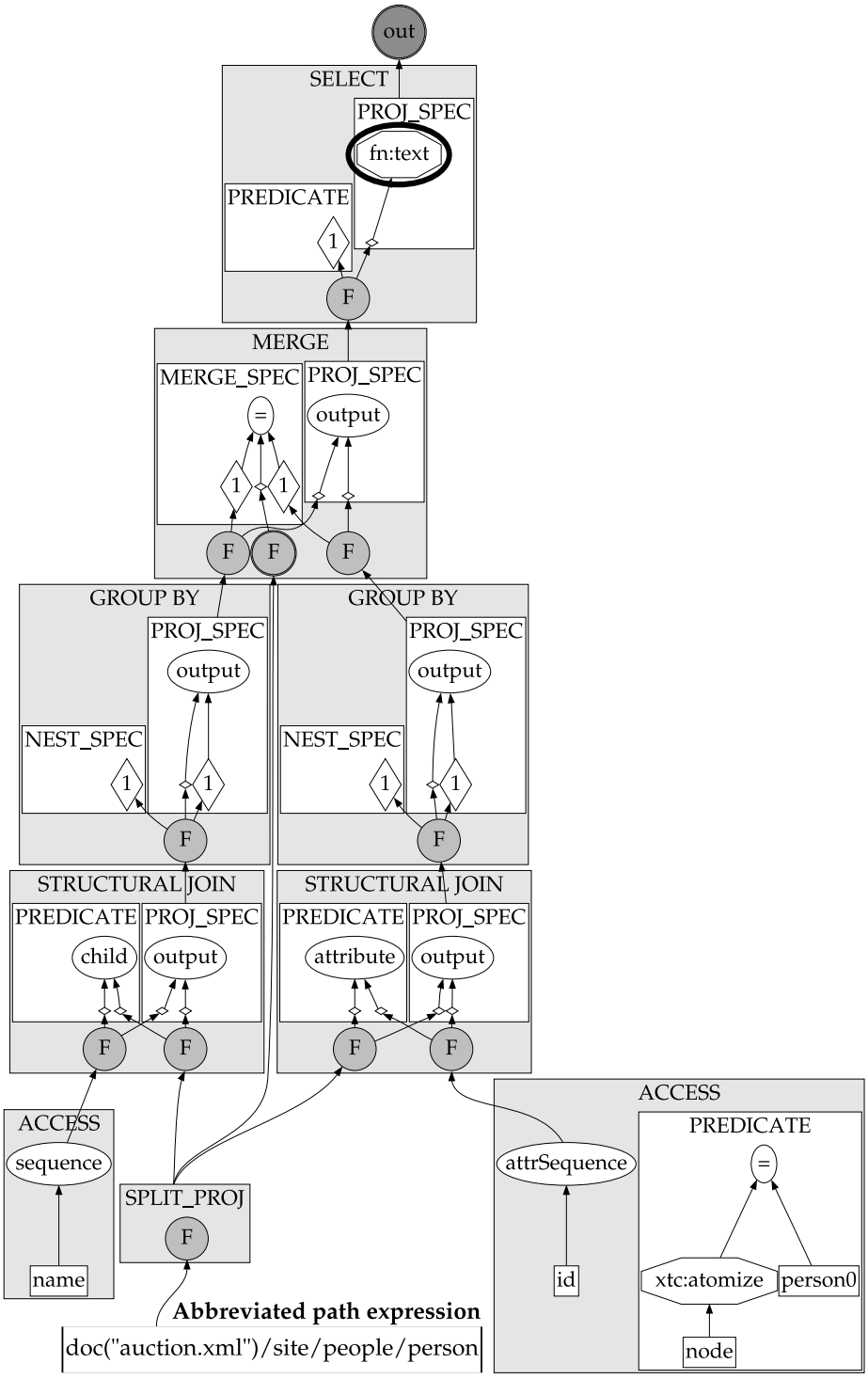
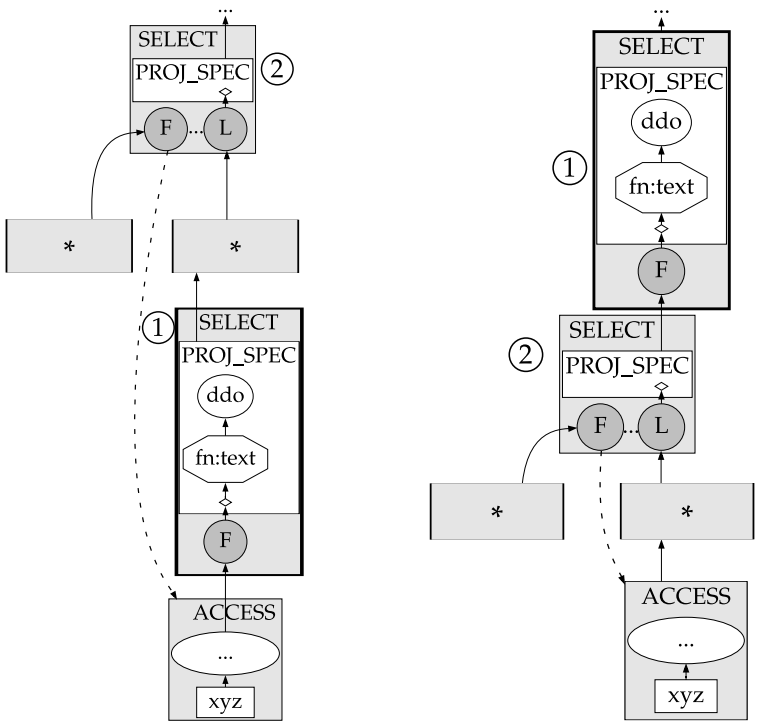


Figure 4.2: Modified XMark benchmark query Q1—after push-up



(a) Left-hand side (b) Right-hand side

Figure 4.3: SelectionPushUp pattern in action

by structural or positional predicates. Let us have a look at Query 4.1 on page 63. Figure 4.1 on page 64 shows the respective XQGM representation. Here, after joining *name* and *person* nodes, all *fn:text()* values for *name* are fetched as well.

The Merge operator returns only those nodes on *doc(...)/site/people/person* that satisfy the *where* condition. Let us assume that the predicate is very selective, hence, we can expect that only few *name/text()* nodes will be part of the final result. Nevertheless, we had to evaluate the *fn:text()* function on all *name* nodes before!

To rectify this suboptimal situation, we will apply a query rewrite that we call *fn:text()* push-up. If we apply this rewrite to the XQGM instance depicted in Figure 4.1, we end up with a modified graph (Figure 4.2) Now, all evaluations of the *fn:text()* function are performed in the top-most Select operator. Consequently, the function is evaluated only for those *name* nodes that are part of the final result. After showing the big picture of the push-up heuristics, we are well-prepared to step on to the specification of the rewrite rules as patterns consisting of a condition (left-hand side) and an action part (right-

hand side). In a classical divide-and-conquer manner, we split this complex task into three patterns that are executed in a sequence: SelectionPushUp, TextAccessIntoMergePushUp, and finally TextAccessIntoSelectPushUp. All patterns can be easily integrated into the pattern-based rewrite engine of XTC (Mathis et al., 2008).

4.1.1 Selection Push-Up

Figure 4.3(a) shows the condition part of the SelectionPushUp pattern: It searches for a unary Select operator that (1) evaluates the *fn:text()* function and (2) has an Access operator as child node. The Access operator receives correlated input from Select operator ② that is, in turn, an ancestor of Select operator ①. On the path between operator ① and operator ②, there may be additional operators whose projection specification must propagate at least those item positions, on which operator ① evaluates the *fn:text()* function and may not contain any value-based predicates. Moreover, the pattern matches only if operator ② propagates the text value to its parent, too.

In Figure 4.3(b), we can see the rewritten query fragment that results from applying the SelectionPushUp pattern. Select operator ① is now the parent of operator ②. The benefit of this rewrite rule may not be obvious at first sight. To understand its real impact on query performance, we have to take a wider perspective: The SelectionPushup pattern may serve as a preliminary step for query unnesting. Query unnesting is a rewrite strategy that iteratively replaces the evaluation of structural predicates using nested and correlated Select operators, for example, as shown in Figures 4.3(a) and 4.3(b), by Structural Join operators, which evaluate structural predicates using set-at-a-time evaluation in contrast to node-at-a-time evaluation. Especially in low-selectivity scenarios, set-at-a-time evaluation outperforms node-at-a-time evaluation (Mathis, 2009). If we apply query unnesting to the query fragment depicted in Figure 4.3(b), we end up with the graph illustrated in Figure 4.4. Let us assume that the Structural Join has a selectivity of 80%. If we would have not applied the SelectionPushUp pattern before, the Select operator, which evaluates the *fn:text()* function, would still reside between the Structural Join and the Access operator. Hence, 20% of the function evalua-

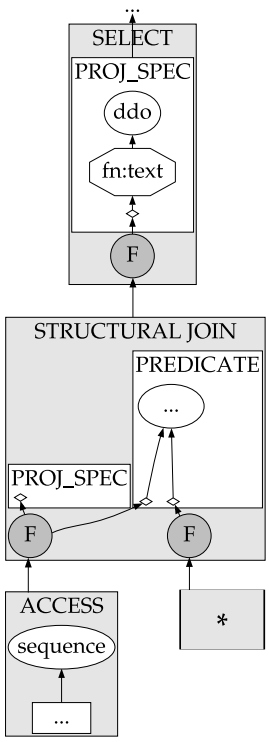
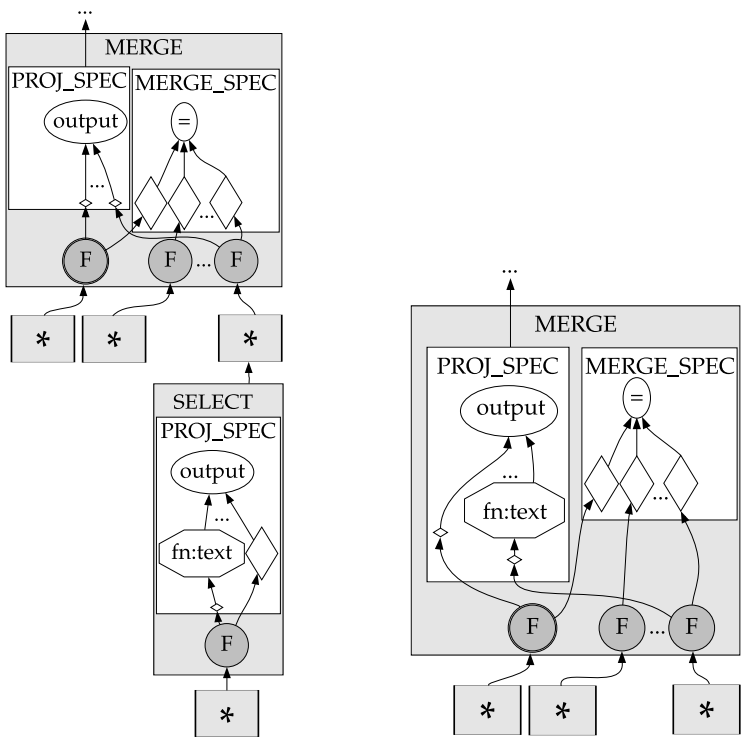


Figure 4.4: Query after unnesting



(a) Left-hand side (b) Right-hand side

Figure 4.5: TextAccessIntoMergePushUp pattern in action

tions are superfluous, because these tuples will not be part of the operator's result. In contrast, after applying the pattern, the function is evaluated *after* the Structural Join and only for those tuples that really satisfied the structural predicate. Even if the structural predicate would have a selectivity of 100%, the query performance of the rewritten query graph would not be affected in a negative way. We will see later in Chapter 12 that the application of the SelectionPushUp pattern will really pay off in terms of a substantial performance gain.

4.1.2 Text Function into Merge Push-Up

The TextAccessIntoMergePushUp pattern helps to push the evaluation of the *fn:text()* function into the projection specification of a Merge operator. Together with the TextAccessIntoSelectPushUp pattern (Section 4.1.3), we can perform the query rewrite already shown in Figures 4.1 and 4.2. In Figure 4.5(a), we can see the conditional part of the TextAccessIntoMergePushUp pattern. First, we are looking for a unary Select operator that (1) evaluates the *fn:text()* function

and whose results are projected out and (2) sent to the Select operator's parent. Starting at the Select operator, we traverse the query graph up to its root, but stop immediately if a Merge operator is found. During traversal, we check for each intermediate operator whether it requires the actual text value, for example, for the evaluation of a predicate. Moreover, we must know whether the text values are always projected out and, hence, finally end up in the Merge operator. If all intermediate operators satisfy these conditions, the pattern matches and the rewrite can be applied. Figure 4.5(b) shows the XQGM fragment after rewrite. Now, the Merge operator evaluates the *fn:text()* function as part of its projection specification, whereupon the superfluous Select operator is removed[1]. If we once again have a look at the XQGM graph shown in Figure 4.1, the TextAccessIntoMergePushUp pattern helps us to push the *fn:text()* function up into the Merge operator. Nevertheless, we have not yet met the goal of our selection push-up heuristics, that is, pushing *fn:text()* function calls as far as possible upwards in XQGM instances.

4.1.3 Text Function into Select Push-Up

Finally, to complete selection push-up, the TextAccessIntoSelectPushUp pattern moves *fn:text()* function calls, which are evaluated in Merge operators, upwards into the top-most unary Select operator.

The TextAccessIntoSelectPushUp rewrite rule looks for Merge operators whose projection specification contains *fn:text()* function calls, for example, as illustrated in Figure 4.5(b).

In this situation, we use the Merge operator as starting point and search for the top-most Select operator that still accesses the tuples produced by the *fn:text()* function call.

If there exist only operators between Select and Merge that (1) do not filter out the *fn:text()* tuples and that (2) do not evaluate value-based predicates requiring the availability of the *fn:text()* tuples, the pattern matches and the rewrite is performed as follows: All accesses to *fn:text()* are moved from the projection specification of the Merge operator into the matched Select operator's projection specification. If the Select operator evaluates a value-based predicate, we have to wrap the corresponding tuple variable reference in the predicate expression with the *fn:text()* function call. Moreover, in the presence of a sorting specification, we have to adjust it accordingly. For example, Figure 4.6 shows the query fragment depicted in Figure 4.5(b) after

[1] In this example, we assume that the Select operator does not evaluate other functions.

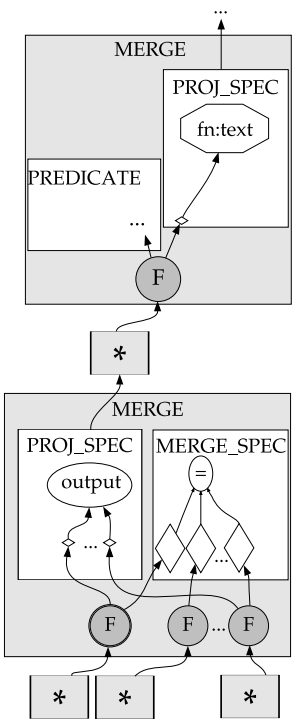


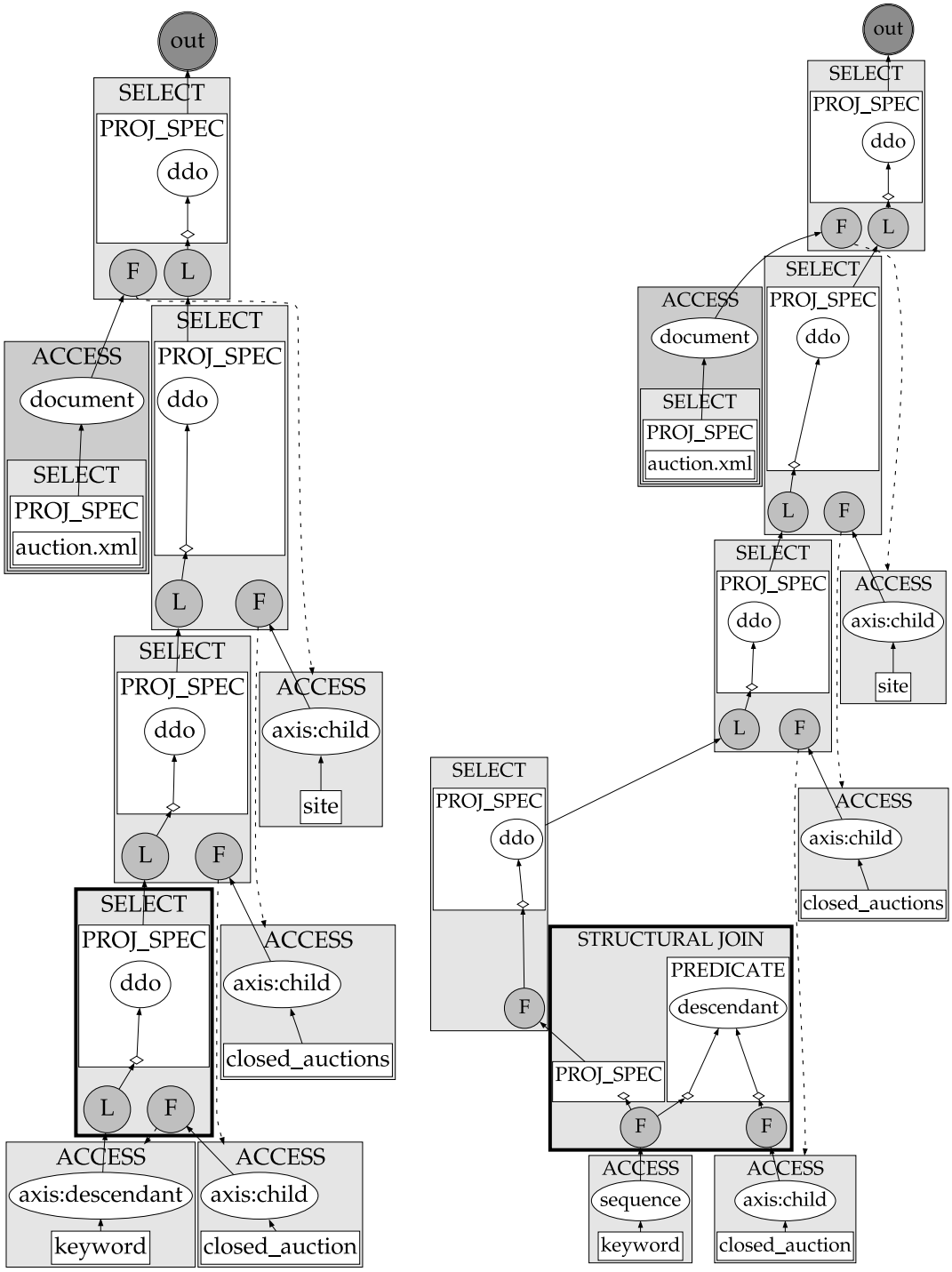
Figure 4.6: After push-up

applying the TextAccessIntoSelectPushUp pattern. Now, the *fn:text()* function is only evaluated for those tuples that "survive" the previous Merge operator.

The three rewrite rules for selection push-up, that we discussed just few lines above, can be easily integrated into the pattern-based query rewrite engine of XTC. They are yet another way of improving query evaluation performance, even before the cost-based query optimizer steps into the scene.

4.2 Cost-Based Query Unnesting

Mathis (2009) introduced *query unnesting*, that is, a means for XML query decorrelation that helps to speed-up XQuery evaluation in low-selectivity scenarios. In Section 2.2 on page 21, we have already shown that the XQuery Core Language (Draper et al., 2007) favors node-at-a-time evaluation for XQuery evaluation. Traditionally, database systems are optimized for set-at-a-time processing. For example, SJs also follow this classical principle. To allow for XQuery evaluation using them, the query evaluation strategy must be trans-



(a) Before unnesting

(b) After first unnesting

Figure 4.7: Cost-aware query unnesting

Query 4.2 XPathMark-A benchmark; query A 3 (Franceschet, 2005)

```
doc("auction.xml")/site/closed_auctions/closed_auction//keyword
```

formed from node-at-a-time processing to set-at-a-time processing (see also Section 2.2 on page 21).

The current approach (Mathis, 2009) handles this transition using a pattern-matching and transformation engine. This engine applies transformations in an eager mode, that is, they are applied as long as a match is found. In contrast to RDBMSs, where algebraic optimization—that is, query rewrite—is purely based on heuristics and only non-algebraic optimization takes cost information into account, we weaken this strict separation in native XDBMSs and make the application of query rewrite rules dependent on their benefit for cost reduction even at the logical level.

Using the refined cardinality inference rules that will be introduced in Chapter 7, we can derive statistical information that help to determine the selectivity of XQGM subexpressions. Based on this information, we are able to control the query unnesting process at a fine-granular level. Hence, we can immediately stop unnesting, if we expect that the rewrite will be counterproductive, that is, results in increased costs.

Let us have a look at an example: Query 4.2 is a simple XPath query that is part of the XPathMark benchmark[2]. Figure 4.7(a) shows the corresponding XQGM representation. So far, query unnesting was not performed, therefore, the dataflow of the query follows the node-at-a-time processing paradigm: for each tuple sequence received by a Select operator via its *for*-quantified tuple variable, the connected location step (e. g., *child::site*) is evaluated using an Access operator that, in turn, sends its output to the consecutive Select operator. This approach resembles the evaluation of nested *for* loops in conventional programming languages like C or of a nested-loops join in RDBMSs.

On the other hand, Figure 4.7(b) shows the same XQGM instance after applying the unnest pattern for the first time. Accordingly, the last location step is evaluated using a single Structural Join operator. Hence, the workflow is now a hybrid of node-at-a-time evaluation and set-at-a-time evaluation.

At this point, you might ask which version can be evaluated more efficiently. The answer to this question mainly depends on the selectivity of the location step. Let us assume that there exist 100 *closed_auction* nodes that satisfy the path expression *site/closed_auctions/closed_auction* and that there are a total number

[2] The benchmark queries are available at: `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html`

of 1,000 *keyword* nodes. In this regard, we distinguish between two fairly antipodal selectivity scenarios: The location step *closed_auction//keyword* is satisfied by:

1. 900 *keyword* nodes (selectivity = 0.9)
2. only 5 *keyword* nodes (selectivity = 0.005)

If we evaluate this subexpression using the nested version as illustrated in Figure 4.7(a), for each *closed_auction*, which serves as context node, we have to traverse the document (e. g., by accessing the DOM interface or by performing a scan over a PAP or SAP) to get the related *keyword* nodes. In scenario 1, this strategy is very expensive, because it causes many random IO operations. In contrast, in scenario 2, this approach is very effective, because there exist only a few *keyword* nodes that potentially cause random IO operations.

In Figure 4.7(b) (unnested version), we access the sequence of all *keyword* nodes. This approach is very effective in scenario 1, because almost all *keyword* nodes contribute to the query result, whereas in scenario 2, many tuples are fetched that do not contribute to the final result. In scenario 1, we consequently opt for the unnested version, whereas in scenario 2, we prefer the nested version to prevent a tremendous performance loss. Both scenarios show two extreme situation that must be considered during query optimization. Therefore, applying query unnesting in an obstinate way is not always a good decision. Now, we have to find a way that helps the rewrite engine to select the proper evaluation strategy. If we recall that database textbooks (e. g., compare Härder and Rahm, 2001) often refer to the classical 1% rule-of-thumb—that is, dependent on disk and file characteristics, sequential IO operations are preferred over random IO operations, if the selectivity of the predicate is larger or equal than 1%—we can use this heuristics as well as a decision criteria for guiding cost-aware query unnesting.

4.3 Summary

In this chapter, we discussed two rewrite techniques that support plan generation—even at the very early stage of algebraic optimization—in preventing bad plans.

First, Section 4.1 discussed three rewrite patterns that help to push-up *fn:text()* accesses as far as possible towards the root of the XQGM graph. In the best case (low predicate selectivity), this rewrite reduces expensive but unnecessary *fn:text()* accesses to a minimum. Even in the worst case, it does not negatively affect the overall query performance.

In Section 4.2, we introduced a decision criteria that allows for cost-aware query unnesting. By taking the estimated selectivity of location steps into account, we can decide whether query unnesting leads to a performance benefit or is counterproductive.

Both rewrite techniques are only complementary means for enabling efficient and effective plan generation, which is at the heart of this thesis. In Chapter 5, we will introduce the plan abstraction that provides the basis for discussing the cost-based optimization and plan generation techniques.

5 Plan Abstraction

"One of the major tasks of Computer Science is systematic abstraction."

(Hartmut Wedekind)

In this chapter, we introduce the plan abstraction that serves as the main data structure in our cost-based query optimization approach.

To allow for a convenient formulation of the query transformation and cardinality inference rules, Section 5.1 introduces a unified notation for plans. In Section 5.2, we discuss the mapping from XQGM instances to plans. Thereafter, Section 5.3 briefly sketches the translation of plans to physical algebra (PAL) operator graphs. Finally, Section 5.4 summarizes this chapter and provides an outlook on the subsequent chapter.

5.1 Introduction

To express and discuss the different query transformation, cardinality inference, and translation strategies, which are introduced in this thesis, we need a formalization of the basic structures that are manipulated by the cost-based query optimizer.

For query optimization in general and plan generation in particular, many XQGM details are irrelevant. Therefore, we abstract from these heavy-weight structures and use a plan abstraction (*plan graph*) that focuses on dynamic properties, for example, the currently assigned implementation or cost values.

Each XQGM operator corresponds to a plan in a plan graph. Table 5.1 summarizes the common plan properties. A plan graph has static and dynamic properties. All *static properties* are derived from the corresponding XQGM instance (e. g., the projection specification or predicates) and are not changed during query optimization. In contrast, *dynamic properties*—which are not present in XQGM instances—, for example, the currently assigned physical operator or cardinality estimates, are continuously manipulated during query optimization.

By applying a logical query transformation rule on a plan graph G, we get a semantically equivalent plan graph G' (Definition 5.1).

	Property	Description	Notation
Static	Predicate	Predicate of the corr. XQGM operator	p_{pred}
	Projection spec.	Items $i_1, \ldots, i_n$ that are projected out	$p[\text{P} : i_1, \ldots, i_n]$
	Sorting specification	Items $i_1, \ldots, i_n$ that must be sorted in ascending document order (ASC) or descending document order (DESC)	$p[\text{S} : i_1(\text{order}), \ldots, i_n(\text{order})]$
	Tuple variables	Tuple variables of XQGM operator	p_{tupVars}
Dynamic	CPU cost	Estimated CPU cost	p_{CPUcost}
	IO cost	Estimated IO cost	p_{IOcost}
	Output cardinality	Estimated output cardinality	$\|p\|$
	Implementation	Currently assigned implementation	p_{impl}
	Parent	The plan's parent node	p_{parent}
	Children	List of the plan's child nodes	p_{children}
	Goal state	Indicator for plan generation	p_{isGoal}
	Abstract domain identifiers	Abstract domain identifiers derived during cardinality estimation	p_{adi}

Table 5.1: Common plan properties

Definition 5.1 (Semantic Equivalence) *Two plan graphs G and G' are called* semantically equivalent *(we write $G \equiv G'$), if and only if G and G' produce exactly the same output tuple sequence.*

During cost-based query optimization, the plan generator derives numerous semantically equivalent plans. Using the cardinality inference rules and the cost model, the plan generator assigns to each plan the corresponding IO and CPU costs as well as the estimated output cardinality. Based on this information, the plan generator determines the cheapest plan and finally translates it into an *Query Execution Plan* (QEP) (Definition 5.2).

Definition 5.2 (Query Execution Plan) *A* Query Execution Plan *$Q(O, E)$ is a tree-structured graph, where O is the set of physical algebra (PAL) operators and $E \subseteq O \times O$ resembles the parent-child relationship between operators.*

Besides the common properties, each operator class may inherit complementary properties from their corresponding XQGM instances. Before we discuss the properties in detail, Table 5.2 shows the different plan types supported by our plan generator.

Accesses to the document root are handled by the DocAccess plan, whereas the Access plan can return attribute nodes and element node sequences; but can also be used for the evaluation of context-dependent location steps (node-at-a-time processing).

In XQGM, projection, sorting, and duplication elimination are integral parts of operators and expressed using the projection specification or the sorting

Plan	XQGM operator	Description
DocAccess	Access operator	Only for document root accesses
Access	Access operator	For attribute, node, and sequence accesses
DDO	Arbitrary operator	Created for each operator that performs duplicate elimination
Project	Arbitrary operator	Created for each operator that has a projection specification
Sort	Arbitrary operator	Created for each operator that contains a sorting specification
GroupBy	GroupBy operator	–
Unnest	Unnest operator	–
Merge	Merge operator	–
Select	Select operator	–
Split	Split operator	–
SplitRef	–	Handles references to Split
ParentResolution	–	Only in combination with CAS index scans
Reference	Tuple variable ref.	–
Union	Set operator	–
Intersect	Set operator	–
Difference	Set operator	–
StructuralJoin($\bowtie$)	Structural Join op.	–
TwigJoin	–	No XQGM equivalent, because no twig discovery is performed at the logical level
IndexAccess	–	Derived during plan generation

Table 5.2: Plan overview

specification. To enhance the flexibility of plan generation and for simplifying the subsequent mapping onto physical algebra operators, we use three different plans to express these tasks—DDO, Project, and Sort, respectively.

Set operations are represented as a single operator in XQGM, where a flag distinguishes between the three actual operations: union, intersection, and difference. In contrast, we use three different plans (Union, Intersect, and Difference) to express the semantics of the three set-based operations, mainly to enhance plan-generation flexibility, too.

The Split plan sends its input to multiple consumers. To reduce the complexity during plan generation—obviously, trees can be handled more easily than graphs—, every Split plan has only one real consumer, whereas the remaining $n-1$ consumers receive their input from a dummy SplitRef plan that—as indicated by its name—simply references the corresponding Split plan. The Reference plan is a substitute for XQGM tuple variable references (Mathis, 2009).

Plan	Property	Description	Notation
DocAccess	Document name	–	p_{docName}
	Collection name	Name of document collection	p_{collName}
Access	Document name	–	p_{docName}
	Collection name	Name of document collection	p_{collName}
	Node test	Predicate on input	p_{nodeTest}
	Access type	Node, sequence, or attribute	$p_{\text{accessType}}$
	Axis	Axis, if access type is node access	p_{axis}
	Correlated op.	ID of correlated operator, if access type is node access	p_{corrID}
Sort	Order spec.	–	$p_{\text{orderSpec}}$
GroupBy	Nesting spec.	–	p_{nestSpec}
Unnest	Nesting spec.	–	p_{nestSpec}
Merge	Merge spec.	–	$p_{\text{mergeSpec}}$
StructuralJoin	Structural pred.	$a_i\,\theta\,b_j$, where θ is an XPath axis and a_i, b_j are tuple items	$\bowtie_{a_i\,\theta\,b_j}$
	Output order	Result sorted by left or right input	$\bowtie_{\text{outOrder}}$
IndexAccess	Index definition	–	p_{indexDef}

Table 5.3: Additional plan properties

The ParentResolution plan can be injected above CAS index scans, if a subsequent plan requires the parent nodes of the content values stored in the index. The semantics of the ParentResolution plan is: For every input tuple, it returns the corresponding parent node and removes duplicates if the parent occurs more than once.

For the representation of the various join types, we use the standard symbols: $\ltimes_p$ (structural left-semi join), $\rtimes_p$ (structural right-semi join), $\bowtie_p$ (structural full join), $⟕_p$ (structural left-outer join), and $⟖_p$ (structural right-outer join). The structural join evaluates a structural predicate p described as follows: $a_i\,\theta\,b_j$, whereupon a_i is an item of tuple sequence q_1, b_j is an item of tuple sequence q_2, and θ is an XPath axis, for example, *descendant*.

A closer look on Table 5.2 reveals that there are two plans that have no correspondent XQGM operator: TwigJoin and IndexAccess. As we have argued in Section 3.2.1 on page 53, we do not apply twig discovery (compare Mathis, 2009) during logical query rewrite. Instead, we will consider HTJs during cost-based plan generation as alternative implementations for SJs. Furthermore, IndexAccess plans are derived during query transformation and may be an alternative way for evaluating cascades of SJs or single HTJs using TAPs.

In Table 5.3, we summarize the additional plan properties. The DocAccess operator additionally provides information on the document name and the collection it belongs to. Likewise, the Access operator may have a node test

(predicate) and can access different types of input. For example, a sequence of nodes (set-at-a-time processing) or attribute node values. Also, it can evaluate a location step or a predicate step with respect to an XPath axis using the tuples emitted by a correlated operator as context nodes.

Sort, GroupBy, Unnest, and Merge are enriched with information about the output ordering ($p_{\text{orderSpec}}$), which item positions shall be unnested (p_{nestSpec}), according to which item positions nesting shall be performed (p_{nestSpec}), and which item positions must be merged ($p_{\text{mergeSpec}}$), respectively.

The StructuralJoin operator contains a structural predicate, that is, it evaluates an XPath axis between its left and right input tuple sequences. Moreover, its output order property describes whether the output is sorted according to the left or right tuple sequence. Finally, IndexAccess plans allow to access the corresponding index definition.

5.2 From XQGM instances to Plan Graphs

Mapping XQGM instances onto plans results in an almost identical graph. Nevertheless, subtle differences exist. To explain the mapping rationale, let us reconsider the slightly modified XQuery expression defined in Section 4.1 on page 63. Figure 5.1 illustrates the associated XQGM graph after applying all rewrite rules described in Mathis (2009) as well as those introduced in Chapter 4.

For deriving the plan, we traverse the XQGM graph in left-most depth-first order and map each XQGM plan onto its corresponding plan. For each XQGM instance, we simply copy the properties listed in Table 5.1 and Table 5.3. Whenever an XQGM operator contains a projection specification, a sorting specification, or an order specification, we "inject" the corresponding plan operators into the plan. Sometimes, XQGM instances are not trees—but pure graphs. XQGM instances may degenerate to graphs, because Split operators may supply several operators with its output. Moreover, tuple variable references can be connected to operators that must not be their parent or child nodes. In general, graph structures make plan generation more difficult—this is especially true for bottom-up plan generation[1]. During XQGM traversal, we map only the first occurrence of a Split operator onto a Split plan. Whenever we touch the Split operator once again during traversal, we use a SplitRef plan instead, which "virtualizes" the actual occurrence. The same idea is used to capture tuple variable references. When plan generation is finished, the virtual references to the Split can be easily reconstructed.

[1] The recent article by Neumann and Moerkotte (2009) might help to overcome this problem.

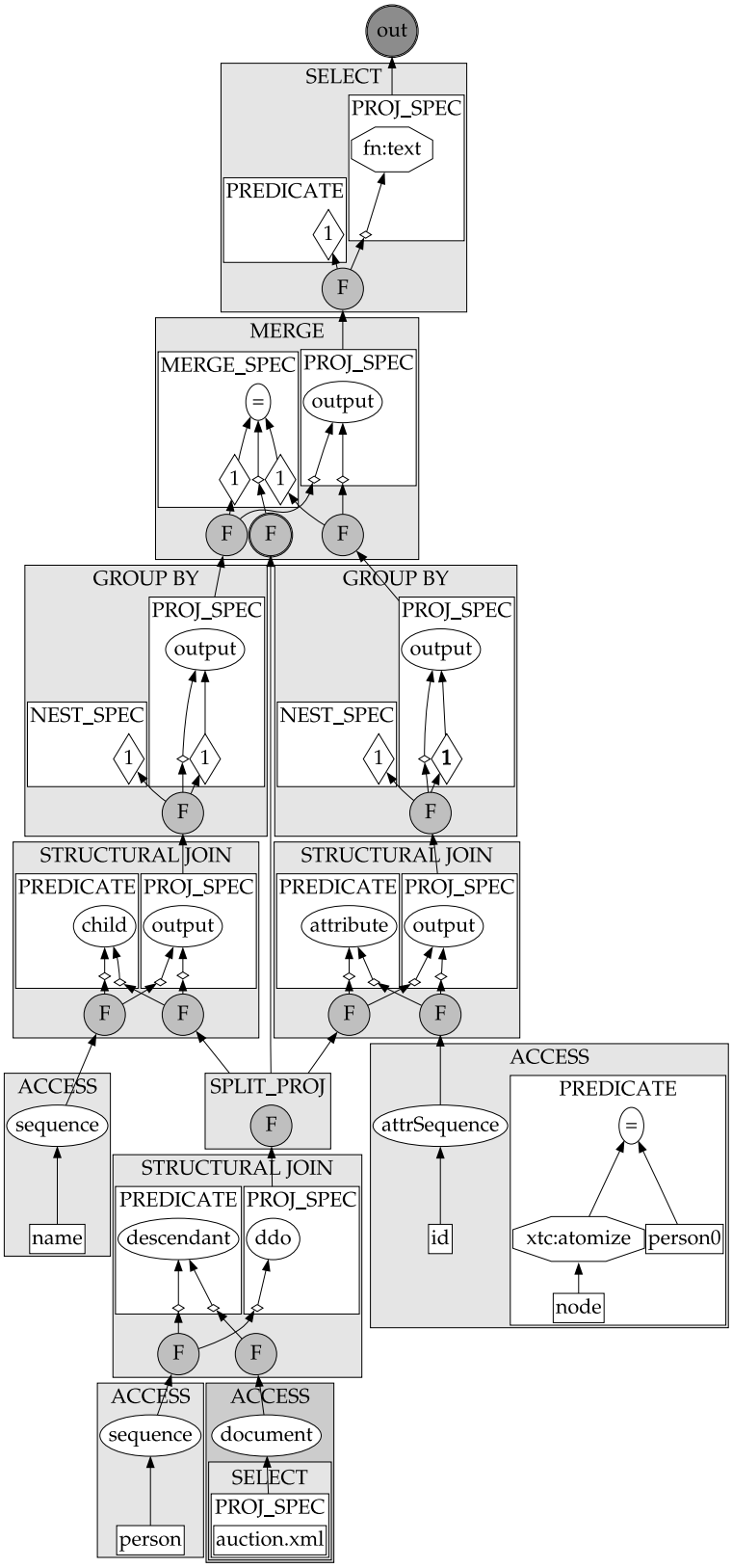
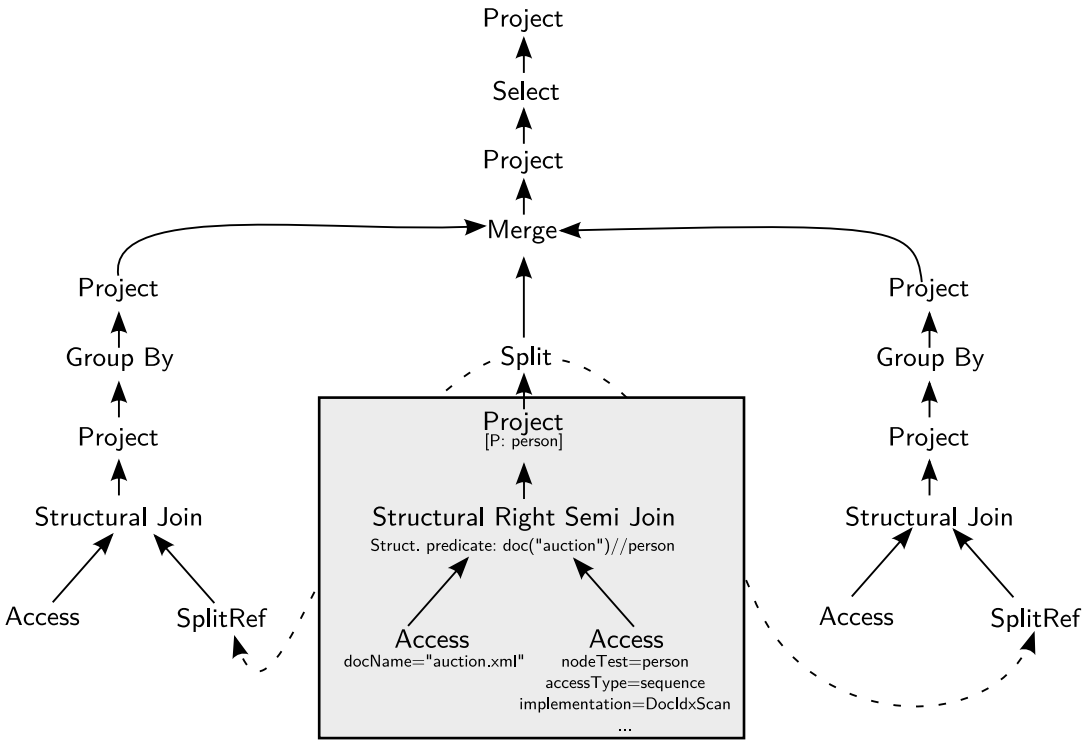
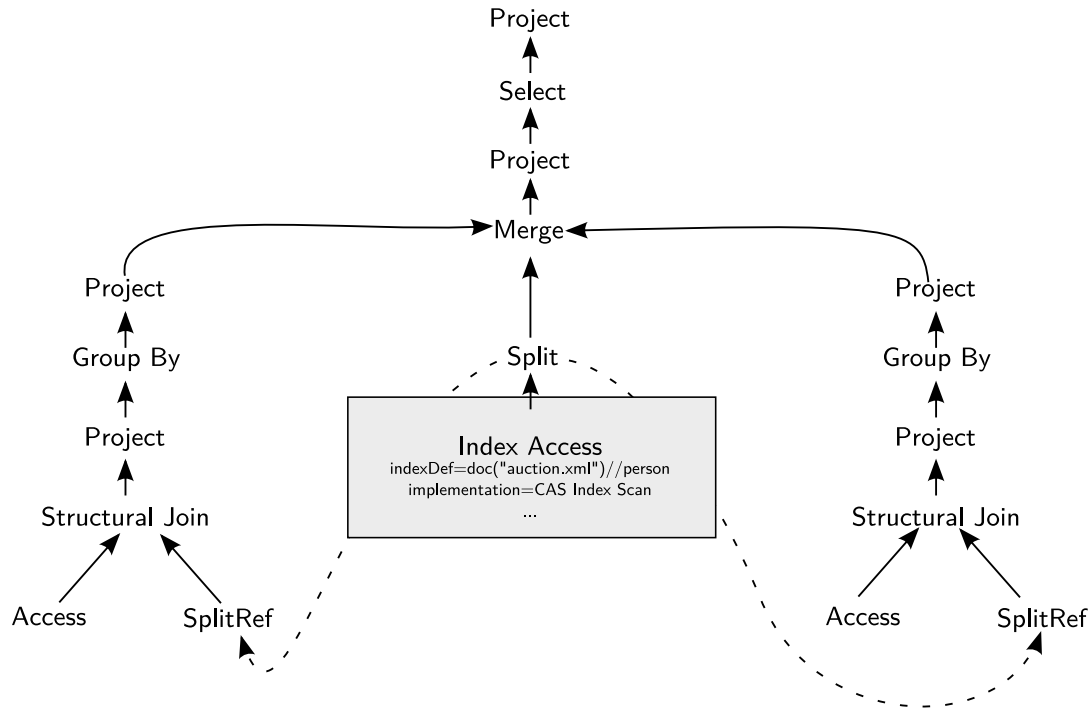


Figure 5.1: XQGM instance for XMark benchmark query Q1



(a) Initial plan graph after XQGM-to-plan mapping



(b) Plan graph after rewrite

Figure 5.2: Plan graph before and after logical query transformation

Figure 5.2(a) shows the plan graph that is derived from the XQGM instance illustrated in Figure 5.1. You can see the various Project operators that were injected during XQGM-to-plan mapping. Moreover, two SplitRef operators are now "virtually" connected to the Split operator (dotted edges).

As we pointed out before, plan graphs mainly focus on the structure of the query as well as the details that are currently relevant for plan generation. The grey box around the subtree of the first location step—*doc("auction.xml")//person*—shows some details. The right-most Access operator performs a node test on *person* nodes. Moreover, the plan generator assigned a document index scan as physical access path. For the remainder of this work, we will only describe the subtrees that are relevant for a specific plan generation task, for example, plan transformation or cost estimation, in detail and keep the remaining parts of the graph as abstract as possible to prevent us from drawing off the attention from important details.

The plan generator will modify the plan graph shown in Figure 5.2(a) in manifold ways. For example, the highlighted subtree could be evaluated using a single TAP. Figure 5.2(b) illustrates this possible rewrite, where the four operators are replaced by a single IndexAccess plan that could be implemented using a scan over a CAS index for path *doc("auction.xml")//person*.

Before, we roughly sketched the mapping from XQGM instances onto plan graphs. The intention was just to give you an idea how it works in principle. Definitely, a more in-detail discussion is necessary for completely understanding this initial step of plan generation. Therefore, Section 10.1.1 on page 160 will treat the implementation of the XQGM-to-plan mapping more comprehensively in the context of the query optimizer's system architecture.

5.3 From Plan Graphs to Query Execution Plans

Mapping a plan graph onto an QEP is the final step of plan generation. In most cases, this procedure results in a one-to-one mapping from plans to physical operators, where a plan p is mapped onto the physical operator that is specified by the p_{impl} property.

Let us reconsider the query illustrated in Figure 5.1 and the corresponding plan graph depicted in Figure 5.2(a), which is used as data structure for plan generation. In Figure 5.3, we can see a possible QEP that was derived by the cost-based query optimizer. In this case, the first location step (*doc("auction.xml")//person*) is evaluated using the structural join algorithm *StackTree* that receives the sequence of *person* nodes by an *Element Index Scan*. The location step between *person* nodes and *name* nodes is evaluated using

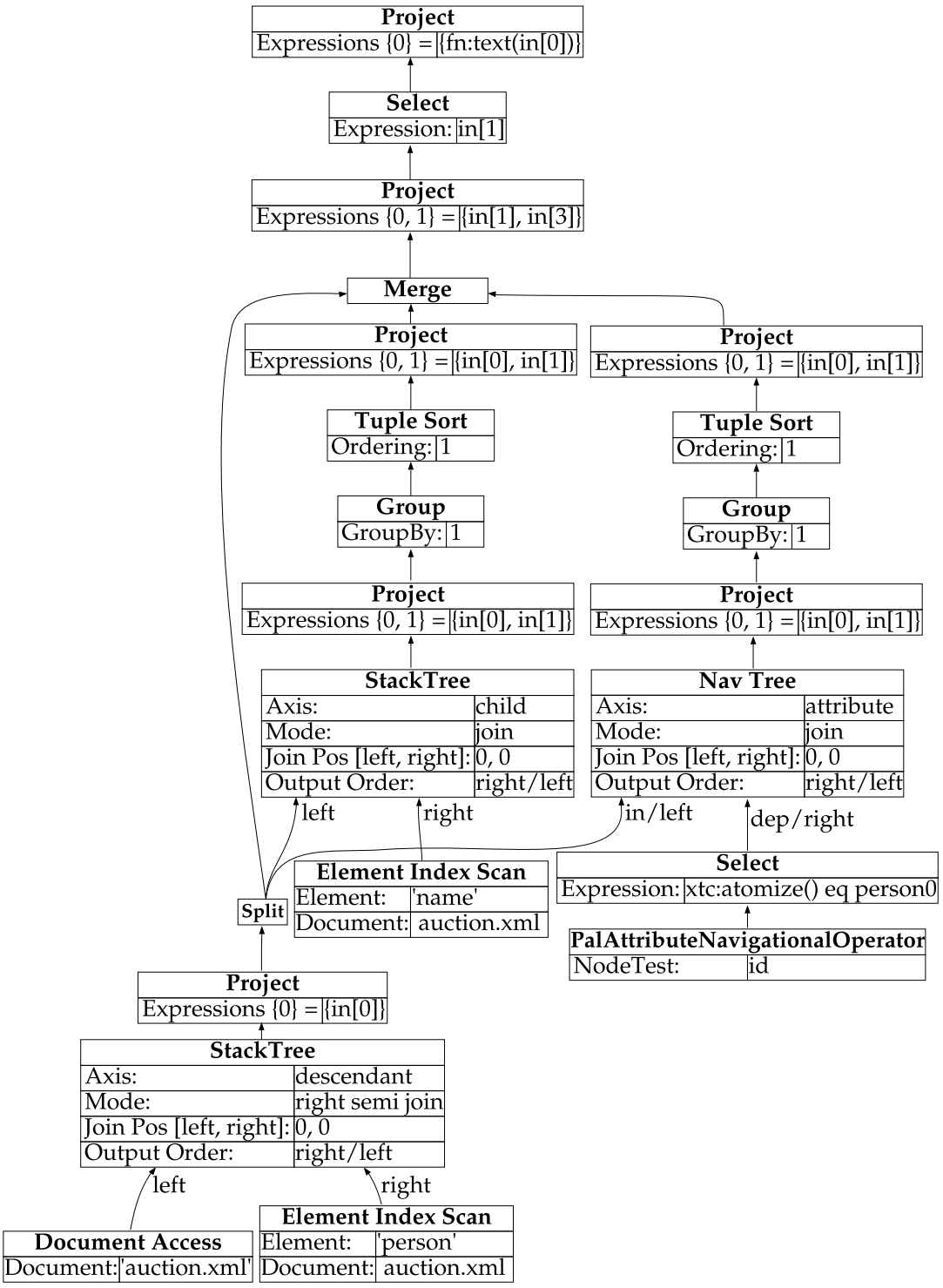


Figure 5.3: A possible QEP for XMark query Q1

StackTree, too. In contrast, for accessing the *id* attributes, a navigational operator (*PalAttributeNavigationalOperator*) is used. For the remaining operators, a direct translation from plan operators to their physical counterparts is done.

Technically, the plan-to-PAL mapping is performed almost the same way as in the heuristics-based query engine (Mathis, 2009). Therefore, in Section 10.1.7 on page 169, we will only roughly sketch in which way our translation procedure differs from the original one.

5.4 Summary

In this chapter, we provided the preliminaries for the discussion of our query optimization approach. Therefore, we introduced the notion of a *plan graph* that consists of *plan* nodes.

Plan graphs are the data structures that are manipulated by the query optimizer introduced in this work (Section 5.1). Plan graphs are an abstraction of XQGM instances that focus on their structure and help to define query transformation and cardinality inference rules more easily. In addition to all properties inherited from the corresponding XQGM instances, which are static, plans have additional dynamic properties that are manipulated during query optimization, for example, CPU cost or the currently assigned implementation.

In Section 5.2 and in Section 5.3, we explained how to transform XQGM instances into plan graphs and in which way they are finally transformed into QEPs.

In Chapter 6 to Chapter 9, we provide the basic ingredients for cost-based query optimization: Chapter 6 introduces a set of plan transformation rules that allow to span the search space. Next, Chapter 7 provides a comprehensive set of rules for cardinality inference that allow for an approximation of the value domains that are taken by XQGM operators at runtime. Thereafter, Chapter 8 will elaborate on our cost model that we use to assign cost factors to plans. Finally, Chapter 9 outlines the bottom-up and top-down plan generation algorithms.

6 Query Transformation

"It is an ill plan that cannot be changed."

(Latin proverb)

In this chapter, we describe various logical query transformation rules for deriving semantically equivalent plans. First, Section 6.1 provides a general discussion of the importance of transformation rules. Next, Section 6.2 introduces the various implementations that can be considered as alternatives.

In Section 6.3, we discuss several structural variations of plans. Though, structural variations of value-based joins, which are discussed in Section 6.3.1, are well-known from the relational scene, subtle differences exist in the XQuery context. Effective rewriting of structural joins is necessary for providing sufficient performance for XQuery expressions. Hence, Section 6.3.2 discusses the set of join associativity rules and the join commutativity rule for structural joins. Join fusion (Section 6.3.3) is yet another logical query transformation that helps to consider holistic twig joins as an alternative for twig query patterns. The most promising rule in terms of expected performance gain is TAP detection (Section 6.3.4). This approach helps to exploit TAPs for query evaluation. Thereafter, Section 6.4 discusses a comprehensive example. Finally, Section 6.5 summarizes this chapter and provides a glimpse on subsequent chapters.

6.1 Introduction

The primary goal of query optimization in database systems is deriving alternative implementations and evaluation orders for given queries as well as identifying the most promising variant. Logical query transformations enable a query optimizer's plan generator to create alternative expressions. The remaining task is left to other optimizer components.

In general, search spaces for database queries have two orthogonal dimensions that are open for variations (Figure 6.1). First, there might exist various *implementations* for a single plan operator. For example, an Access plan can be implemented using a DOM-navigational operator or a scan over the document

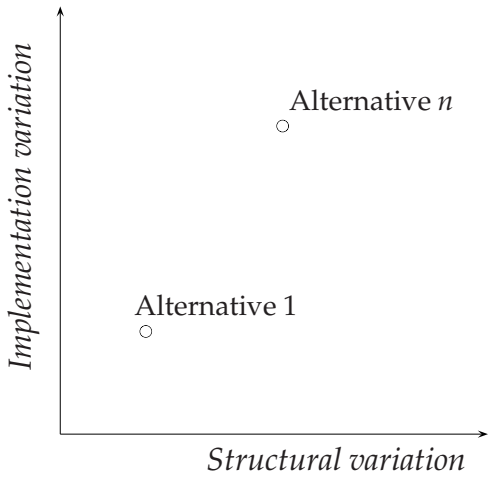


Figure 6.1: The two dimensions of logical query transformations

index. On the other hand, there are manifold ways for varying the structure of plans. For example, the evaluation order of a subtree can be modified (e. g., join reordering), the input order of an operator can be changed (e. g., join commutativity), a cascade of non-leaf nodes can be substituted by an operator (e. g., join fusion), and finally, a subtree can be completely replaced by a single operator (e. g., TAP detection).

If we compare native XDBMSs and RDBMSs with respect to their possible search space variations, native XDBMSs offer a much larger variety of operators: There is a plethora of access paths that can be considered in the *implementation variation* dimension. Moreover, structural relationships, in addition to classical value-based relationships, increase the possibilities in the *structural variation* dimension, too.

Structural relationships are of utmost importance in native XDBMSs. Often, for identifying an element in an XML document, a path expression consisting of several location steps must be evaluated. Traditionally, such path expressions are evaluated using cascades of structural join operators. Even if we only consider the structural variation dimension and neglect implementation variation, the number of possible evaluation orders increases—in the worst case—exponentially with the total number of joins involved. As a consequence, search spaces in native XDBMSs can become even larger than in RDBMSs. We will see later in Chapter 9 which plan generation strategies are able to handle these new challenges.

6.2 Implementation Variation

In this section, we will discuss the *implementation variation* dimension. Mathis (2009) already introduced the physical implementations (PAL operators) for various XQGM operators. We retain these alternatives, but allow the plan generator to choose the best implementation depending on its cost, rather than on hard-wired recommendations.

6.2.1 Overview

For your convenience, we summarize the different implementations for plans in Table 6.1 on page 88. Selecting the cheapest access path for fetching single nodes or node sequences is crucial for the performance of QEPs. For Access plans, we support five different access methods:

- *Document Index Scan*: This operator performs a scan over the document index to get a node sequence (e. g., all *book* element nodes) and may evaluate a value-based predicate.
- *Navigational Operator*: For each XPath axis, there exists a specific navigation operator that retrieves for a given context node all nodes that are structurally related to it via the given XPath axis. For example, for the *child* axis, this operator is called *Child Navigational Operator*.
- *Element Index Scan*: This operator allows to fetch all element nodes having the same element name from the element index.
- *Path Index Scan*: This operator can be used for retrieving attribute or element nodes from a path index that contains all paths ending on a given element or attribute name.
- *CAS Index Scan*: If an Access plan evaluates a value-based predicate on its input element or attribute sequence, a CAS index that contains all paths ending on the specified element or attribute name might be the appropriate access path.

Currently, StructuralJoin plans are evaluated using *Extended StackTree* and *NavTree*. Extended StackTree is a classical structural join operator (compare Section 2.2.1), whereas NavTree's evaluation scheme resembles that of traditional nested-loops joins in RDBMSs.

The Select plan has five different implementations that are strongly dependent on the properties of the plan:

Plan	PAL Operator	Description
Access	Document Index Scan	–
	Navigational Operator	Navigational access with respect to a context node; one for each XPath axis
	Element Index Scan	–
	Path Index Scan	Only for single loc. steps, e. g., *//book*
	CAS Index Scan	Only for single loc. steps, e. g., *//title*
IndexAccess	Path Index Scan	For arbitrary document paths
	CAS Index Scan	For arbitrary document paths
StructuralJoin	Extended StackTree	A pedigreed structural join operator
	NavTree	Similar to a relational nested-loops join
Select	Value-Based Hash Join	Binary Select, value-based eq. pred.
	Value-Based Merge Join	Binary Select, value-based pred.
	Value-Based Nested-Loops Join	Binary Select, value-based pred.
	Select Operator	Multi-way Select operator with pred.
	Select Operator with Lazy or Eager Tuple Generator	Evaluation of *for-let* bindings
Twig Join	Extended TwigOpt	–
DocAccess	Document Access	For document root access
DDO	DDO Operator	For duplicate elimination
Project	Project Operator	–
Sort	Tuple Sort Operator	–
GroupBy	GroupBy Operator	–
Unnest	Unnest Operator	–
Merge	Merge Operator	–
Split	Split Operator	–
Reference	Tuple Access Operator	
Union	Union Operator	–
Intersect	Intersect Operator	–
Difference	Except Operator	–

Table 6.1: Implementation alternatives for plans

- *Value-Based Hash Join*: This alternative can be used for the implementation of binary Select plans (having exactly two tuple variables) that evaluate a value-based equality predicate.
- *Value-Based Merge Join*: Using this operator, binary Select plans can be implemented that evaluate either value-based equality or non-equality predicates.

- *Value-Based Nested-Loops Join*: This operator represents yet another implementation for evaluating value-based joins.
- *Select Operator*: This is a general alternative for Select plans that may evaluate arbitrary predicates and can have multiple inputs.
- *Select Operator with Tuple Generator*: This alternative is used for Select plans that mimic the *for-let* bindings that could not be transformed to structural joins in the course of query unnesting (compare Mathis, 2009).

As we have argued in Section 3.2.1, HTJs are not as important for efficient query evaluation as earlier expected. Nevertheless, they are an alternative way for twig query evaluation that must be considered by a typical native XDBMS query engine. TwigJoin plans can be derived from cascades of StructuralJoin plans using the *join fusion* transformation rules (Section 6.3.3). Today, we can only dispose of a single—but very powerful—HTJ operator *Extended TwigOpt* (Mathis, 2009).

For the remaining operators, for example, Project or Split, we currently do not provide alternative implementations. Though, it might be helpful to add further alternatives in the future. For example, additional implementations for Sort plans, for example, specialized external-sorting algorithms, could be beneficial.

6.2.2 Rules for Implementation Variation

After providing an overview of the various implementation alternatives for plan operators, we are now ready to discuss the implementation variation rules for Access, IndexAccess, StructuralJoin, and Select in detail.

For the remainder of this chapter, we use the following notation to describe the query transformation rules: the implementation of plan p can be changed from i_1 to i_2, if and only if the *condition* is satisfied, where plan p' is an exact copy of p that only differs in the selected implementation[1]:

$$\frac{p_{\text{impl}=i_1} \wedge \text{condition}}{p'_{\text{impl}} \longleftarrow i_2}$$

For Access plans, a *Document Index Scan* is the default implementation. Figure 6.2 on page 90 shows the corresponding implementation variation rules.

Rule IS-ACCESS-1 shows the implementation variation for an Access plan that handles node accesses. Here, a *Navigational Operator* can be used as an

[1] Please note, we use the "⟵" symbol for assignments and the "=" symbol for comparisons.

$$\frac{\text{Access } p \wedge p_{\text{impl=Document Index Scan}} \wedge p_{\text{accessType=node}} \wedge p_{\text{axis} \neq null} \wedge p_{\text{corrID} \neq -1}}{p'_{\text{impl} \longleftarrow \text{Navigational Operator}}} \quad \text{(IS-ACCESS-1)}$$

$$\frac{\text{Access } p \wedge p_{\text{impl=Document Index Scan}} \wedge p_{\text{nodeTest=}e_1} \wedge p_{\text{accessType=sequence}} \wedge p_{\text{collName=}c_1} \wedge p_{\text{docName=}d_1} \wedge \text{indexAvailable}(d_1, c_1, e_1, \text{Element Index})}{p'_{\text{impl} \longleftarrow \text{Element Index Scan}}} \quad \text{(IS-ACCESS-2)}$$

$$\frac{\text{Access } p \wedge p_{\text{impl=Document Index Scan}} \wedge p_{\text{nodeTest=}e_1} \wedge p_{\text{accessType=sequence}} \wedge p_{\text{collName=}c_1} \wedge p_{\text{docName=}d_1} \wedge \text{indexAvailable}(d_1, c_1, /\!/e_1, \text{Path Index})}{p'_{\text{impl} \longleftarrow \text{Path Index Scan}}} \quad \text{(IS-ACCESS-3)}$$

$$\frac{\text{Access } p \wedge p_{\text{impl=Document Index Scan}} \wedge p_{\text{nodeTest=}e_1} \wedge (p_{\text{accessType=sequence}} \vee p_{\text{accessType=attribute}}) \wedge p_{\text{pred=}p_1} \wedge p_{\text{collName=}c_1} \wedge p_{\text{docName=}d_1} \wedge \text{indexAvailable}(d_1, c_1, /\!/e_1, p_1, \text{CAS Index})}{p'_{\text{impl} \longleftarrow \text{CAS Index Scan}}} \quad \text{(IS-ACCESS-4)}$$

Figure 6.2: Implementation Variation for Access plans

alternative implementation for a document index scan, if the axis and the context node is provided.

An *Element Index Scan* is a possible implementation for an Access plan, if there exists an element index (function indexAvailable) that contains all elements having name e_1 in document d_1 of collection c_1 (Rule IS-ACCESS-2).

A *Path Index Scan* (Rule IS-ACCESS-3) may be employed, if this index allows to evaluate the path expression $/\!/e_1$, that is, it contains all paths of document d_1 that end on element e_1.

Finally, Rule IS-ACCESS-4 describes the precondition that qualifies a *CAS Index Scan* as complementary implementation. The Access plan may access a node sequence or an attribute node and evaluate a value-based predicate. If subsequent operators do not need the content values returned by the CAS index scan, but require their parent (element or attribute) nodes instead, a

$$\frac{\mathsf{StructuralJoin}\, p \;\wedge\; p_{\text{impl=Extended StackTree}}}{p'_{\text{impl} \longleftarrow \text{NavTree}}} \qquad \text{(IS-SJ)}$$

Figure 6.3: Implementation Variation for StructuralJoin plans

ParentResolution plan must be injected between the Access operator and its parent operator.

As we have already pointed out before, IndexAccess plans are created during TAP detection. In this case, we do not need specific implementation variation rules, because during TAP detection, for every matching index, a separate plan is created.

For StructuralJoin plans, we can currently dispose of two different evaluation algorithms: *NavTree* and *Extended StackTree*. Figure 6.3 shows the corresponding IS-SJ rule. Here, the Extended StackTree algorithm is the default implementation for StructuralJoin plans.

For Select plans that evaluate a value-based predicate, we use the *Value-Based Nested-Loops Join* algorithm as default implementation. As an alternative implementation for it, the plan generator can use the *Value-Based Merge Join* operator, if the inputs of the Select plan are sorted in document order[2] (Figure 6.4 IS-SELECT-1) .

If a Select plan evaluates a value-based equality predicate, the *Value-Based Hash Join* algorithm qualifies as alternative implementation. Figure 6.4 shows the corresponding implementation variation rule IS-SELECT-2.

6.3 Structural Variation

After introducing the implementation variation rules in Section 6.2, we can now focus on the second aspect of query transformation. Structural variations are possible in two dimensions: First, they can replace the evaluation order of operators. Second, several adjacent operators may be replaced by a single one.

In some cases, it is more intuitive for the reader to get a visual impression of the plan structure and its equivalent rewrite, instead of simply looking at mere inference rules. Therefore, we will use the following notation $A \equiv B$, where A and B are plans, to indicate that A and B are semantically equivalent (see Definition 5.1 on page 76).

[2]This property can be checked using the function isInDocOrder.

$$\frac{\begin{array}{c}\textsf{Select}\ p\ \wedge\ p_{\text{impl=Value-Based Nested-Loops Join}}\ \wedge \\ p_{\text{pred}=p_1}\ \wedge\ \text{isValueBased}(p_1)\ \wedge\ \text{isInDocOrder}(p_{\text{children}[0]})\ \wedge \\ \text{isInDocOrder}(p_{\text{children}[1]})\end{array}}{p'_{\text{impl}\longleftarrow\text{ Value-Based Merge Join}}} \quad \text{(IS-SELECT-1)}$$

$$\frac{\begin{array}{c}\textsf{Select}\ p\ \wedge \\ p_{\text{impl}\neq\text{Value-Based Hash Join}}\ \wedge \\ p_{\text{pred}=p_1}\ \wedge\ \text{isValueBased}(p_1)\ \wedge\ \text{isEquality}(p_1)\end{array}}{p'_{\text{impl}\longleftarrow\text{ Value-Based Hash Join}}} \quad \text{(IS-SELECT-2)}$$

Figure 6.4: Implementation Variation for Select plans

6.3.1 Rewrite of Value-Based Joins

Everybody that dealt with RDBMSs in the past, knows value-based joins very well. In native XDBMSs, they are complementary to structural joins. Figure 6.5 shows an XQGM fragment of a value-based join. Select operator ① iterates over the input sequence bound to the *for*-quantified tuple variable. Each item is sent to Select operator ②'s *let*-quantified tuple variable as current evaluation context. Now, operator ② iterates over the tuple sequence provided by the operator bound to its *for*-quantified tuple variable and checks using the *fn:data()* function which items have the same content value as the current context item, which was provided by operator ①. For each match, the result is propagated to operator ①. Operator ① emits tuples where each sequence of matches received from operator ② is nested below the current context tuple. Therefore, a certain order is predetermined that must be preserved by a query transformation rule.

In the relational world, value-based join commutativity is a well-known transformation rule, where the left and the right join partner are exchanged. This is possible, because the classical relational data model does not support nested tuples. In contrast, for value-based joins in XQGM, join commutativity would mean to exchange the subtrees outlined with $*$ below operator ① and ②, respectively. By doing so, we would violate the nesting order proposed in the original XQGM graph. Hence, join commutativity is not possible for value-based joins.

Now, the question arises whether value-based joins in native XDBMSs are at least associative, that is, allow for exchanging the join order of two consecutive

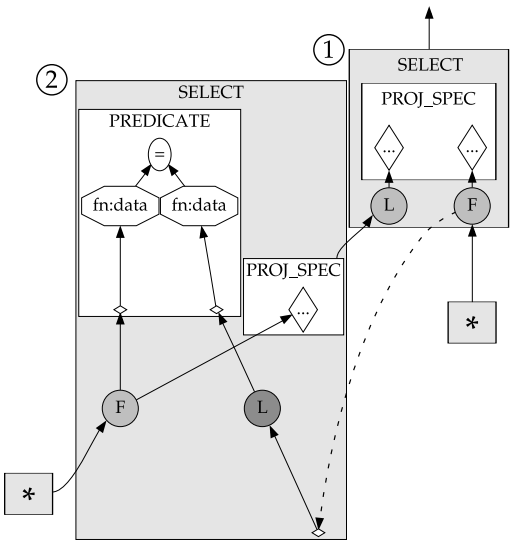


Figure 6.5: XQGM fragment of a value-based join

joins. Unfortunately, here, the situation remains unchanged: In general, it is not possible to preserve the correct nesting order. Therefore, join associativity is also not an option for value-based joins in native XDBMSs.

Though we cannot do anything on the logical level for speeding-up value-based join evaluation, we can at least influence the way how these joins are evaluated on the physical level. Let us assume the query optimizer decides to evaluate the value-based join depicted in Figure 6.5 using the *Value-Based Hash Join* operator. In this case, the performance-critical question is, for which input shall the hash table be built and which input is probed against the hash table. Traditionally, the hash table is built for the smaller input sequence and the larger input sequence is probed against it (compare Härder and Rahm, 2001). To estimate the tuple sequence sizes, which help to answer this question, we can employ the cardinality inference rules that will be presented in Chapter 7.

6.3.2 Rewrite of Structural Joins

Structural joins are the main ingredients of native XML query processors. This class of operators has—under certain conditions—the same properties as relational joins: *commutativity* and *associativity* (Weiner et al., 2008b).

$$\frac{\mathsf{StructuralJoin}\ p \wedge p_{\text{axis}\neq\texttt{self}}\ a \wedge p_{\text{children}[0]}\ c_0 \wedge p_{\text{children}[1]}\ c_1}{p'_{\text{axis}\longleftarrow\text{invertAxis}(a)} \wedge p'_{\text{children}[0]\longleftarrow c_1} \wedge p'_{\text{children}[1]\longleftarrow c_0}} \quad \text{(SJ-COMM)}$$

Figure 6.6: StructuralJoin commutativity

Join Commutativity

Join commutativity, that is, the exchange of the left and the right input of a structural join operator can be easily accomplished. Structural joins evaluate structural predicates (XPath axes) that can be partitioned into forward axes, for example, *child*, and reverse axes, for example, *parent*. Each forward axis, except for the *self* axis, has a corresponding reverse axis and vice versa.

The StructuralJoin commutativity rule (SJ-COMM), which is depicted in Figure 6.6, performs the join order exchange in two steps: First, function *invertAxis* returns for a forward axis the corresponding reverse axis and vice versa. Second, the left and the right inputs are swapped.

Join Associativity

In the XQuery world, describing StructuralJoin associativity using a single rule is foiled by the need for early duplicate elimination and for preserving the document order. Though early duplicate elimination is only necessary at the physical level, we can help to preserve this goal even at the logical level. In contrast, the document order must be guaranteed at the logical level.

On the one hand, every plan that forms the root of a query plan has to perform duplicate elimination and sorting, independent of its operator type. On the other hand, plans that have incoming and outgoing edges potentially need to apply duplicate elimination.

Fortunately, not every StructuralJoin plan needs additional duplicate elimination. For example, a structural full join will not create any duplicates, independent of the structural predicate it evaluates. Contrariwise, a structural semi join can produce duplicates.

We can partition structural semi joins into two equivalence classes depending on the necessity of duplicate elimination: (1) semi join operators where only tuples of one incoming tuple sequence can contain duplicates after join evaluation (join operators that evaluate, for example, *child* or *previous-sibling* axes), and (2) semi joins where both incoming tuple sequences can contain

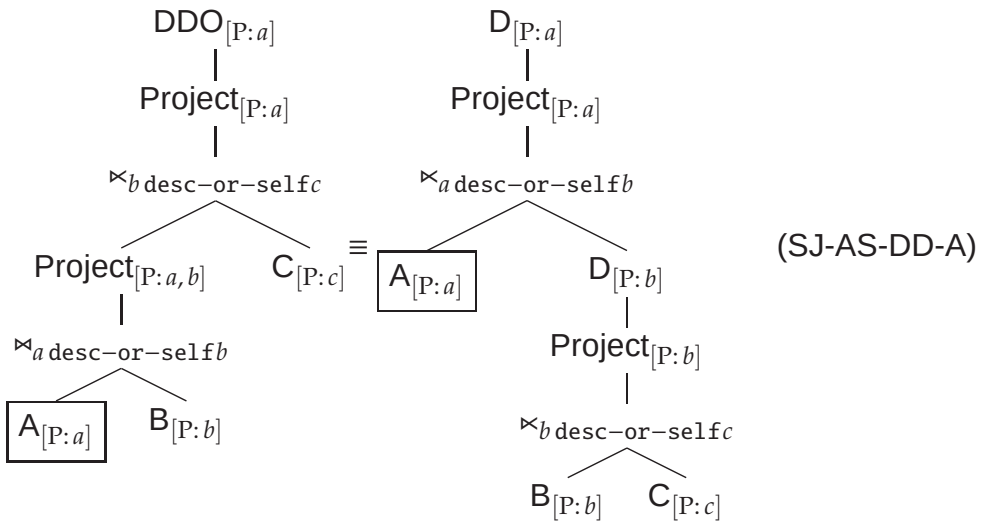


Figure 6.7: A sample StructuralJoin associativity rule

duplicates after join evaluation (join operators that evaluate, for example, *descendant* or *previous* axes).

Let *a* denote the left join partner, *b* denote the right join partner of a structural semi join *p* that evaluates the *child* axis. If *p* only produces *a* tuples satisfying the structural predicate, then duplicate elimination has to be performed, because every node can have multiple child nodes. In contrast, if *p* only delivers *b* tuples as output, then duplicate elimination is not needed, because every node has at most one parent node. If *p* would evaluate a *descendant* axis, then duplicate elimination could become necessary in both cases, because every node can have multiple descendants and multiple ancestors.

To provide a complete set of associativity rules, all combinations of axes have to be considered. Additionally, different outputs need to be taken into account, too. An input plan is called an *output node* if the tuple sequence emitted by it contributes to the query result or is processed in a subsequent plan. Figure 6.7 shows the associativity rule for one output node (plan A) and two adjacent StructuralJoin plans that evaluate the *descendant-or-self* axis[3]. To support a more fine-granular treatment of sorting and duplicate elimination, we replace DDO plans, which only eliminate duplicates, by D. Furthermore, we assume that the output of each join operator is implicitly sorted by the input that is used by a subsequent plan or that contributes to the final result. On the left hand side of Figure 6.7, a structural full join (⋈) is performed

[3] This query graph corresponds to the following XPath expression: a[.//b//c].
To simplify the presentation, we have not fully expanded the "//" abbreviation to *descendant-or-self::node()/*, instead, we only use *desc-or-self* for short.

between tuples of A and B, which needs no additional sorting or duplicate elimination.

The corresponding right-hand side, which contains only left-semi joins ($\ltimes$), requires duplicate elimination and sorting for two reasons: (1) it has only incoming edges and (2) each tuple of the incoming tuple sequence can have multiple descendant *c* nodes. On the right side, a StructuralJoin is performed first between B and C. Because this structural relationship is evaluated using a semi join, we need additional duplicate elimination, because every *b* node can have multiple *c* descendants. The following semi join operator needs duplicate elimination for the same reason, but it needs no additional sorting, due to our implicit sorting assumption.

In essence, all StructuralJoin associativity rules follow the principles discussed in this section[4]. You can look at more join associativity rules in Appendix B on page 221.

6.3.3 Join Fusion

In Section 3.2.1, we argued that applying twig discovery too early in the optimization process, that is, identifying opportunities for using HTJs at the logical level, is harmful. Nevertheless, HTJs are yet another way for evaluating structural patterns. Therefore, our plan generator must take them into consideration. Employing this operator is really useful, if the query engine has to evaluate path expressions where the majority of location steps evaluate the *descendant* axis (Bruno et al., 2002).

Figure 6.8 shows the two StructuralJoin fusion rules for three location steps involving four Access operators and up to two StructuralJoin plans. Using both rules, the plan generator can iteratively replace the StructuralJoin plans by a single TwigJoin. Discussing a generalized version of join fusion, that is, fusion of *n*-way joins is really bulky. For this reason, we restrict our discussion to the example shown in Figure 6.8. However, the general idea should be comprehensible.

Figure 6.8(a) shows the initial rule for join fusion (SJ-FUSION-1), which is always applied bottom-up. Here, two StructuralJoin plans are replaced using a 3-way TwigJoin that receives its inputs from three Access plans A_1, A_2, and A_3[5]. On the right-hand side of Figure 6.8(a), items a_1 and a_2 are joined in the first step, whereas the second steps allows for joining a_1 or a_2 with a_3.

[4] In fact, we already introduced a similar set of SJ associativity rules in Weiner (2007). However, at that time, they were discussed in the context of a very primitive predecessor of today's XQGM.

[5] In Figure 6.8, we simply abbreviated the Access identifiers by A_x for to the sake of readability.

Figure 6.8: The StructuralJoin fusion rule

Which join partner will be actually chosen in the second step depends on the projection specification of the lower StructuralJoin subtree (left-hand side).

Rule SJ-FUSION-1 helped us to derive heterogeneous plans, that is, plans that may consist of StructuralJoin plans and a TwigJoin plan. Figure 6.8(b) shows the second rule (SJ-FUSION-2) that is used as a follow-up to rule SJ-FUSION-1. Using this rewrite, we can further integrate StructuralJoin plans into the TwigJoin. Again, the last location step can contain a join between a_1, a_2, or a_3 with a_4 depending on the structure of the twig pattern.

In Weiner and Härder (2009), we introduced a cost-aware decision criteria that helps the plan generator to check whether applying join fusion leads to a performance gain or loss. We will discuss this criteria in Chapter 8 where we deal with cost-related aspects of plan generation.

6.3.4 TAP Detection

In Section 3.1.3, we introduced TAPs as the most versatile class of access paths that modern native XDBMSs can dispose of. Allowing the plan generator to make use of them motivated the *TAP detection* rules that we will discuss in this section.

Figure 6.9 shows the TAP detection rules (TAP-1 and TAP-2). The first rule replaces a cascade of structural joins with a corresponding IndexAccess plan, if there exists an index for the associated document that is equivalent to the path expression evaluated by the structural join operators. All structural joins must be right-semi joins to preserve the correct output semantics. If a predicate step like *a[b]* shall be supported, which is only possible for the last step of a path expression, a ParentResolution plan must be injected during plan generation, to guarantee the correct output semantics. Moreover, we restrict both TAP detection rules to location steps with "vertical directions", such as *child* or *descendant-or-self*[6], because the evaluation of "horizontal" axes like *following-sibling* cannot be efficiently supported by these access paths.

Rule TAP-2 describes the second step of TAP detection. It permits to integrate further location steps into an IndexAccess plan. Figure 6.9(b) shows the definition of the rule. A location step $(a_n\ \theta_n\ a_{n+1})$ can be integrated into a present IndexAccess plan, if there exists an index definition in the document that matches the following path: $a_1\ \theta_1\ a_2\ \ldots\ a_{n-1}\ \theta_{n-1}\ a_n\ \theta_n\ a_{n+1}$.

Using both rules, we can iteratively replace cascades of StructuralJoin and Access plans by a single IndexAccess plan. Even at the logical level, the benefits of this approach are obvious: (1) We reduce the total number of

[6] We additionally support the *attribute* axis, if it is exclusively evaluated in the last location step.

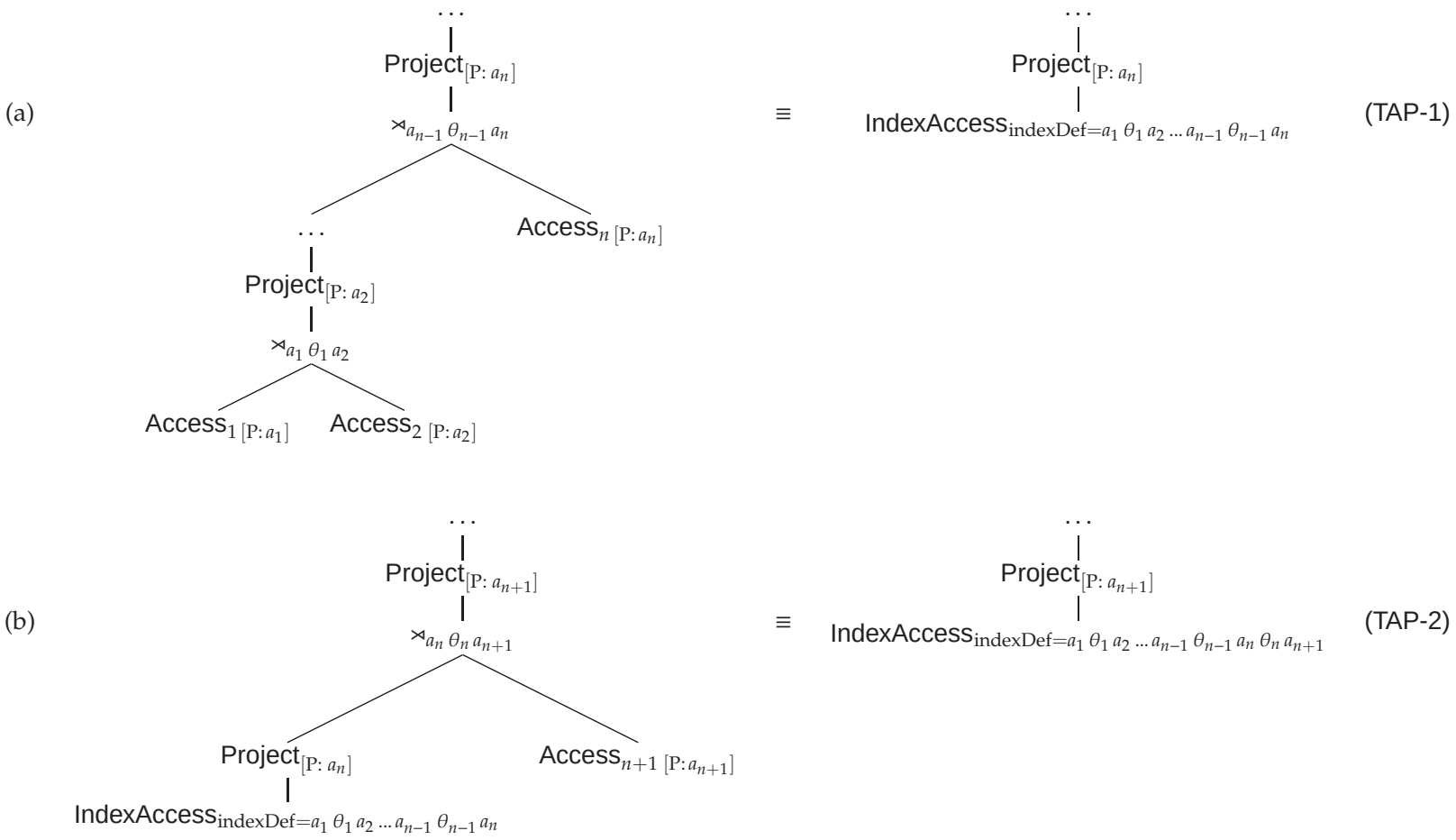


Figure 6.9: TAP detection

Query 6.1 XMark benchmark query Q1

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

operators involved in the query. Hence, we can save valuable CPU costs, because we do not need to evaluate the complete path expression, but can resort to the results of a materialized view on paths, that is, a path index or CAS index and (2) we dramatically reduce random IO accesses, because we only need to scan a single access path instead of evaluating n Access plans for an n-step path expression.

6.4 Example

After the rather sober discussion of the various rules for *implementation variation* and *structural variation*, it is time to have a look at an example. Therefore, let us consider Query 6.1. Figure 6.10 shows the corresponding XQGM instance. In Section 4.1 and Section 5.2, we already discussed a slightly modified version of this query. In the current example, the predicate *[@id = "person0"]* is—in contrast to the previous example—not evaluated in an XQuery *where* clause, but now, expressed as an XPath predicate step. This simple modification has severe consequences on the shape of the XQGM instance, due to additional opportunities for query unnesting. In this case, the final XQGM instance looks much simpler than before. For the rest of this example, let us assume that there exist three indexes for document *auction.xml*:

1. An element index on *people* and *person* nodes
2. A path index on *doc("auction.xml")/site/people*
3. A CAS index on all *id* attribute content nodes $\left(\text{we write } \textit{//@id [String]}\right)$

Figure 6.11 shows the initial plan, right after mapping the XQGM instance on the corresponding plan representation. Subtree ① is equivalent to the path expression *doc("auction.xml")/site/people/person*. First, we could modify the currently assigned implementations of the Access operators for *people* and *person* using rule IS-ACCESS-2. As a consequence, we could derive three new instances of subtree ①: In the first two subtrees, we would alternate the exclusive assignment of one element index scan to *people* and *person* nodes, respectively. In the third case, both Access plans would be implemented

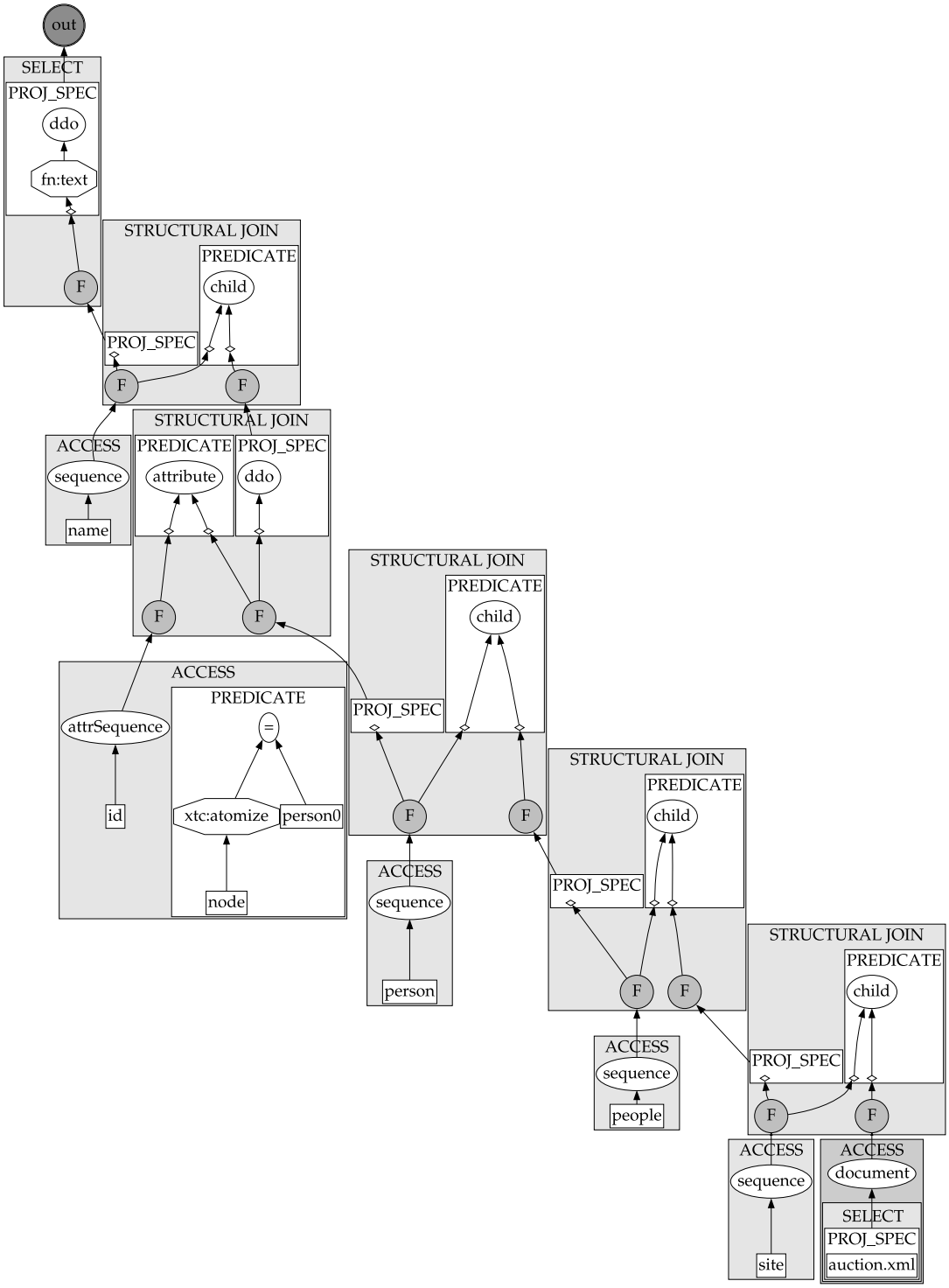


Figure 6.10: XQGM instance of XMark query Q1

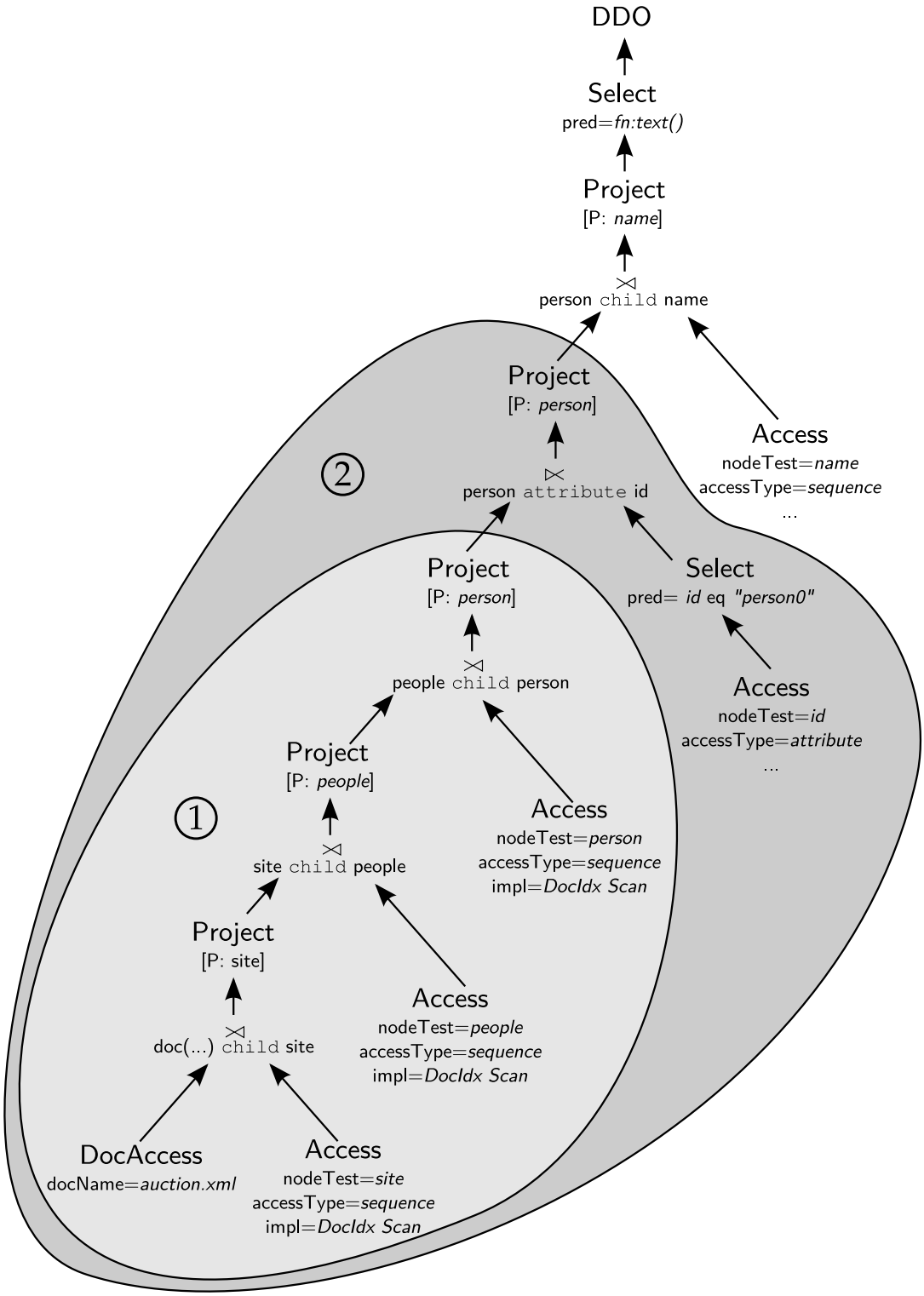
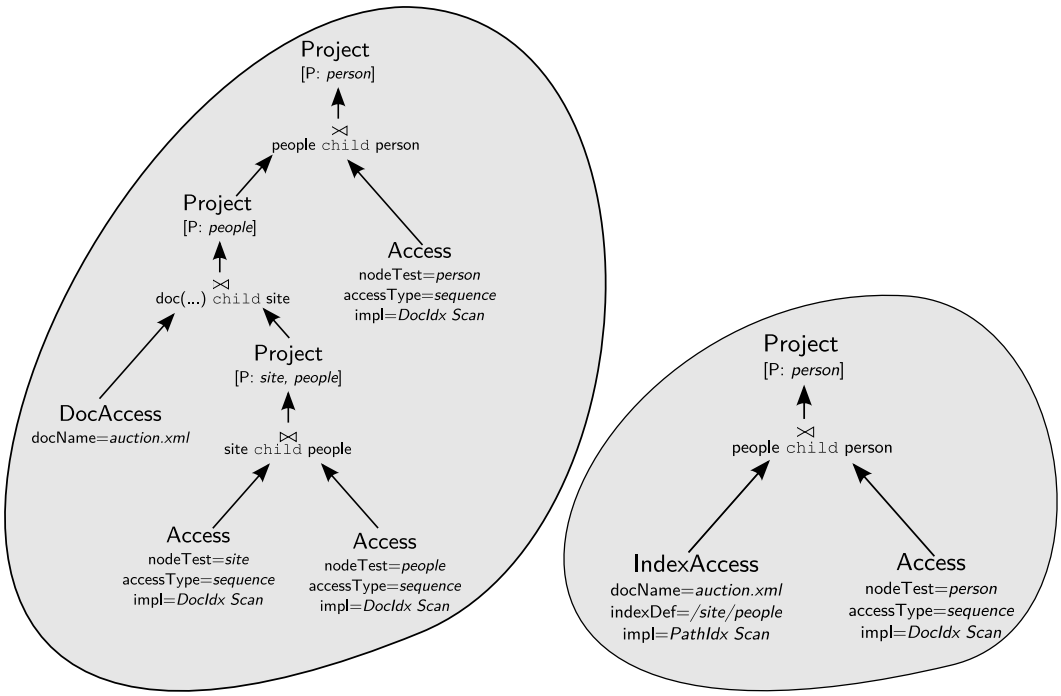


Figure 6.11: Initial plan for XMark Query Q1



(a) Join reordering (b) TAP detection

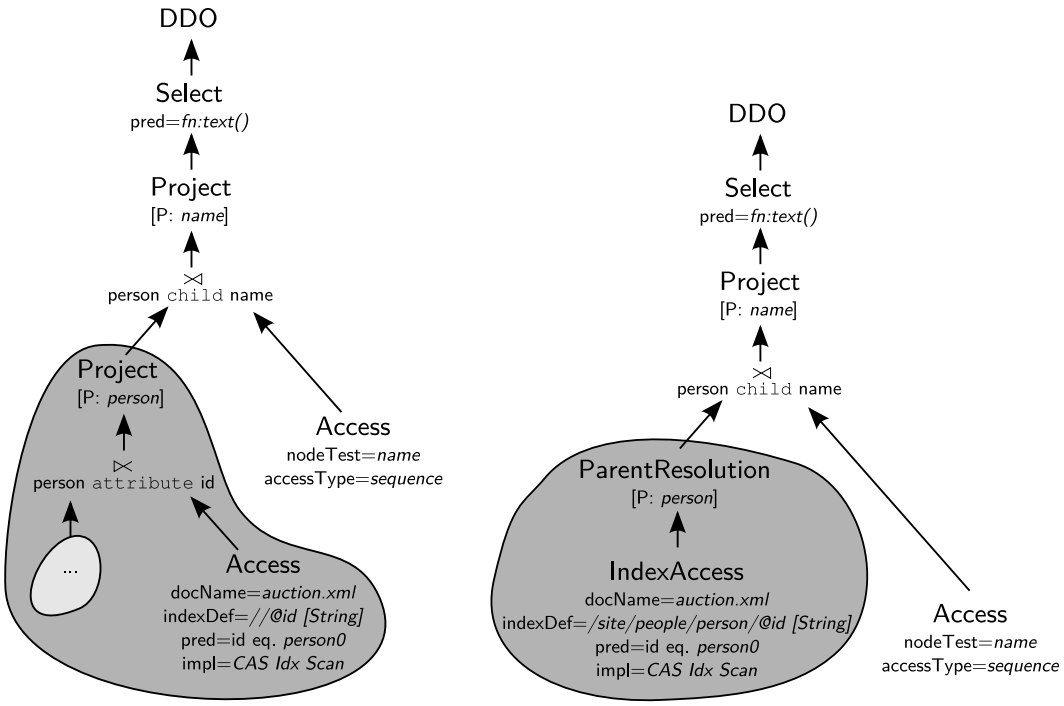
Figure 6.12: Examples of logical query transformations on subtree ①

using element index scans. In total, we get four variants for subtree ①, just by varying the implementations.

For each of these variants, we can perform join reordering. If we reconsider the initial subtree ①, where all Access plans are evaluated using document index scans and apply the join associativity rule SJ-AS-CC-C (Appendix B), we can derive the subtree illustrated in Figure 6.12(a). Now, a structural full join is evaluated between *site* and *people* nodes, because we must retain the *site* nodes that are joined with the virtual document root in the subsequent join.

Applying TAP detection for path *doc("auction.xml")/site/people* on the initial subtree ① results in the plan depicted in Figure 6.12(b). In addition to the two location steps that are replaced by a path index scan, the adjacent Project plan becomes superfluous, because, by definition, the path index returns only *people* nodes. Depending on the clustering mode of the path index, we have to additionally inject a Sort plan above the IndexAccess plan if the path index is clustered by PCRs and not by DeweyIDs.

Before, we only discussed some variations of subtree ①. Thus, let us now focus on subtree ②. Here, we can once again vary the implementation of the Access operators. Using rule IS-ACCESS-4, we can exploit the CAS



(a) Implementation variation (b) TAP detection

Figure 6.13: Logical query transformations on subtree ②

index on *id* attribute nodes. Figure 6.13(a) illustrates subtree ② after applying the transformation. The Access operator can directly evaluate the predicate *@id="person0"*on the CAS index, hence, we can remove the Select operator above it.

In Section 3.1.3, we learned that TAPs are always accompanied by the path synopsis, which is a kind of dynamic schema. Let us assume that a look at the path synopsis tells us that *id* attribute nodes exist only below the path */site/people/person*. Hence, we can reuse the CAS index by applying the TAP detection rule to cover subtree ① and subtree ②. Figure 6.13(b) shows the corresponding plan. To preserve the correct semantics of the plan after rewrite, we inject a ParentResolution plan that returns the duplicate-free sequence of all *person* nodes whose related *id* attribute nodes were returned by the CAS index scan performed by the IndexAccess plan.

Now, you might ask whether Figure 6.13(b) represents the ultimate plan. Actually, the only structural variation that is possible now, is applying the join commutativity rule SJ-COMM on the remaining StructuralJoin plan. Besides that, we can vary the implementations of the StructuralJoin and the remaining

Access plan. Even by neglecting cost information, the plan depicted in Figure 6.13(b) promises to be more efficient compared to the initial plan (Figure 6.11), because the query transformations reduced the number of Access and StructuralJoin plans.

6.5 Summary

In this chapter, we introduced approaches for deriving alternative plans in two dimensions of the search space: implementation variation and structural variation. The first approach offers the plan generator multiple ways for evaluating single plans using different implementations, for example, different access paths for Access plans. The second strategy allows for modifying the structure of plan fragments in such a way, that a more efficient evaluation can be expected.

Section 6.1 motivated the necessity of implementation variation and structural variation for effective query optimization. Next, Section 6.2 discussed several means for implementation variation. Thereafter, Section 6.3 provided opportunities for deriving differently shaped plan fragments. In detail, Section 6.3.1 explained why the optimization of value-based joins is difficult in native XDBMSs. In contrast, we introduced numerous rules for optimizing XPath path expressions, for example, using SJ reordering or TAP detection. As the major part of this chapter was mere theory, Section 6.4 provided a comprehensive example to show the practical application and implications of the various rules.

In the upcoming Chapter 7, we will detail the inference rules that help to estimate the output cardinalities of plans. In combination with the cost model, which will be introduced in Chapter 8, and the logical query transformations presented in this chapter, we can finally implement a cost-based XQuery optimizer.

7 Cardinality Estimation

"Greetings! I am the Count. They call me the Count because I love to count things."

(Count von Count, Sesame Street)

Reliable cardinality estimation is key for effective cost-based query optimization. In this chapter, we enhance previous works on reliable cardinality estimation for XQuery and define an inference rule set that helps to estimate the cardinalities of XQGM expressions taken at runtime[1].

Section 7.1 provides an introduction to the challenges of XQuery cardinality estimation. Thereafter, we introduce in Section 7.2 important definitions such as abstract domain identifiers. In Section 7.3, we define and discuss the inference rule set. Next, Section 7.4 compares the present approach with earlier related works. Finally, Section 7.5 summarizes this chapter and provides a glimpse on subsequent chapters.

7.1 Introduction

In Section 4.2 on page 70, we introduced cost-based query unnesting. This algebraic rewrite requires statistical information for deciding whether unnesting shall be applied or not. Reliable cardinality information is even more important for effective plan generation: Using the logical query transformations presented in Chapter 6, the plan generator can derive a plethora of evaluation orders for XQuery expressions. The grand challenge for cost-based query optimization is now to pick only good plans out of this tremendously large search space. To solve this task, cost factors—whose calculation is based on cardinality estimates for query sub-expressions—are assigned to each alternative for distinguishing good plans from bad ones.

Full cardinality estimation support for XQuery is a strongly neglected research topic in native XDBMSs (Weiner, 2011). Interestingly, the vast majority of cardinality estimation frameworks developed for native XDBMSs only al-

[1] This chapter is partially based on Weiner (2011), whose content is reused in accordance with the publisher's copyright assignment (Gesellschaft für Informatik e. V.).

Query 7.1 Simplified query definition for W3C Use Case "R" query Q13

```
<result> {
 for $uid in distinct-values(doc("bids.xml")//userid),
 $u in doc("users.xml")//user_tuple[userid = $uid]
 let $b := doc("bids.xml")//bid_tuple[userid = $uid]
 return
  <bidder name="{$u/name}" bidCount="{count($b)}"/>
} </result>
```

lows for the estimation of XPath cardinalities (compare Section 7.4). Fairly, XPath is an important fragment of XQuery and necessary for defining search conditions on hierarchical document structures; but restricting cardinality estimation to it is not enough to allow for effective XQuery optimization.

For getting deeper insights into this fundamental problem of XQuery cardinality estimation, let us discuss the estimation capabilities of present approaches with the help of Query 7.1, whose graphical representation is given in Figure 7.1. The query returns for each user that placed a bid the user name and the total number of bids. First, the result of expression *distinct-values(doc("ids.xml")//userid)* is sent from operator ① to operator ②. Select operator ② triggers the subsequent workflow. In operator ③, the first value-based join is evaluated. Operator ④ serves for evaluating the second expression in the *for* clause, which is bound to *$u*. The structural outer join (operator ⑤) returns *(userid, user_tuple)* tuples and does not eliminate *user_tuple* nodes, even if there is no matching join partner. This is necessary, because the correct output semantics for empty sequences in the final *return* clause must be guaranteed. After the evaluation of the value-based join, the qualifying *user_tuple* nodes are sent to operator ⑥, which contribute to the *name* attribute in the query result. Having a look at the query definition shows that there is a second value-based join in the *let* clause (operator ⑨). Operator ⑦ provides the evaluation context (dashed line) for the Access operator below operator ⑧. Finally, the result is passed to operator ② which, in turn, hands it over to operator ⑩.

Current cardinality estimation frameworks cannot provide decent results—even for very simple queries like Query 7.1 (also compare Section 7.4). Let us exemplify the restrictions of present frameworks using our sample query. Most frameworks can only provide estimates for the outputs of the XPath expressions that are bound to *$uid*. This situation is not satisfying, because many optimization problems, for example, selecting an appropriate implementation for value-based joins (Section 6.3.1) or finding the optimal SJ order (Section 6.3.2), are based on fuzzy and coarse-grained heuristics rather than

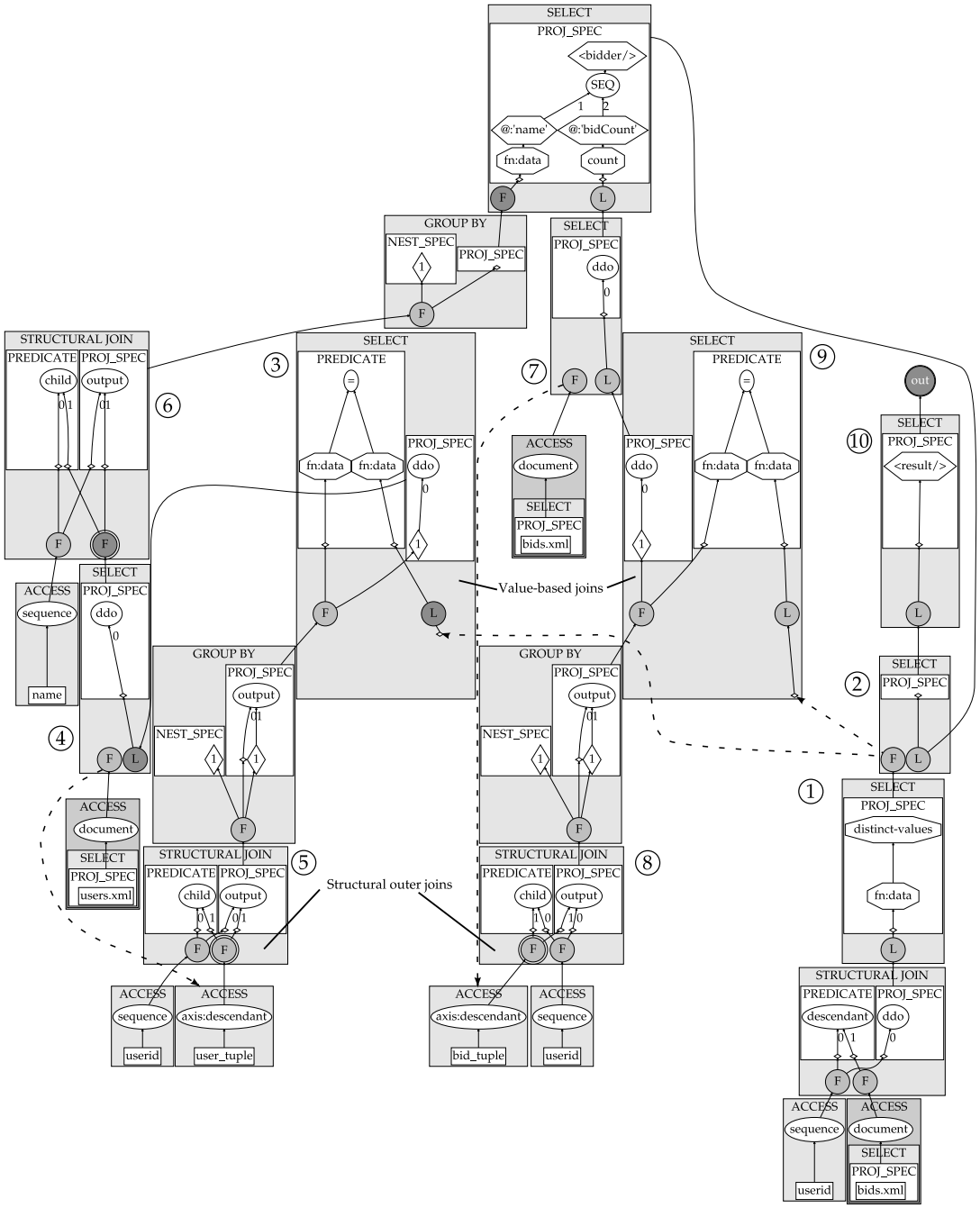


Figure 7.1: Simplified W3C XQuery use case "R" query Q 13 (Weiner, 2011)

costs. For example, if the plan generator can dispose of a hash-based join operator for value-based join evaluation, we can only hope that we chose the smaller input for hashing and the larger input for probing. If our guess shapes up as wrong, we consequently will face a tremendous performance loss.

Moreover, cardinality estimates for the final query result cannot be provided, too. Accordingly, this might have great influence on the selection of an appropriate materialization strategy, for example, in the presence of few final results, late materialization might be preferred over early materialization.

By systematically reviewing recent works (Section 7.4), we state that the concept of *abstract domain identifiers* and the corresponding cardinality inference rules, which were introduced by Teubner et al. (2008), provide an elegant methodology for reliable XQuery cardinality estimation. Moreover, their approach is more advanced than any other XQuery cardinality estimation framework—at least to the best of our knowledge. Unfortunately, their approach cannot be used out-of-the-box for our purposes, because it relies on a completely different algebra and is tailor-made for the relational XQuery processor Pathfinder, which is part of XQuery/MonetDB (see Table 2.1 on page 36). Consequently, we must define a novel set of inference rules that cover all language constructs of XQGM operators such as all variants of binary and n-ary SJ operators (i. e., semi joins, outer joins, and full joins).

7.2 Preliminaries

In this section, we recapitulate important notions such as *abstract domain identifiers* and *active domains* that are necessary to develop the cardinality inference rules (Section 7.2.1). Next, we discuss the generalization of the classical 10% rule, which was introduced in the context of System R as a default selectivity for predicates (Section 7.2.2).

7.2.1 Nomenclature

Teubner et al. (2008) introduced the concept of *abstract domain identifiers*. For your convenience, we repeat the definition and adjust it to our needs in the context of XQGM. As we have already learned in Section 3.1.4, each XQGM operator consumes or emits tuple sequences. Naturally, the same statement holds for plans, because they are simply a different view on XQGM instances. Let $s =< t_1, \ldots, t_n >$ be a tuple sequence where each $t_j = [i_{j1}, \ldots, i_{jm}]$ is a tuple with m items. Obviously, this tuple sequence has a "fixed schema" with m columns (denoted by $c_1, \ldots, c_m$), that is, every tuple has m (probably

	c_1	...	c_m
t_1	i_{11}	...	i_{1m}
⋮	⋮	⋱	⋮
t_n	i_{n1}	...	i_{nm}

Table 7.1: Hypothetical schema

nested) items where each i_{jk} shares a common domain (for an arbitrarily but consistently chosen k). In Table 7.1, we illustrate this hypothetical schema. Abstract domain identifiers (Definition 7.1) allow to estimate the value space of tuple items that are taken by XQGM expressions at runtime.

Definition 7.1 (Abstract domain identifier) An *abstract domain identifier* represents the domain taken by tuple items at runtime, which we denote by Greek letters such as α or β.

Definitions 7.2–7.5 introduce the notions used for the specification of the numerous cardinality inference rules.

Definition 7.2 (Active domain) We call the set of all values taken by tuples $t_1 \ldots, t_n$ in column c_i the *active domain* of c_i and denote it by α_i.

Definition 7.3 (Domain size) The *domain size* of c_i is the total number of distinct values in c_i. We refer to the domain size of c_i by $\|\alpha_i\|$.

Definition 7.4 (Result domain set) We call $\mathrm{dom}(o) = \{c_1^{\alpha_1}, \ldots, c_m^{\alpha_m}\}$ the *result domain set* of the tuple sequence produced by XQGM operator o as output[2], where $c_i^{\alpha_i}$ is the i-th column of the tuple sequence emitted by o with the corresponding active domain α_i.

Definition 7.5 (Inclusion relationship) For the abstract domain identifiers α and β, we define the reflexive and transitive *inclusion relationship* $\beta \sqsubseteq \alpha$ as follows[3]: $\beta \sqsubseteq \alpha \Longleftrightarrow \forall\, b \in \beta : b \in \alpha$.

7.2.2 The Generalized Ten-Percent Rule

The 10% heuristics was introduced by Selinger et al. (1979) in the context of System R. The rationale of this rule is very simple: If there is no statistical

[2] Please note, here we assume that the identifiers $\alpha_1, \ldots, \alpha_m$ have not been used before.

[3] Henceforth, we will denote the assignment of an inferred value by "$=^!$" and an inferred inclusion relationship by "$\sqsubseteq^!$".

information on the value distribution of a column that a query optimizer can dispose of, we assume that a selection predicate (comparison with a constant factor) on this column retains only 10% of its input. Even after more than 30 years, the 10% heuristics serves as default value in modern database systems (compare, e. g., Chaudhuri, 1998; Härder and Rahm, 2001).

Teubner et al. (2008) generalized this rule in such a way that it can be used in their XQuery cardinality inference framework. For your convenience, we briefly discuss the generalization, because we will also make excessive use of it in Section 7.3.

Let us assume that we have a hypothetical schema with m columns. Furthermore, we assume that we have a selection predicate p with selectivity $\sigma(p)$, where $0 \leq \sigma(p) \leq 1$ holds. If we apply p to the hypothetical schema depicted in Table 7.1 and assume an independence between p and column c_i, we are able to estimate the output domain size of c_i. Let $|s|$ be the cardinality of the tuple sequence, that is, the total number of tuples, as illustrated in Table 7.1, and $\|\gamma_1\|$ be the domain size of column c_i in the operator's input tuple sequence (Table 7.2).

On the average, each $a \in \gamma_1$ can be found $|s|/\|\gamma_1\|$ times in the tuple sequence s. Accordingly, the probability that all occurrences of a are not selected by applying p on s is [4]:

$$P_{\notin} = \Big(1 - \sigma(p)\Big)^{|s|/\|\gamma_1\|}$$

Consequently, the probability that at least one representative of a remains in the output tuple sequence is given by: $1 - P_{\notin} = P_{\in}$. We get:

$$P_{\in} = 1 - \Big(1 - \sigma(p)\Big)^{|s|/\|\gamma_1\|}$$

Definition 7.6 is used for the specification of the inference rules: If an XQGM operator or a plan filters its input tuple sequence using a predicate on column c, we estimate the selectivity using XTC's cardinality estimation framework EXsum or use a default constant. For every column that is independent, that is, not affected by the predicate, we use the generalized 10% rule to derive a rough estimation of the output domain size.

Definition 7.6 (Generalized 10% rule) Let $\sigma(p)$ be the selectivity of predicate p that is applied to a tuple sequence s with $|s|$ tuples. Let s have a hypothetical schema as illustrated in Table 7.2. We assume that p is independent of column

[4] If not stated otherwise, we will use $\sigma(p) = 1/10$ in the definition of the inference rules in Section 7.3.

...	c_i	...
	$i_{1\,i}$	
⋮	⋮	⋮
	$i_{1\,i}$	
⋮	⋮	⋮
	$i_{k\,i}$	
⋮	⋮	⋮
	$i_{k\,i}$	

Table 7.2: Generalization of the 10% rule

c_i whose active domain is referred to by γ_1. Hence, we estimate the output domain size $\|\gamma_2\|$ for c_i after applying p on s by:

$$\|\gamma_2\| = \|\gamma_1\| \cdot \Big(1 - \big(1 - \sigma(p)\big)^{|s|/\|\gamma_1\|}\Big).$$

Example

Let us assume that operator o emits the hypothetical schema shown in Table 7.3. Here, we have three active domains α_1, β_1, and γ_1 for columns a, b, and c, respectively. The result domain of o is: $\mathrm{dom}(o) = \{a^{\alpha_1}, b^{\beta_1}, c^{\gamma_1}\}$. Column a has only one distinct value ($\|\alpha_1\| = 1$), column b has three distinct values ($\|\beta_1\| = 3$), and, finally, column c has three distinct values ($\|\gamma_1\| = 3$), too.

If the tuple sequence shown in Table 7.3 serves as input for a Select plan that filters out all items in c whose node value is not equal to "Thomas Mann", an access to a histogram built on column c values might provide the selectivity of the predicate on c, for example, $\sigma(\mathrm{p}) = 1/3$ resulting in an output domain size of $\|\gamma_2\| = 1$. For columns a and b, we use Definition 7.6 to estimate the corresponding output domain sizes $\|\alpha_2\|$ and $\|\beta_2\|$, respectively:

$$\|\alpha_2\| = 1 \cdot \Big(1 - \big(1 - 1/10\big)^{3/1}\Big) = 1 - \big(9/10\big)^3 = 271/1000$$

$$\|\beta_2\| = 3 \cdot \Big(1 - \big(1 - 1/10\big)^{3/3}\Big) = 3 \cdot \big(1 - (9/10)\big) = 3 \cdot 1/10 = 3/10$$

According to the previous calculation, we derive the new output domain sizes for α_2 ($\|\alpha_2\| = 271/1000$) and β_2 ($\|\beta_2\| = 3/10$). To simplify calculations

	a	*b*	*c*
t_1	[node: books; DeweyID: 1.5]	[node: book; DeweyID: 1.5.5]	[node: author; value:"Charles Dickens"; DeweyID: 1.5.5.9.5]
t_2	[node: books; DeweyID: 1.5]	[node: book; DeweyID: 1.5.9]	[node: author; value:"Thomas Mann"; DeweyID: 1.5.9.9.5]
t_3	[node: books; DeweyID: 1.5]	[node: book; DeweyID: 1.5.13]	[node: author; value:"Charles Dickens"; DeweyID: 1.5.13.9.5]

Table 7.3: Abstract domains in action

in subsequent estimation steps, floating point values will always be rounded up to the closest integer value. In our example, if the estimates are rounded up, the estimated output domain sizes are exact. Fairly, this is not always the case, because the generalized 10% rule provides only a rough estimation. Nevertheless, we will see in Chapter 12 that the estimates are precise enough to effectively support the cost-based query optimizer.

7.3 Cardinality Inference

Now, we are well-prepared to introduce our cardinality inference framework, which is at the heart of XTC's cost estimator. In this section, we use the following notation to describe the inference rules [5]:

$$\frac{\text{necessary condition}}{\text{inference}}$$

where *necessary condition* describes a predicate that must be satisfied and that is necessary for applying the inference rule. Accordingly, *inference* defines which cardinality estimate is assigned to the tuple sequence of output plan o and determines the structure and estimates for the corresponding result domain set, which is denoted as $\text{dom}(o)$.

7.3.1 Access Operators

Inference Rule Set 7.1 shows the inference rules for various Access plans. For cardinality inference, which is dataflow oriented, we always start at the leave nodes of plans. Hence, Access plans serve as entry points for cardinality estimation, where we can assign exact numbers derived from the statistical information stored in the database catalog. Based on this information, subsequent inference steps approximate the result sizes of intermediate plans. First,

[5] As our rule set is complementary to the work of Teubner et al. (2008), we borrow their notation.

$$\frac{}{\mathrm{dom}(\mathsf{DocAccess}) = \left\{c^{\alpha} \wedge \|\alpha\| =^{!} 1\right\}} \quad \text{(CARD-DOC-ACCESS)}$$

$$\frac{}{\mathrm{dom}(\mathsf{Access}_{\text{nodeTest}=e;\ \text{accessType}=\textit{sequence};\ \text{pred}=p}) = \left\{c^{\alpha} \wedge \|\alpha\| =^{!} |e| \cdot \sigma(p)\right\}} \quad \text{(CARD-ACCESS)}$$

$$\frac{b^{\beta} \in \mathrm{dom}(\square) \wedge \square \text{ provides evaluation context}}{\mathrm{dom}(\mathsf{Access}_{\text{nodeTest}=e;\ \text{accessType}=\textit{node};\ \text{axis}=\theta;\ \text{pred}=p}) = \left\{a^{\alpha} \wedge \|\alpha\| =^{!} |e| \cdot \sigma(b\,\theta\,e)\right\}} \quad \text{(CARD-CTX-ACCESS)}$$

Inference Rule Set 7.1: Inference rules for Access operators

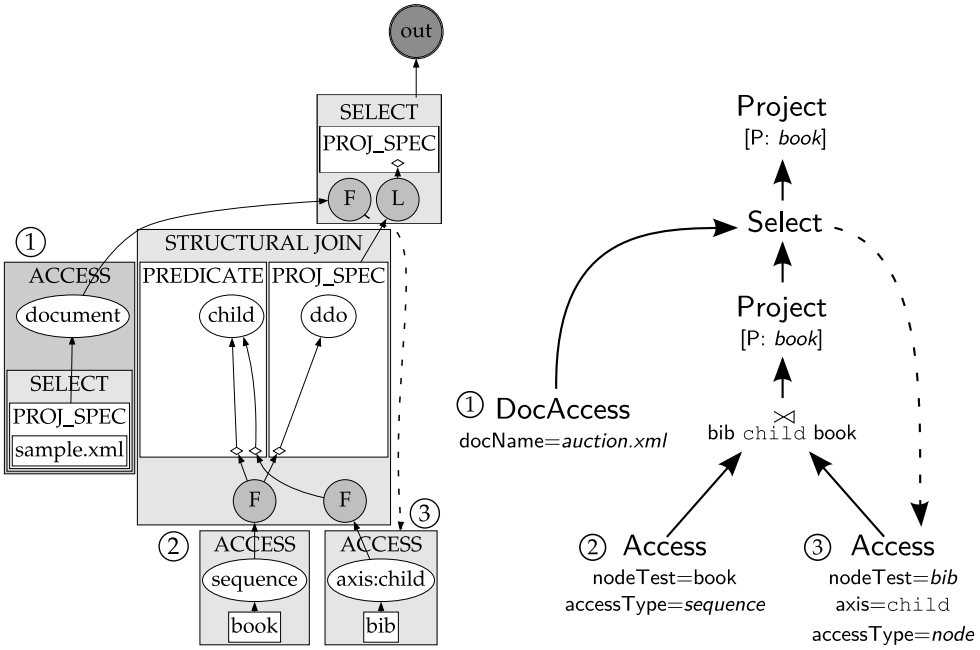
rule CARD-DOC-ACCESS is the most primitive inference rule and serves for deriving the input cardinality of the DocAccess plan that provides the initial evaluation context (i. e., document root) for query evaluation.

We use rule CARD-ACCESS for Access plans providing sequences of element or attribute nodes with name e (node test on e) for deriving their result domain sets. Each plan may evaluate an optional predicate p. We estimate an Access plan's cardinality by the total number of element or attribute names that share the name e. Selectivity estimation for p can be done using histograms or by simply using System R's famous 10% heuristics (Selinger et al., 1979). Finally, for approximating the cardinality of Access plans whose output depends on an evaluation context, we employ rule CARD-CTX-ACCESS. Such Access plans are used in subexpressions that cannot be unnested using the rules defined by Mathis (2009). We can use the active domain β of the context-providing operator as a starting point for cardinality inference. In expression $\sigma(b\,\theta\,e)$, b is the current context item, θ is the corresponding XPath axis and e is the tuple stream issued by the current Access operator[6].

Example

To exemplify the application of the inference rules for Access plans defined in Inference Rule Set 7.1, let us consider the simple query *doc("auction.xml")/bib/book*. In Figure 7.2(a) and Figure 7.2(b), we can see the corresponding XQGM instance and a possible plan, respectively. Let us focus on the plan depicted in Figure 7.2(b): For cardinality estimation of operator ①, we use the CARD-DOC-ACCESS rule. Operator ② retrieves the sequence

[6]Please note, in $\sigma(b\,\theta\,e)$, selectivity estimation is actually based on β and not on $|b|$.



(a) XQGM instance (b) Corresponding plan

Figure 7.2: Cardinality inference for Access plans

of all *book* nodes. In this situation, rule CARD-ACCESS is used to estimate the corresponding result domain set. Finally, operator ③ receives the current evaluation context from the Select plan above and uses it for the evaluation of the axis step ($\theta = child$) between *bib* nodes and the current evaluation context. Instead of estimating the cardinality of each individual step, we estimate the cardinalities and the result domain set of all steps at once using rule CARD-CTX-ACCESS. This is possible, because we already know the active domain of the input bound to the *for*-quantified tuple variable of the Select operator in Figure 7.2(a).

7.3.2 Cardinality Estimation for Structural Joins

In this section, we discuss the inference rules for the numerous variants of StructuralJoin plans. We use the well-known symbols for denoting the respective join types: $\ltimes_p$ (structural left-semi join), $\rtimes_p$ (structural right-semi join), $\bowtie_p$ (structural full join), $⟕_p$ (structural left-outer join), and $⟖_p$ (structural right-outer join). The structural join evaluates a structural predicate p

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \;\wedge\; b_j^{\beta_j} \in \mathrm{dom}(q_2)}{\begin{array}{c}|q_1 \ltimes_{a_i \theta b_j} q_2| = \|a_i\| \cdot \sigma\Big(a_i[\theta\, b_j]\Big) \;\wedge\; \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \Big\{c_2^{\gamma_2} \Big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge \\ c_1^{\gamma_1} \in \mathrm{dom}(q_1) \backslash \{a_i^{\alpha_i}\} \;\wedge\; \|\gamma_2\| =^! \|\gamma_1\| \cdot \Big[1 - (1 - 1/10)^{|q_1|/\|\gamma_1\|}\Big]\Big\} \cup \\ \Big\{c^{\gamma} \Big| \gamma \sqsubseteq^! \alpha_i \;\wedge\; \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\Big(a_i[\theta\, b_j]\Big)\Big\}\end{array}} \quad \text{(CARD-SJ-1)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \;\wedge\; b_j^{\beta_j} \in \mathrm{dom}(q_2)}{\begin{array}{c}|q_1 \ltimes_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i\, \theta\, b_j) \;\wedge\; \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \Big\{c_2^{\gamma_2} \Big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge \\ c_1^{\gamma_1} \in \mathrm{dom}(q_2) \backslash \{b_j^{\beta_j}\} \;\wedge\; \|\gamma_2\| =^! \|\gamma_1\| \cdot \Big[1 - (1 - 1/10)^{|q_2|/\|\gamma_1\|}\Big]\Big\} \cup \\ \Big\{c^{\gamma} \Big| \gamma \sqsubseteq^! \beta_j \;\wedge\; \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i\, \theta\, b_j)\Big\}\end{array}} \quad \text{(CARD-SJ-2)}$$

Inference Rule Set 7.2: Inference rules for structural semi joins

described as follows: $a_i\, \theta\, b_j$, where a_i is an item of tuple sequence q_1, b_j is an item of tuple sequence q_2, and θ is an XPath axis, for example, *descendant*.

At first sight, the specification of the inference rules seems to be cumbersome, however, their rationale is very simple: For approximating the result size of active domains affected by structural predicate θ, we use two data structures: For path expressions without path predicates, we employ the PS (see Section 3.1.3 on page 45) to derive accurate cardinalities. In contrast, if the path expression involves path predicates, we get the estimated cardinality using XTC's XPath cardinality estimation framework called *EXsum* (Aguiar Moraes Filho, 2010).

Expression $\sigma\Big(a_i[\theta\, b_j]\Big)$ returns the selectivity of a_i items connected to b_j items via the θ axis, that is, the percentage of a_i nodes satisfying the structural predicate. On the other hand, $\sigma(a_i\, \theta\, b_j)$ returns the selectivity of b_j items. For all remaining items, we use Definition 7.6 to estimate the new cardinalities of active domains that are not directly affected by the structural predicate, that is, non-join items: We use Definition 7.6 and assume $\sigma(p) = 1/10$ resulting in $\|\gamma_2\| =^! \|\gamma_1\| \cdot \Big[1 - (1 - 1/10)^{|q|/\|\gamma_1\|}\Big]$, where $|q|$ is the cardinality of input plan q, γ_1 is the active domain cardinality of a column contained in the input tuple sequence of q, and $\|\gamma_2\|$ is the inferred cardinality for the corresponding active domain in the output tuple sequence.

Rules CARD-SJ-1 and CARD-SJ-2 (Inference Rule Set 7.2) depict the inference rules for structural left-semi joins and right-semi joins, respectively. Here, the

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2) \wedge a_i \theta b_j \text{ is location step}}{\begin{array}{c}|q_1 \bowtie_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i \,\theta\, b_j) \wedge \mathrm{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{c_2^{\gamma_2} \middle| \right. \\ \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \left\{a_i^{\alpha_i}, b_j^{\beta_j}\right\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - \right. \\ \left. \left. (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|} \right] \right\} \cup \left\{c^\gamma \middle| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\left(a_i[\theta b_j]\right) \right\} \cup \\ \left\{c^\gamma \middle| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma_j\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}\end{array}} \quad \text{(CARD-SJ-3)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2) \wedge a_i \theta b_j \text{ is predicate step}}{\begin{array}{c}|q_1 \bowtie_{a_i \theta b_j} q_2| = \|a_i\| \cdot \sigma\left(a_i[\theta b_j]\right) \wedge \mathrm{dom}(q_1 \bowtie_{a_i \theta b_j} q_2) = \left\{c_2^{\gamma_2} \middle| \right. \\ \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \left\{a_i^{\alpha_i}, b_j^{\beta_j}\right\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - \right. \\ \left. \left. (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|} \right] \right\} \cup \left\{c^\gamma \middle| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\left(a_i[\theta b_j]\right) \right\} \cup \\ \left\{c^\gamma \middle| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}\end{array}} \quad \text{(CARD-SJ-4)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{\begin{array}{c}|q_1 \ltimes_{a_i \theta b_j} q_2| = \|\beta_j\| \wedge \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \left\{b_j^{\beta_j}\right\} \cup \\ \left\{c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \setminus \left\{a_i^{\alpha_i}, b_j^{\beta_j}\right\} \wedge \right. \\ \left. \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\right] \right\} \cup \\ \left\{c^\gamma \middle| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\left(a_i[\theta b_j]\right) \right\}\end{array}} \quad \text{(CARD-SJ-5)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{\begin{array}{c}|q_1 \rtimes_{a_i \theta b_j} q_2| = \|\alpha_i\| \wedge \mathrm{dom}(q_1 \rtimes_{a_i \theta b_j} q_2) = \left\{a_i^{\alpha_i}\right\} \cup \\ \left\{c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \setminus \left\{a_i^{\alpha_i}, b_j^{\beta_j}\right\} \wedge \right. \\ \left. \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\right] \right\} \cup \\ \left\{c^\gamma \middle| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \right\}\end{array}} \quad \text{(CARD-SJ-6)}$$

Inference Rule Set 7.3: Inference rules for full joins and outer joins

q_1

	a_1	a_2	a_3
t_1	i_1	j_1	k_1
t_1	i_1	j_1	k_1
t_3	i_3	j_3	k_2
t_4	i_4	j_4	k_2

(a) Left input

q_2

	b_1	b_2	b_3	b_4
t_1	l_1	m_1	n_1	o_1
t_2	l_3	m_2	n_1	o_2
t_3	l_1	m_3	n_1	o_2

(b) Right input

$q_1 \ltimes_{a_1 \theta b_1} q_2$

	c_1	c_2	c_3
t_1	i_1	j_1	k_1
t_2	i_1	j_1	k_1
t_3	i_3	j_3	k_2

(c) Join result

Table 7.4: Output tuple sequences of operators q_1 and q_2

result domain set is equal to the domain set of the left or right input operator, respectively. The cardinalities of the active domains' join items are estimated using the path synopsis or EXsum and all remaining cardinalities of the active domains in the output domain set are approximated using the generalized 10% rule.

Inference Rule Set 7.3 shows the inference rules for structural full joins. Rule CARD-SJ-3 and rule CARD-SJ-4 show the corresponding definitions, which distinguish between the evaluation of location steps and predicate steps. Both rules estimate the output cardinality of the join operator and the active domains of join items a_i and b_j using the path synopsis. Once again, the generalized 10% rule helps to approximate the active domains for items that are independent of the join predicate.

In Figure 7.1 on page 109, several structural outer joins are used (e. g., operator ⑤). Inference rules CARD-SJ-5 and CARD-SJ-6 allow for cardinality inference of structural left-outer joins and structural right-outer joins, respectively[7]. The active domain of the join item whose tuple sequence contributes to the output's outer part remains unchanged, and the cardinality of the other join item is adjusted according to EXsum's estimation. For all remaining active domains, the generalized 10% rule is applied.

Example

Let us have a look at the cardinality inference rule for structural left-semi joins (CARD-SJ-1). Therefore, let us assume that input operator q_1 emits $\text{dom}(q_1) = \left\{a_1^{\alpha_1}, a_2^{\alpha_2}, a_3^{\alpha_3}\right\}$ and input operator q_2 provides $\text{dom}(q_2) = \left\{b_1^{\beta_1}, b_2^{\beta_2}, b_3^{\beta_3}, b_4^{\beta_4}\right\}$. Moreover, we assume that a_1 and a_2 are join attributes where i_p items from q_1 and l_p items from q_2 satisfy the structural predicate θ, for example, $(i_1 \, \theta \, l_1) = \mathit{true}$. Table 7.4(a) shows the output tuple sequences of

[7] Please note, we do not use structural full-outer joins in XQGM.

operator q_1 and 7.4(b) illustrates the output of operator q_2. Additionally, 7.4(c) shows the join result after duplicate elimination.

Consequently, for $\mathrm{dom}(q_1)$, we get: $\|\alpha_1\| = 3$, $\|\alpha_2\| = 3$, and $\|\alpha_3\| = 2$ as well as for $\mathrm{dom}(q_2)$: $\|\beta_1\| = 2$, $\|\beta_2\| = 3$, $\|\beta_3\| = 1$, and $\|\beta_4\| = 2$.

Now, let us assume that we are evaluating a structural left-semi join between q_1 and q_2. Hence, we have to use rule CARD-SJ-1 for cardinality inference. First, we estimate the result cardinality:

$$|q_1 \ltimes_{a_1\,\theta\,b_1} q_2| = \|\alpha_1\| \cdot \sigma\Big(a_1[\theta\, b_1]\Big) = 3 \cdot 2/3 = 2$$

Let us assume that the join result has the following result domain set:

$$\mathrm{dom}\Big(q_1 \ltimes_{a_1\,\theta\,b_1} q_2\Big) = \Big\{c_1^{\gamma_1}, c_2^{\gamma_2}, c_3^{\gamma_3}\Big\}$$

The active domain size of c_1, which equals to the former join column a_1, is estimated as follows:

$$\|\gamma_1\| = \|\alpha_1\| \cdot \sigma\Big(a_1[\theta\, b_1]\Big) = 3 \cdot 2/3 = 2$$

For the remaining columns c_2 and c_3, we use the generalized 10% heuristics:

$$\|\gamma_2\| = \|\alpha_2\| \cdot \Big(1 - (9/10)^{4/3}\Big) = 3 \cdot \Big(1 - \sqrt[3]{6561/10{,}000}\Big) \approx 0.393$$
$$\|\gamma_3\| = \|\alpha_3\| \cdot \Big(1 - (9/10)^{4/2}\Big) = 2 \cdot \Big(1 - (9/10)^{4/2}\Big) = 2 \cdot \Big(1 - 81/100\Big) = 38/100$$

By rounding $\|\gamma_2\|$ and $\|\gamma_3\|$ up to the next integer, we get 1 for both active domains. Figure 7.4(c) shows the join result after duplicate elimination. In this situation, the inferred domain size $\|\gamma_1\|$ is estimated correctly, whereas for $\|\gamma_2\|$ and $\|\gamma_3\|$ the sizes are underestimated. Moreover, in our example, the output cardinality is expected to be 2 instead of 3.

7.3.3 Inference Rules for Grouping, Unnesting, and Miscellaneous Operators

Grouping and unnesting are important operations in XQGM. They allow to create nested and flat tuples, respectively. Document order and duplicate elimination are important features of XQuery. Moreover, projection helps to minimize memory consumption by cutting off items that are irrelevant for subsequent operations. Finally, providing inputs for multiple consumers by a single Split plan helps to keeps QEPs manageable. In this section, we discuss the inference rules for the diverse plan types mentioned before. Inference Rule Set 7.4 presents the corresponding definitions.

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q)}{|\mathsf{GroupBy}_i(q)| = \|\alpha_i\|} \quad \text{(CARD-GROUP-BY)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q)}{|\mathsf{Unnest}_i(q)| = \mathrm{flatCard}_i(q)} \quad \text{(CARD-UNNEST)}$$

$$\frac{\square \in \{\mathsf{Split}, \mathsf{Project}\}}{\mathrm{dom}(\square) = \left\{ a_i^{\alpha_i} \,\middle|\, a_i^{\alpha_i} \in \mathrm{dom}(q) \wedge a_i \in \square_{\mathrm{projSpec}} \right\}} \quad \text{(CARD-MISC-1)}$$

$$\frac{\square \in \{\mathsf{Sort}, \mathsf{DDO}\}}{|\square(q)| = |q| \cdot 2/3} \quad \text{(CARD-MISC-2)}$$

Inference Rule Set 7.4: Grouping, unnesting, and miscellaneous operators

Rule CARD-GROUP-BY shows the inference for tuple grouping. Grouping is always performed with respect to a specific item position i. The result domain set of GroupBy is equal to the input domain set provided by operator q. Therefore, it is not necessary to derive cardinality estimates for individual items. But, instead, we must calculate the new output cardinality, which can be simply achieved by using the domain size of the grouping item i, which we denote by $\|\alpha_i\|$.

As unnesting is the inverse operation of grouping, we employ a similar approach for estimating the output cardinality of Unnest plans (rule CARD-UNNEST). Here, we use the function $\mathrm{flatCard}_i(q)$ that "flattens" the nested input tuple sequence provided by operator q with respect to item position i. Again, no modifications of the result domain set is performed.

The Split operator sends its input to multiple consumers and the Project operator serves as classical projection operator. Rule CARD-MISC-1 simply derives the result domain set by taking only those active domains into account that are referred to in the projection specification. In this case, the cardinality estimates remains unmodified.

Conventionally, every XQGM operator returns tuple sequences sorted in document order and tries to reduce duplicates to a minimum. If duplicates cannot be avoided, for example, if a full join using the *descendant* axis is performed, additional duplicate elimination might become necessary (also see Section 6.3.2). Sort plans and DDO plans retain a certain sort order or eliminate duplicates. Inference rule CARD-MISC-2 defines the corresponding rule for output cardinality estimation. Normally, we expect that most tuple

$$\frac{}{|\mathsf{Merge}(q_1,\ldots,q_n)| = \prod_{i=1}^{n} |q_i| \cdot 1/10 \wedge \mathrm{dom}(\mathsf{Merge}(q_1,\ldots,q_n)) \supseteq \left\{ c_2^{\gamma_2} \,\middle|\, \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^{n} \mathrm{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{(\prod_{i=1}^{n} |q_i|)/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-MERGE-1)}$$

$$\frac{q_i \text{ delivers outer sequence}}{|\mathsf{Merge}(q_1,\ldots,q_n)| = |q_i| \wedge \mathrm{dom}(\mathsf{Merge}(q_1,\ldots,q_n)) \supseteq \left\{ c_2^{\gamma_2} \,\middle|\, \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^{n} \mathrm{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{|q_i|/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-MERGE-2)}$$

Inference Rule Set 7.5: Inference rules for Merge plans

sequences are almost duplicate free and sorted, hence, we assume that two thirds of their input tuples will "survive" sorting or duplicate elimination.

7.3.4 Inference Rules for Merge

Rules CARD-MERGE-1 and CARD-MERGE-2, which are part of Inference Rule Set 7.5, show the inference rules for Merge plans. A Merge plan contains only *for*-quantified tuple variables and calculates the Cartesian product on its input tuple streams. Each plan contains a so-called merge specification (property *mergeSpec*) that describes a complex selection predicate on the Cartesian product. The predicate selects all tuples that have equal values for given positions in the tuple sequence. For the sake of simplicity, we use the 10% rule to determine the output cardinality.

A special case is handled by rule CARD-MERGE-2: If an outer tuple variable appears in the merge specification of Merge, the outer semantics—which is well-known from outer joins—is used, that is, for every tuple sequence, where a match does not exist, the tuple still appears in the Cartesian product and all non-matching items are replaced by empty sequences (Mathis, 2009). Here, the output cardinality is simply estimated using the cardinality of the tuple sequence associated with the operator connected to the outer tuple variable.

$$\mathsf{Select}_p(q_1,\ldots,q_n)\ s \wedge Q_{\text{for, let}} = \Big\{ q \,\Big|\, q \in \{q_1\ \ldots,q_n\} \wedge \Big(\text{isForQuantified}\Big(q, \mathsf{Select}_p(q_1,\ldots,q_n)\Big) \vee \text{isLetQuantified}\Big(q, \mathsf{Select}_p(q_1,\ldots,q_n)\Big)\Big)\Big\} \wedge$$

$$Q_{\text{exists}} = \Big\{ q \,\Big|\, q \in \{q_1\ \ldots,q_n\} \wedge \text{isExistsQuantified}\Big(q, \mathsf{Select}_p(q_1,\ldots,q_n)\Big)\Big\}$$

$$|\mathsf{Select}_p(q_1,\ldots,q_n)| = \prod_{q \in Q_{\text{for, let}} \wedge q \in s_{\text{projSpec}}} |q| \cdot 1/10 \wedge$$

$$\text{dom}(\mathsf{Select}(q_1,\ldots,q_n)) = Q'_{\text{for, let}} \cup Q'_{\text{exists}} \wedge$$

$$Q'_{\text{for, let}} = \Big\{ c_2^{\gamma_2} \Big| \gamma_2 \sqsubseteq \gamma_{1j} \wedge c_{1j}^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in Q_{\text{for, let}} \wedge \|\gamma_2\| =^{!} \|\gamma_{1j}\| \cdot [1-(1-1/10)^{|q_j/\|\gamma_{1j}\|}]\Big\} \wedge$$

$$Q'_{\text{exists}} = \Big\{ c_2^{\gamma_2} \Big| \gamma_2 \sqsubseteq \gamma_{1j} \wedge c_{1j}^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in Q_{\text{exists}} \wedge \|\gamma_2\| =^{!} \|\gamma_{1j}\| \cdot [1-(1-1/2)^{|q_j/\|\gamma_{1j}\|}]\Big\} \qquad \text{(CARD-SELECT)}$$

Inference Rule Set 7.6: Inference rules for Select plans

7.3.5 Inference Rules for Select

In XQGM, the Select operator serves for calculating the Cartesian product. In contrast to the Merge operator, the Select operator can contain tuple variables with mixed quantifiers: *for*, *let*, or *exists*. The Select operator is the most versatile operator in XQGM, as it allows to express, amongst others, value-based joins and simple selection predicates as well as triggering XQuery *for-let* bindings.

Rule CARD-SELECT, which is defined in Inference Rule Set 7.6, describes the cardinality inference for Select plans[8]. The different *for*-quantified tuple variables "drive" the output generation process, that is, they determine the actual output cardinality. On the other hand, the tuple sequences bound to *let*-quantified tuple variables are "nested" into the results generated by *for*-quantified tuple variables, whereas *exists*-quantified tuple variables simply allow for existence tests.

Having a look at the definition of rule CARD-SELECT shows that it uses a similar approach for output cardinality estimation as described before for

[8] We assume that a Select plan always evaluates a predicate p. Fairly, there are also Select plans that do not evaluate predicates, but evaluate only complex Cartesian products. For these plans, we apply the same rule, but, instead, use a selectivity factor of 100%. For the sake of brevity, we do not give a formal definition of this trivial case.

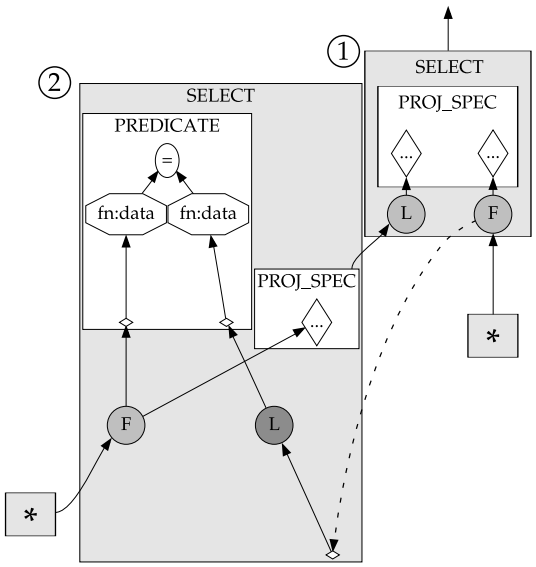


Figure 7.3: Value-based join

Merge plans. In the necessary condition, we assume two sets: $Q_{\text{for, let}}$ and Q_{exists} that contain all input operators of Select that are connected to a *for*-quantified or *let*-quantified tuple variable as well as an *exists*-quantified tuple variable, respectively. For estimating the output cardinality of Select plans, we consider exactly those operators that are contained in $Q_{\text{for, let}}$ and that are also referred to in the projection specification.

For the estimation of the output domain sizes of inputs bound to *for*-quantified and *let*-quantified tuple variables, we use the the generalized 10% rule with a default selectivity of $1/10$, whereas for approximating the output domain sizes of *exists*-quantified tuple variables, we employ a default factor of $1/2$.

Example

As mentioned before, Select operators play an important role in XQGM. Therefore, we will have a look at some examples how cardinality inference will work for these operators. Applying rule CARD-SELECT to Select plans that contain only a single *for*-quantified or *let*-quantified tuple variable is trivial. Therefore, we will only look at two more interesting cases: (1) evaluation of value-based joins (2) triggering of *for-let*-bindings.

Section 7.1 on page 109 already discussed a query containing two value-based joins, which are illustrated as operator ③ and operator ⑨ in Figure 7.1.

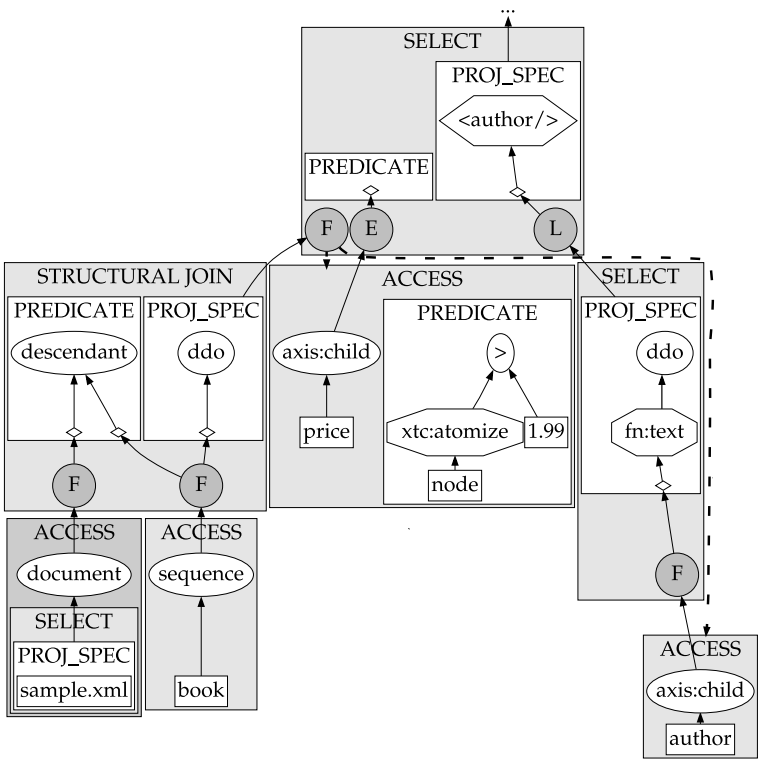


Figure 7.4: Complex selections

The general shape of a value-based join is depicted in Figure 7.3. Even though only operator ② evaluates the join predicate, it always needs another Select operator that triggers the evaluation process (here, it is operator ①). Operator ② projects out only those tuples that are provided by the *for*-quantified tuple variable and that satisfy the predicate. The tuples received via the *let*-quantified tuple variable are only used for predicate checking and do not participate in the result. Operator ① triggers the join evaluation by iterating over its *for*-quantified tuple variable's input tuple sequence and sends each item to operator ②. The join result, which is bound to the *let*-quantified tuple variable, can be part of the result sent to subsequent operators if it is contained in operator ①'s projection specification. Nevertheless, it will always be nested below the current context item that served for predicate evaluation.

For the second example, let us consider the XQGM instance depicted in Figure 7.4 on page 125. Here, the top-most Select operator contains a *for*-quantified, a *let*-quantified, and an *exists*-quantified tuple variable. In this case, the *let*-quantified tuple variable is connected to the projection specification. Hence, in this situation, it is the only tuple variable that must be considered

for cardinality inference. Though the *for*-quantified tuple variable does not directly influence the output cardinality, it triggers the emission of output nodes—one sequence (containing one or more *author* nodes) per *book* element.

$$\frac{|q_1| = |q_2| = \ldots = |q_n|}{|\cup_{i=1}^{n} q_i| = \sum_{i=1}^{n} |q_i| \wedge \mathrm{dom}(\cup_{i=1}^{n} q_i) = \cup_{k=1}^{|\mathrm{dom}(q_1)|} \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^{!} \gamma_1 \wedge c_{1k}^{\gamma_1} \in \mathrm{dom}(q_1) \wedge \|\gamma_2\| =^{!} \sum_{i=1}^{n} \|\gamma_{ik}\| \right\}} \quad \text{(CARD-UNION)}$$

$$\frac{a_1^{\alpha_1} \in \mathrm{dom}(q_1) \wedge \ldots \wedge a_n^{\alpha_n} \in \mathrm{dom}(q_n) \wedge |q_k| = \min\{|q_1|, \ldots, |q_n|\}}{|\cap_{i=1}^{n} q_i| = |q_k| \cdot 2/3 \wedge \mathrm{dom}(\cap_{i=1}^{n} q_i) = \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^{!} \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_k) \wedge \|\gamma_2\| =^{!} \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{|q_k|/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-INTERSECT)}$$

$$\frac{}{|q_1 \setminus q_2| = |q_1| \cdot 1/10 \wedge \mathrm{dom}(q_1 \setminus q_2) = \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^{!} \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \wedge c_1^{\gamma_1} \notin \mathrm{dom}(q_2) \wedge |\gamma_2\| =^{!} \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{|q_1|/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-DIFFERENCE)}$$

Inference Rule Set 7.7: Inference rules for set operators

7.3.6 Inference Rules for Set Operators

As we have already mentioned in Section 5.1 on page 75, there exists a specific plan type for each set operation: union (Union), intersection (Intersect), and difference (Difference). In XTC, Union and Intersect consume n inputs and only Difference is a binary operator. The corresponding rules are specified in Inference Rule Set 7.7[9].

Rule CARD-UNION describes the cardinality inference for Union plans. For this operator, we assume that inputs $q_1 \ldots q_n$ have the same output cardinality and all input tuple sequences have the same active domains whose value range may differ. In this case, γ_{ik} denotes the active domain of operator i in column k.

[9] To improve readability, we replaced the plan names Union, Intersect, and Difference by their corresponding mathematical symbols "∪", "∩", and "\", respectively.

The cardinality inference for Intersect plans is given by rule CARD-INTERSECT, where q_k denotes the first operator whose cardinality is minimal with respect to the cardinality of the remaining operators. In experiments with our query optimizer, we found out that a constant factor of $2/3$ is a good heuristics for estimating the selectivity of n-way Intersect plans.

Finally, rule CARD-DIFFERENCE illustrates the cardinality inference of the binary Difference plans, where we simply fall back to the generalized 10% rule to estimate the domain sizes.

7.4 Related Work

Cardinality estimation in semi-structured database systems became first relevant in the context of the *Lore* project (McHugh et al., 1997). In Lore, DataGuides (Goldman and Widom, 1997) provide a simple method for handling the cardinalities of unique paths in XML documents.

XTC's statistics manager reuses this principle to provide the basic statistical information needed to bootstrap our cardinality inference rules (compare Section 3.1). If there are non-unique paths in an XML document, the DataGuide is insufficient to provide the necessary information. In recent years, numerous researchers proposed concepts for estimating the path cardinalities in such situations (e. g., compare the approaches of Aboulnaga et al., 2001; Freire et al., 2002; Zhang et al., 2006; Balmin et al., 2006; Fisher and Maneth, 2007; Aguiar Moraes Filho, 2010). Most of them are only concentrating on estimation accuracy and on minimal space consumption of their data structures. Even though path expressions are important building blocks of XQuery, these approaches do not help to optimize more complex queries.

The early work of Sartiani (2004) claims to discuss cardinality estimation of FLWR expressions[10], but focuses mostly on *for* expressions.

Having a look at native XML database management systems that provide cost-based query optimizers, for example, *Natix* (Fiebig et al., 2002) or *Timber* (Jagadish et al., 2002), shows that their cardinality estimation capabilities are restricted to cardinality estimation of simple path expressions[11].

To the best of our knowledge, MonetDB/XQuery, together with its XQuery compiler *Pathfinder*, is the only (non-native) XML database management system that supports XQuery cardinality estimation (Teubner et al., 2008). They use the general approach of abstract domain identifiers to estimate the value space that is taken by tuple items at runtime. As their cardinality inference

[10] Their approach does not cover complete FLWOR expressions (they omit *order-by* clauses).
[11] A comparison of different native XDBMSs is summarized in Table 2.1 on page 36.

rules are strongly tied to their logical algebra, they cannot be directly used in the context of XQGM. In contrast, the present work reuses their concept of abstract domain identifiers, but introduces a novel set of inference rules that allow to gain reliable cardinality estimates for XQGM instances.

7.5 Summary

In this chapter, we introduced our cardinality inference framework, which is based on the work of Teubner et al. (2008). In Section 7.1, we motivated the need for advanced XQuery cardinality estimation. Next, Section 7.2 provided the formal basis for the definition of the inference rules. Thereafter, we discussed the various inference rules in Section 7.3. In Section 7.4, we had a brief look at the shortcomings of related works.

In the forthcoming chapter, we introduce the cost model for our query optimizer. There, for effective cost estimation, we can dispose of our cardinality inference framework to derive the relevant cardinality information that will be bound to the cost formulae' parameters.

8 Cost Estimation

"We are always in search of the redeeming formula, the crystallizing thought."

(Etty Hillesum)

In previous chapters, we primarily dealt with logical aspects of query optimization by defining query rewrite rules (Chapter 4), introducing a plan abstraction (Chapter 5) that helped to specify various query transformation rules (Chapter 6), which empower a plan generator to derive numerous evaluation alternatives for an XQuery expression. In Chapter 7, we provided a comprehensive framework for cardinality inference. Now, we have almost all ingredients to build a plan generator that can span a tremendously large search spaces for XQuery expressions. To cope with them, we finally need a means for early eliminating bad plans. In the past, cost factors—which are provided by a cost model and that are assigned to plans—helped to distinguish efficient from inefficient implementations.

Section 8.1 provides preliminary definitions, which are necessary for defining the cost formulæ in Section 8.2. Finally, Section 8.3 summarizes this chapter and points out what you can expect in the subsequent chapters.

8.1 Introduction

The *cost model* is a system-dependent set of cost formulæ describing the costs of every physical algebra operator based on statistics provided by the system catalog as well as by using the cardinality estimates provided by the system's cardinality estimation framework.

In general, the cost of a physical algebra operator is estimated as the sum of Cost_{IO} and weighted Cost_{CPU}:

$$\text{Cost}_{\text{total}} = \text{Cost}_{\text{IO}} + w \cdot \text{Cost}_{\text{CPU}}$$

In our case, Cost_{IO} describe the costs raised by accessing an XML node, for example, an element node. In the context of database systems, this translates to costs for loading the data pages holding these nodes into the database

Constant	Description
$PageFetchCost_{IO}$	Cost for fetching a data page from the hard disk and loading it into the database buffer
$SortCost_{CPU}$	Cost for sorting a value
$ComparisonCost_{CPU}$	Cost for comparing two values in the main memory
$GroupingCost_{CPU}$	Cost for incrementally aggregating a value in the main memory
$UnnestingCost_{CPU}$	Cost for incrementally unnesting a value out of a tuple sequence
$HashingCost_{CPU}$	Cost for hashing a single value
$ProbingCost_{CPU}$	Cost for probing a single value in the hash table

Table 8.1: Overview of constant factors for cost formulæ

buffer[1]. On the other hand, Cost_{CPU} is raised by touching each node and sending it to a consecutive operator or by applying predicates to it. The weight w helps to adjust the estimates to CPU- or IO-bound situations.

Before we specify the different cost formulæ, we introduce some constants and functions that help to simplify their definition. The concrete values for the constant factors are system-specific, because they are primarily determined by the current hardware and operating system setup. A possible means for gaining the appropriate values is a calibration tool that runs several benchmarks on a specific platform to derive realistic values.

Table 8.1 summarizes the constant factors that we use in our cost formulæ. The IO costs of access paths are estimated based on the total numbers of pages that must be fetched from disk into a cold database buffer. Constant $PageFetchCost_{IO}$ specifies the cost for fetching a single page into the buffer. The CPU costs for sorting a single value and comparing a single value with a constant factor or another value is conducted using $SortCost_{CPU}$ and $ComparisonCost_{CPU}$, respectively. The XQGM data model supports nested tuples, hence, nesting and unnesting are important operations. The average cost for nesting a value into a tuple sequence is given by $GroupingCost_{CPU}$, whereas $UnnestingCost_{CPU}$ provides an average estimate for unnesting an item. Value-based hash join operators have two operations that primarily determine their costs: hashing and probing. In this context, $HashingCost_{CPU}$ represents the costs for hashing a single value and $ProbingCost_{CPU}$ provides the estimated cost for probing a single value.

Table 8.2 shows the various functions that we use in our cost model. The evaluation cost for a predicate p are system-specific and can be determined using the function $\text{EvaluationCost}(p)$. The function $\text{TotalCard}(x)$ returns the total number of elements in a data structure x (e. g., the total number of entries

[1] Please note, we conservatively assume that none of the required data pages currently resides in the buffer. Hence, for all pages, access costs must be accounted.

Function	Description
EvaluationCost(p)	Evaluation cost of predicate p
TotalCard(x)	Total number of elements in x
Card$_x(y)$	Total number of y in x
PageCard(x)	Total number of pages consumed by x
Selectivity(p)	The selectivity of predicate p with $0 \leq$ Selectivity$(p) \leq 1.0$
PathCard(p)	The total number of instances of path p
PathSelectivity$_i(e)$	Percentage of pages consumed by path e in index i: $\text{PathCard}(e)/\text{TotalCard}(i)$
BlockingFactor(x)	Average number of values per page in x: $\text{TotalCard}(x)/\text{PageCard}(x)$
h(i)	Height of B*-tree i

Table 8.2: Overview of functions for cost formulæ

in a document index). Additionally, we can use function Card$_x(y)$ to find out how many y's occur in data structure x (e. g., how many *book* element nodes are contained in an element index). We can use function PageCard(x) to get the total number of pages that are consumed by an access path x. With the help of Selectivity(p), we can determine the selectivity of a value-based predicate p. In many situations, it is useful to know the total number of occurrences of a path p. This statistical information is returned by function PathCard(p) that, in turn, uses the PS or EXsum to estimate the result. Finally, function h(i) identifies the height of B*-tree i, that is, the total number of pages that must be accessed to reach the first leaf node of i, if left-most depth-first traversal is applied.

8.2 The Cost Model

In this section, we define the cost formulæ for all physical operators that are relevant for XQuery processing in XTC. First, Section 8.2.1 introduces the cost formulæ of all access paths. In Section 8.2.2, we elaborate on the cost formulæ for PPOs. Thereafter, Section 8.2.3 shows how we estimate the costs of Select operators and value-based join operators. Next, Section 8.2.4 specifies the cost formulæ for grouping, unnesting, merging, and sorting as well as for all set operations.

8.2.1 Access Paths

In Section 3.1.3 on page 43, we introduced the access paths of XTC, which we classified into three classes: PAP (e. g., document index), SAP (e. g., element index), and TAP (e. g., path index and CAS index).

Document Root Access

Estimating the costs for accessing the root node of an XML document is very simple. Cost Formula 8.1 on page 132 illustrates the corresponding definition. For retrieving the document root, we have to fetch at least one page. Here, the CPU costs are negligible, hence, we assume only a figurative cost of 1.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= 1 \cdot \mathit{PageFetchCost}_{IO} \\ \text{Cost}_{\text{CPU}} &= 1 \end{aligned}$$

Cost Formula 8.1: Document root access

Document Index

The document index is the only access path in XTC that is available per default.

Figure 3.3(a) on page 44 depicts the document index in a schematic manner. To perform a complete scan over a document index d, we first have to descent to the first page. This operation raises $\text{h}(d)$ pages fetches. Next, we have to scan all data pages, where the total number of data pages is provided by function $\text{PageCard}(d)$. Actually, we only need to touch $\text{PageCard}(d) - 1$ data pages, because the first data page was already considered in the first estimation step.

The CPU costs of a document index scan are primarily dependent on the total number of nodes stored in the index, which can be determined using function $\text{TotalCard}(d)$. Finally, a document index scan can evaluate a predicate p on each node. The constant cost factor[2] for evaluating p is calculated using $\text{EvaluationCost}(p)$. Cost Formula 8.2 shows the complete specification.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= \Big[\text{h}(d) + \text{PageCard}(d) - 1\Big] \cdot \mathit{PageFetchCost}_{IO} \\ \text{Cost}_{\text{CPU}} &= \text{TotalCard}(d) \cdot \text{EvaluationCost}(p). \end{aligned}$$

Cost Formula 8.2: Document index scan

[2] Please note, if there is no predicate, $\text{EvaluationCost}(p)$ always returns 1.

Element Index

The element index was already introduced in Section 3.1.3. In Figure 3.3(b) on page 44, you can see an illustration of this index type. For accessing all elements with node name e from element index i, we once again have to descent to the data pages: First, we need to access $h(i_n)$ pages of the name directory to find the entry point for the relevant node-references index. In turn, to reach the data pages of the node-references index, we need $h(i_r)$ additional page fetches. Now, we can use the same approach as applied for the document index: We simply scan all data pages of the node-references index for element e. We assume that all element nodes in i are equally distributed over all node-references indexes. Hence, we can estimate the number of pages that must be fetched using $^{\text{Card}_i(e)}/_{\text{TotalCard}(i)} \cdot \text{PageCard}(i)$, where $\text{Card}_i(e)$ is the total number of e nodes in i, $\text{TotalCard}(i)$ returns the total number of element nodes in i, and $\text{PageCard}(i)$ represents the total number of pages consumed by i.

The estimation of the CPU costs follows the principle already applied for document index scans—except that we only use the total number of elements stored in the corresponding node-references index, instead of taking all nodes into account. Cost Formula 8.3 shows the respective definition.

$$\text{Cost}_{\text{IO}} = \left[h(i_n) + h(i_r) + \left\lceil \frac{\text{Card}_i(e)}{\text{TotalCard}(i)} \cdot \text{PageCard}(i) \right\rceil - 1 \right] \cdot \mathit{PageFetchCost}_{IO}$$

$$\text{Cost}_{\text{CPU}} = \text{Card}_i(e) \cdot \text{EvaluationCost}(p)$$

Cost Formula 8.3: Element index scan

Path Index

In Section 3.1.3, we already emphasized the importance of path indexes for efficient query processing in XTC. Let $p = \text{doc}(d)/n_1/n_2/\ldots/n_n$ denote a path in document d. A path index i, which covers p, provides access to all n_n leaf nodes.

We can distinguish between two clustering modes for path indexes: PCR clustering and SPLID clustering. If a path index is PCR-clustered and p matches more than one PCR, we need to sort the result in document order. In this situation, the plan generator injects a Sort plan above the index scan and takes the additional costs into account. On the other hand, if the path index

$$\text{Cost}_{\text{IO}} = \left\lceil \text{h}(i) + \left\lceil \text{PathSelectivity}_{\text{i}}(p) \cdot \text{PageCard}(i) \right\rceil - 1 \right\rceil \cdot \mathit{PageFetchCost}_{IO}$$
$$\text{Cost}_{\text{CPU}} = \text{Card}_i(p) \cdot \text{EvaluationCost}(\mathit{pred})$$

Cost Formula 8.4: Path index scan

is SPLID-clustered, we have to scan all data pages of the index to be sure that we do not miss a match.

Cost Formula 8.4 depicts the corresponding definition. For IO and CPU cost estimation, we apply the same principle as before for document indexes and element indexes. The fraction of data pages that must be fetched is primarily determined by function $\text{PathSelectivity}_{\text{i}}(p)$. If the index is SPLID-clustered, all data pages must be scanned, therefore, $\text{PathSelectivity}_{\text{i}}(p) = 1.0$. For PCR-clustered path indexes, $\text{PathSelectivity}_{\text{i}}(p)$ is defined as $^{\text{PathCard}(p)}/_{\text{TotalCard}(i)}$ (Table 8.2), where $\text{PathCard}(p)$ can be easily inferred using the PS and $\text{TotalCard}(i)$ is provided by XTC's metadata catalog.

Content-and-Structure Index

In Section 3.1.3, we learned that content-and-structure (CAS) indexes are hybrids of path and content indexes.

Let $p = \text{doc}(d)/n_1/n_2/\ldots/n_n/c$ be a path expression in document d, where c is a content node and n_n is either an attribute node or an element node. A CAS index provides access to all values of t and can handle point queries as well as range queries. Hence, the scan cost is reciprocally proportional to the selectivity of predicate p, that is, the lower the selectivity the more data pages must be read. If the index is SPLID-clustered, all data pages must be read to find all relevant records. Thus, we define $\text{PathSelectivity}_{\text{i}}(e) = 1.0$ in this case. If the CAS index is PCR-clustered, we might be forced to inject an additional sort operator above, for the same reason as explained before for path index scans.

The IO cost and CPU cost for an CAS index scan are depicted in Cost Formula 8.5. The formula is almost identical to the formula of a path index scan. Additionally, we use the function $\text{Selectivity}(p)$ to pay attention to range or point queries. The statistics that are necessary to calculate $\text{Selectivity}(p)$ can be gained from histograms or must be estimated using standard heuristics like the classical 10 % rule.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \Big[\text{h}(i) + \Big\lceil \text{PathSelectivity}_{\text{i}}(e) \cdot \text{Selectivity}(p) \cdot \text{PageCard}(i) \Big\rceil - 1\Big] \\ &\quad \cdot \mathit{PageFetchCost}_{IO} \\ \text{Cost}_{\text{CPU}} &= \text{Card}_i(p) \cdot \text{Selectivity}(p) \cdot \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.5: CAS index scan

Navigational Operator

Table 6.1 on page 88 mentioned that XTC provides a navigational access operator that returns all nodes being structurally related to a given context node. Such context-dependent accesses are possible for all types of access paths: PAPs, SAPs, and TAPs. Let $s = c\,\theta\,n$ be a location step and $s' = c[\theta\,n]$ be a predicate step, where c is the context node, θ is an XPath axis, and n is a node name (that might contain an additional point predicate or range predicate). We use EXsum to estimate the selectivity of s or s' and multiply it with the IO cost (as specified by Cost Formulæ 8.2–8.5) that would be raised if we would completely scan the corresponding access path. Doing cardinality estimation for every context node is too costly and is expected to provide only little gain in precision. Thus, we assume that all context nodes have an equal number of n nodes related to it via θ and calculate the estimate only once.

8.2.2 Path Processing Operators

If no path indexes or CAS indexes can be exploited, structural relationships—or cascades of them formed by path expressions—must be evaluated using PPOs such as *NavTree* (context-dependent nested-loops join), *Extended Stack-Tree* (structural join), or *Extended Twig Opt* (holistic twig join)—a detailed description of these operators is given by Mathis (2009).

NavTree

As already mentioned in Section 6.2.1, NavTree resembles the functionality of a classical nested-loops join, except that it is context-dependent. Cost Formula 8.6 shows how we estimate the IO and CPU costs of NavTree.

We assume that the left input provides the evaluation context. Consequently, the IO cost are raised only once for this input. In contrast, for every tuple provided by the left input, the right input is evaluated. Therefore, $\text{TotalCard}(\mathit{left}) \cdot \text{Cost}_{\text{IO}}(\mathit{right})$ IO costs are raised for the right input. In fact,

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\mathit{left}) + \text{TotalCard}(\mathit{left}) \cdot \text{Cost}_{\text{IO}}(\mathit{right}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\mathit{left}) + \text{TotalCard}(\mathit{left}) \cdot \text{Cost}_{\text{CPU}}(\mathit{right}) \cdot \\ &\quad \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.6: NavTree

the CPU costs of NavTree are estimated using the same idea. The only difference is function EvaluationCost(p) that is used to estimate the costs for evaluating the structural predicate p, that is, an XPath axis.

Extended StackTree and Extended TwigOpt

In Section 6.3.2 on page 93, we introduced several transformation rules for structural joins. To allow the plan generator to decide which alternative is the most promising one, CostFormula 8.7 provides a way for deriving the expected IO and CPU costs. In a certain way, *Extended StackTree* (see Section 2.2.1 on page 24) is similar to a relational merge join, where each input is read only once, hence, we can simply sum up the IO costs of the left and right input.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\mathit{left}) + \text{Cost}_{\text{IO}}(\mathit{right}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\mathit{left}) + \text{Cost}_{\text{CPU}}(\mathit{right}) + \text{TotalCard}(\mathit{left}) \cdot \\ &\quad \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.7: Extended StackTree

The CPU costs of Extended StackTree are estimated as the sum of the CPU costs of its input operators and the costs for evaluating the structural predicate p. Holistic twig joins are able to evaluate a cascade of structural joins at once. For estimating the costs of *Extended TwigOpt* (Section 2.2.1), we use Cost Formula 8.8 and apply it to every location or predicate step expressed by the twig pattern.

At first sight, Cost Formula 8.7 and Cost Formula 8.8 have identical definitions. Having a closer look at them reveals that they differ in the shape of their EvaluationCost(p) function. We will see later in Chapter 12 that, in many situations, the execution times of structural joins and holistic twig joins

do not differ significantly. Therefore, we use these simplified cost formulæ for cost estimation. In Section 12.4.2 we will show how the EvaluationCost(p) function can be derived for Extended StackTree and Extended TwigOpt.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\textit{left}) + \text{Cost}_{\text{IO}}(\textit{right}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\textit{left}) + \text{Cost}_{\text{CPU}}(\textit{right}) + \text{TotalCard}(\textit{left}) \cdot \\ &\quad \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.8: Extended TwigOpt

8.2.3 Select Operators

In previous chapters, for example, in Chapter 7, we saw that Select plans can occur in various shapes. First, we will look at the cost formulæ of Select plans evaluating value-based joins. Next, we discuss cost estimation for simple (unary) Select plans. Finally, we will focus on the cost formulæ of complex (n-ary) Select plans.

Value-Based Joins

The definitions of the cost formulæ for value-based joins are very close to their relational counterparts. Cost Formula 8.9 illustrates the IO and CPU cost estimation for the value-based nested-loops join. The IO costs are estimated by simply following the nested-loops evaluation paradigm: For every left input tuple, we have to compare it to the whole input tuple sequence provided by the right input. The same approach is used for CPU cost estimation, except that we now additionally consider, in every step, the cost for evaluating the value-based predicate.

$$\begin{aligned}\text{Cost}_{\text{IO}} =& \text{Cost}_{\text{IO}}(\textit{left}) + \text{TotalCard}(\textit{left}) \cdot \text{Cost}_{\text{IO}}(\textit{right}) \\ \text{Cost}_{\text{CPU}} =& \text{Cost}_{\text{CPU}}(\textit{left}) + \text{TotalCard}(\textit{left}) \cdot \text{Cost}_{\text{CPU}}(\textit{right}) \cdot \\ & \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.9: Value-based nested-loops join

In Cost Formula 8.10, we can see how the cost for evaluating a value-based merge join is determined. Using the IO costs of the left input—$\text{Cost}_{\text{IO}}(\textit{left})$—and the right input—$\text{Cost}_{\text{IO}}(\textit{right})$—, we simply sum up both numbers to derive the IO cost estimate, because we only need to read each input once. For estimating the CPU costs, we use the CPU cost estimates of the input operators and additionally consider the evaluation cost of the value-based predicate p. Function EvaluationCost(p) returns the cost for evaluating p for two tuples—one from each input stream. Conservatively, we multiply it with the highest input cardinality.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\textit{left}) + \text{Cost}_{\text{IO}}(\textit{right}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\textit{left}) + \text{Cost}_{\text{CPU}}(\textit{right}) \\ &\quad + \max\Big(\text{TotalCard}(\textit{left}), \text{TotalCard}(\textit{right})\Big) \cdot \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.10: Value-based merge join

Finally, Cost Formula 8.11 depicts the formula for a value-based hash join. For IO cost estimation, we use the same idea as already described for value-based merge joins. In contrast, for estimating the CPU costs, we consider the two main operations of a hash join: hashing and probing. Besides taken the CPU costs of input operators into account, we assume that the left input is hashed and the right input is probed against the hash table. Hence, the costs for hashing are dependent on the cardinality of the left input, whereas the probing costs are directly related to the cardinality of the right input. Using the constants $\textit{HashingCost}_{CPU}$ and $\textit{ProbingCost}_{CPU}$—which describe the costs for hashing and probing a single value, respectively—we can fine-tune the formula to match a specific system environment.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\textit{left}) + \text{Cost}_{\text{IO}}(\textit{right}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\textit{left}) + \text{Cost}_{\text{CPU}}(\textit{right}) \\ &\quad + \text{TotalCard}(\textit{left}) \cdot \textit{HashingCost}_{CPU} + \text{TotalCard}(\textit{right}) \cdot \\ &\quad \textit{ProbingCost}_{CPU}\end{aligned}$$

Cost Formula 8.11: Value-based hash join

Simple and Complex Selection

Simple Select plans, which contain only one tuple variable, serve for evaluating comparison predicates or positional predicates. Cost Formula 8.12 illustrates the corresponding definition. Evaluating a simple selection does not raise additional IO. Hence, we only need to take the IO cost of its input operator into account. Moreover, the CPU cost is determined using the CPU cost of the input operator and the cost for checking the predicate on each input tuple.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\mathit{input}) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(\mathit{input}) + \text{TotalCard}(\mathit{input}) \cdot \text{EvaluationCost}(p) \end{aligned}$$

Cost Formula 8.12: Simple selection

A complex Select plan evaluates multiple *for*-quantified, *let*-quantified, and *exists*-quantified tuple variables. Checking *exists*-quantified tuple variables does not raise IO costs, because the actual tuples used for predicate checking are provided by *for*-quantified and *let*-quantified tuple variables. Let the Select operator have n input operators, where the first one triggers the evaluation process. Cost Formula 8.13 shows the corresponding estimation method. The IO costs are estimated using a generalization of the cost formula for nested-loops joins (Cost Formula 8.9). Quintessentially, the Select operator evaluates a complex Cartesian product on n input streams. Therefore, we estimate the CPU costs as the sum of all input operator's CPU costs and the product of all costs for accessing the input operator's tuple sequences.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(\mathit{input}_1) + \text{TotalCard}(\mathit{input}_1) \cdot \sum_{i=2}^{n} \text{Cost}_{\text{IO}}(\mathit{input}_i) \\ \text{Cost}_{\text{CPU}} &= \sum_{i=1}^{n} \text{Cost}_{\text{CPU}}(\mathit{input}_i) + \prod_{i=1}^{n} \text{TotalCard}(\mathit{input}_i) \cdot \text{EvaluationCost}(p) \end{aligned}$$

Cost Formula 8.13: Complex selection

8.2.4 Miscellaneous Operators

In this section, we introduce the cost formulæ of the remaining physical operators. First, we will have a look at the formulæ for the Project operator and the DDO operator. Next, we introduce the cost formulæ for GroupBy and Unnest. Thereafter, we discuss cost estimation for the Split operator. As a follow-up action, we define the cost formulæ for Merge and the three set-based operators (Union, Intersect, and Except). Finally, we have a look at the cost formula for the Sort operator.

Projection and Duplicate Elimination

In our cost model, we use the same cost formula for the Project operator and the DDO operator. Cost Formula 8.14 depicts the corresponding definition. The IO cost is simply derived from the associated input operator. We assume that the cost for comparing an input tuple with the projection specification (Project operator) or checking whether the tuple is a duplicate (DDO) raises a constant cost. Therefore, we multiply the input cardinality with $ComparisonCost_{CPU}$, which describes the cost for performing an in-memory comparison.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(input) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(input) + \text{TotalCard}(input) \cdot ComparisonCost_{CPU} \end{aligned}$$

Cost Formula 8.14: Project and DDO

GroupBy and Unnest

GroupBy and Unnest are two complementary operators in XTC. For cost estimation, we use the same idea as for the Project and DDO operator. For considering the cost for grouping a single tuple, we assume that this raises $GroupingCost_{CPU}$ cost units. In contrast, let us assume that unnesting a tuple out of a tuple sequence costs $UnnestingCost_{CPU}$ units. Let $c \in \{GroupingCost_{CPU}, UnnestingCost_{CPU}\}$, then Cost Formula 8.15 shows the corresponding definition for GroupBy and Unnest, respectively.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(input)\\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(input) + \text{TotalCard}(input) \cdot c\end{aligned}$$

Cost Formula 8.15: GroupBy and Unnest

Split

The Split operator is a trivial operator in our physical algebra. It simply splits the data flow and passes its input to multiple consumers. For the sake of completeness, Cost Formula 8.16 shows the simplistic definition.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(input)\\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(input)\end{aligned}$$

Cost Formula 8.16: Split operator

Merge and Set

For cost estimation of Merge and the three set-based operators (Union, Intersect, and Difference), we use Cost Formula 8.17. Here, the IO cost is determined by summing up all IO costs of the input operators. For CPU cost estimation, we assess the CPU costs of the input operators and the costs for evaluating the Cartesian product on n inputs as total CPU costs for Merge and Set.

$$\begin{aligned}\text{Cost}_{\text{IO}} &= \sum_{i=1}^{n} \text{Cost}_{\text{IO}}(input_i)\\ \text{Cost}_{\text{CPU}} &= \sum_{i=1}^{n} \text{Cost}_{\text{CPU}}(input_i) + \prod_{i=1}^{n} \text{TotalCard}(input_i) \cdot \text{EvaluationCost}(p)\end{aligned}$$

Cost Formula 8.17: Merge and set operators

Sort

Finally, we estimate the costs of the Sort operator using Cost Formula 8.18. Here, we use *quicksort* (Hoare, 1962) as an in-memory sort algorithm. Though, no additional IO is raised. Hence, the IO cost information is provided by the input operator. According to Cormen et al. (2001), quicksort's average case complexity is in $O(n \cdot \log n)$. Therefore, we can use this relationship for estimating the CPU costs of the Sort operator, where we replace n by the input cardinality and assume a constant factor $SortCost_{CPU}$ that represents the average sort cost per value.

$$\begin{aligned} \text{Cost}_{\text{IO}} &= \text{Cost}_{\text{IO}}(input) \\ \text{Cost}_{\text{CPU}} &= \text{Cost}_{\text{CPU}}(input) + \Big[\text{TotalCard}(input) \cdot \log\Big(\text{TotalCard}(input)\Big)\Big] \cdot \\ & \quad SortCost_{CPU} \end{aligned}$$

Cost Formula 8.18: Sort operator

8.3 Summary

In this chapter, we provided the cost formulæ for XTC's physical algebra operators. In Section 8.1, we introduced the nomenclature necessary for the definition of the cost formulæ. Thereafter, Section 8.2 discussed the definition of the numerous cost formulæ.

By ending this chapter, we almost reach the end of Part II. So far, we developed all ingredients to build a cost-based XQuery optimizer. In the following chapter, we will finally look at the plan generator that combines the concepts for query rewrite and transformation as well as cardinality and cost estimation.

9 Plan Generation

"A good plan, violently executed now, is better than a perfect plan executed next week."

(George S. Patton)

In this chapter, we reconsider all concepts developed in Part II so far and put all pieces together to finally establish our plan generation approach.

First, Section 9.1 discusses the importance of effective plan generation for efficient XQuery optimization in native XDBMSs.

Section 9.2 details our strategies for plan generation. First and foremost, Section 9.2.1 describes the preliminary steps necessary to start plan generation. Section 9.2.2 introduces our bottom-up plan generation algorithm, which is used as default plan generation strategy. Thereafter, we provide insights into our top-down plan generation algorithms in Section 9.2.3.

In Section 9.3, we review related research papers and recommend textbooks for further reading.

Finally, we summarize this chapter in Section 9.4. As this chapter completes Part II, we take a retrospective view on the concepts developed so far.

9.1 Introduction

In Section 2.2.1 on page 24, we argued that one of the major challenges of cost-based XQuery optimization is handling the ever growing set of PPOs and indexes. Due to the duality of value-based and structural predicates in XQuery, the search space becomes even larger than in the relational case.

Before we look at search space sizes of XQuery expressions, let us first analyze the situation in the relational case. We consider a relational SPJ (select-project-join) query with n relations. For such a query, the upper bound for the search space size is the total number of permutations that can be derived. Hence, we can derive $n!$ possible join orders in the worst case (Moerkotte, 2009), if we allow cross products and assume that every relation can be joined with another one. Actually, in real-world query processing scenarios, this hypothetical upper bound is not met, because (1) the domains of join attributes differ and (2) the plan generator's ability to derive all possible

join orders is constrained by heuristics, for example, interesting join orders (see Section 2.3.1).

If we consider XPath expressions without path predicates, for example, `a/b/c`, we observe that they are so-called *chain queries* without cross products (Moerkotte, 2009). This fact leads to a dramatic restriction of the search space size. For example, for the aforementioned query, a join order like $\Big((a \bowtie_{\texttt{child}} c) \bowtie_{\texttt{child}} b\Big)$ is not allowed, because a is not directly related to c via the *child* axis.

The search space sizes for chain queries without cross products depend on the shapes of query trees that can be derived by the plan generator. According to Moerkotte (2009), for a chain query with n relations, we can calculate the search space sizes for left-deep plans, zig-zag plans, and bushy plans[1], respectively, using the following formulæ:

Left-deep plans	*Zig-zag plans*	*Bushy plans*
2^{n-1}	2^{2n-3}	$2^{n-1} \cdot C(n-1)$

For the further discussion, we denote the search space size of a chain query without cross product by $f(n)$. Let us assume that we can choose for each relation out of k access paths and each join operator has m different implementations. For each possible join order, we can derive m^{n-1} implementation variants for the join operators and k^n implementation variants (access paths) for the n relations. In total, the search space for such a query can be calculated using:

$$f(n) \cdot m^{n-1} \cdot k^n$$

If we transfer this to the XML world and consider an XPath expression, then n is equal to the total number of node tests involved, m is the total number of binary PPOs available, and k is the total number of PAPs and SAPs present in a system.

If we take this for granted, we observe the following: For typical XPath expressions, n will be much larger than in the relational world, because n directly depends on the depth of the queried document path. Moreover, m and k will also be larger than in the relational context, because the set of PPOs and access paths is still growing.

So far, the formula for search-space size approximation considers only StructuralJoin plans. In Chapter 6, we introduced *join fusion* and *TAP detection* as additional means for deriving query plans.

Using join fusion, we can derive hybrid plans consisting of StructuralJoin plans and a TwigJoin as well as plans that completely evaluate a path expres-

[1] Here, numbers $C(n)$ are referred to as *Catalan numbers* (Cormen et al., 2001), where $C(n) = \frac{1}{n+1}\binom{2n}{n}$.

n	SJ reordering	Join fusion	TAP detection	Total
1	4	–	–	4
2	64	–	1	65
3	1,024	64	9	1,097
4	16,384	1,280	73	17,737
5	262,144	21,504	585	283,733
6	4,194,304	348,160	4,681	4,547,145

Table 9.1: Search space sizes for path expressions with $1 \leq n \leq 6$ node tests

sion using a single TwigJoin. We can apply join fusion only, if the join tree involves two or more structural joins. The total number of hybrid join orders—for a path expression with $n > 2$ node tests—having at least one TwigJoin plan can be calculated using the following formula[2]: $1 + \sum_{i=2}^{n-2} f(n-i)$.

Let us assume that we have o TwigJoin implementations, k access paths, and m StructuralJoin implementations, we get: $k^n \cdot o \cdot \left(1 + \sum_{i=2}^{n-2} f(n-i) \cdot m^{n-i-1}\right)$ hybrid plans.

Additionally, we can iteratively replace (parts of) cascades of StructuralJoin plans by a single IndexAccess plan (see Section 6.3.4). In this situation, we do not consider different join orders. TAP detection can be applied on path expressions with $n > 1$ node tests. For example, if there is an IndexAccess plan that covers two location steps of an n-step path expression, we can still assign different implementations to $n-3$ Access plans as well as implementations to $n-3$ StructuralJoin plans. Furthermore, let us assume that there are j different implementations for IndexAccess plans, k implementations for Access plans, and m implementations for StructuralJoin plans. We calculate the total number of different plans generated by TAP detection using the following formula: $j \cdot \sum_{i=2}^{n} (k \cdot m)^{n-i}$.

Finally, the total number of plans that can be derived for n-step path expressions using our plan generation approach is:

$$\underbrace{f(n) \cdot m^{n-1} \cdot k^n}_{\text{SJ reordering}} + \underbrace{k^n \cdot o \cdot \left(1 + \sum_{i=2}^{n-2} f(n-i) \cdot m^{n-i-1}\right)}_{\text{Join fusion}} + \underbrace{j \cdot \sum_{i=2}^{n} (k \cdot m)^{n-i}}_{\text{TAP detection}}$$

Table 9.1 illustrates the search space size for a typical query optimizer configuration, where we assume $m = 2$, $k = 4$, $o = 1$, $j = 1$, and left-deep plans, that is, $f(n) = 2^{n-1}$. The results reflect the search space sizes for path expressions with $1 \leq n \leq 6$ node tests.

[2]See Appendix D on page 231 for the formal proof.

Structural join reordering still dominates the search space size. Though, by additionally considering all possible plans created by join fusion and TAP detection, the search space is notably increased. Hence, a complete traversal of the search space is not feasible in acceptable time. Therefore, we have to restrict the search space using two strategies: (1) prohibit the application of non-promising transformation rules and (2) early pruning of inefficient subtrees. In the relational world, especially in the case of System R, strategy 1 primarily focuses on interesting join orders (see Section 2.1.2 on page 14). Our plan generator does not consider StructuralJoin plans that introduce additional structural full joins. Moreover, for join fusion, we only consider left-deep plans. In Section 2.1.2, we also learned that exhaustive search strategies traverse the complete search space. Even if we apply strategy 1, the search space will remain fairly large. Therefore, in strategy 2, for every iteration of the exhaustive plan generation algorithm, we only retain the cheapest solution and do not consider more expensive ones any further. This idea is also borrowed from classical relational query optimization. As already shown by Ono and Lohman (1990), if a Starburst-style plan generator with bottom-up enumeration is used, linear queries with a hundred or more tables, which are similar to SJ trees for path expressions, can be optimized in acceptable time. In Chapter 12, we will see that an XQuery optimizer that considers a search space restricted by strategies 1 and 2, can deliver decent results.

9.2 Strategies for Plan Generation

Section 9.2.1 discusses preliminary steps for plan generation that are performed for every plan generation strategy. The cost-based XQuery optimizer of XTC uses a bottom-up plan generation algorithm (Section 9.2.2) as default search strategy. Nevertheless, our optimization framework is not restricted to it. If query processing scenarios involve queries with very long path expressions, for example, expressions that consist of 15 or more location steps, we provide several top-down strategies (Section 9.2.3), which do—in contrast to our bottom-up strategy—not inspect the entire search space, but take only random walks through it to cope with the immensely large search space.

9.2.1 Preliminaries

Before we can start with plan generation, we have to perform some preliminary tasks. Each plan generation strategy takes an XQGM instance as input. In Section 5.2 on page 79, we discussed how XQGM instances are mapped

Input: A stack *Open* with *Goal*.size() ≥ 1
Output: A stack *Goal* containing the cheapest plan, where *Goal*.size()=1 and $\mathcal{P}$ is a set of plans with $\mathcal{P} \in$ Goal, $|\mathcal{P}| = 1$

```
1  Goal ← ∅;
2  while ¬ Open.isEmpty() do
3  |  P ← Open.pop();
4  |  if ¬ isGoal(P) then
5  |  |  S ← ∅, S' ← ∅;
   |  |  /* Derive all semantically equivalent plans.          */
   |  |  /* Invariant:  |P|=1                                  */
6  |  |  S ← action(P);
   |  |  /* Eliminate all sub-optimal alternatives            */
7  |  |  S' ← prune(S);
8  |  |  Open.push(S')
9  |  else
   |  |  /* Invariant:  |P| = 1                               */
10 |  |  Goal.push(P);
11 |  end
12 end
13 return Goal;
```

Algorithm 9.1: Skeleton bottom-up search algorithm (according to Lanzelotte and Valduriez, 1991)

onto plan graphs—in most cases, by a simple 1:1 mapping. As we have already pointed out in Section 5.2, plan graphs are actually trees, because all operators consuming from XQGM Split operators receive their inputs from dummy SplitRef plans. For the previously derived plan graph, we perform a bottom-up traversal and apply the cardinality inference rules, which we introduced in Chapter 7, to annotate each plan with the corresponding output cardinalities ($|p|$) and derive the abstract domain identifiers for each operator (p_{adi}). Whenever we transform a plan into a semantically equivalent alternative, we simply copy the cardinality information as well as the abstract domain identifiers to allow for cost estimation.

Each search strategy takes a stack as input (*Open*) and returns a stack (*Goal*) as well. In both cases, the stack elements are sets of semantically equivalent plans. For bottom-up strategies, we perform a left-most depth-first traversal of the plan graph and push each plan in reverse order onto the *Open* stack in such a way that each plan forms a singleton set. Moreover, all pointers to parent and child plans are eliminated for the stack elements. Top-down strategies use a complete plan graph as starting point. Therefore, its *Open* stack contains exactly one element. The single stack element is a set that contains

the plan graph's root node as single element. Here, we do not eliminate the parent and child pointers, but keep the hierarchical structure intact.

Input: A set of plans $\mathcal{P}$
Output: A set of plans $\mathcal{P}''$

```
1  P' ← ∅, P'' ← ∅;
   /* Apply implementation variation rules to plan in P.          */
   /* Invariant:  |P| = 1                                         */
2  P' ← expand(P[0]);
   /* Apply structural variation rules to all plans in P'         */
3  P'' ← transform(P');
4  return P''
```

Function 9.1: action($\mathcal{P}$)

Input: A set of plans $\mathcal{P}$ with $|\mathcal{P}| \geq 1$
Output: A singleton set $\mathcal{P}'$ with $|\mathcal{P}'| = 1$

```
1  minCost ← 0;
2  p' ← NIL ;
3  forall p ∈ P do
   |  /* Estimate the sum of IO and CPU cost                       */
4  |  currCost ← getCostEstimate(p);
5  |  if p' = NIL then
6  |  |  p' ← p;
7  |  |  minCost ← currCost;
8  |  end
9  |  else
10 |  |  if currCost < minCost then
11 |  |  |  p' ← p;
12 |  |  |  minCost ← currCost;
13 |  |  end
14 |  end
15 end
16 P' ← {p'};
17 return P'
```

Function 9.2: prune($\mathcal{P}$)

9.2.2 Bottom-Up Plan Generation

Let us assume that we already can dispose of the *Open* stack that contains all plan nodes in reverse traversal order as single-set elements. Let $\mathcal{P}$ be a set of semantically equivalent plans and $\mathcal{S}$ be the set of all plans—including those contained in $\mathcal{P}$—that can be derived from plans contained in $\mathcal{P}$ using

the transformation rules introduced in Section 6.2 and 6.3, for example, join fusion.

Algorithm 9.1 shows the skeleton of our exhaustive bottom-up search algorithm, which follows the idea of generalized search strategy frameworks introduced by Lanzelotte and Valduriez (1991). Function isGoal (line 4) returns *true*, if and only if, $\forall\, p \in \mathcal{P} : p_{\text{isGoal}=true}$, otherwise it returns *false*.

The two most important parts of this basic algorithm are lines 6 and 7. In line 6, we use Function 9.1 (action) to derive semantically equivalent plans (expressed as the set of successor plans $\mathcal{S}$) from the input set $\mathcal{P}$, where $\mathcal{P} \subseteq \mathcal{S}$ holds.

Function 9.1 (action) encapsulates all wisdom on how semantically equivalent alternatives are generated. In the scope of this function, the generation of alternatives is performed in two separate operations: expand and transform. Function 9.3 (expand) creates alternatives by applying implementation variation rules (Section 6.2). In contrast, Function 9.4 (transform) uses structural variation rules (Section 6.3) to get further alternatives.

We have pointed out before that the complete search space cannot be traversed due to its exponential growth. Therefore, line 7 calls Function 9.2 (prune). This function inspects each alternative plan in $\mathcal{P}$ and retains only a single plan, whose estimated cost is minimal amongst all alternatives. Cost estimation is performed using the getCostEstimate function. During the initialization step, we already assigned cardinalities to each plan. For cost estimation, function getCostEstimate simply performs a bottom-up traversal on plan p and uses the cost formulæ introduced in Chapter 8 to get the estimated cost (currCost), which is calculated as the weighted sum of IO cost (p_{IOcost}) and CPU cost (p_{CPUcost}). Based on the estimate, we decide in line 10 whether the current plan is cheaper than previous alternatives. If this is the case, we remember its cost and the plan itself. Finally, we return $\mathcal{P}'$ in line 16, which contains the cheapest plan.

Function 9.3 (expand) uses the already finished plans, which were put on stack Goal by previous plan generation steps. If the input plan p has no children (line 1–11), that is, it is a leaf node such as an Access plan, we simply retrieve all possible implementations for it from the metadata catalog (line 3) and create exact copies of p that only differ in their implementation (line 4–9). If p is an inner node, for example, a StructuralJoin plan, we have to do a little bit more work. If we already have produced more than one goal plan, we have to derive all possible combinations of them with respect to different implementations. First, we check how many inputs are required by plan p (line 15). If only one input is needed, we simply fetch the top-most plan set from the stack and create the set of singleton plan sets P. Otherwise (line

Input: A plan p
Output: A set of plans $\mathcal{P}$

```
1  if p_{|children|=0} then
2      P' ← ∅;
       /* Derive all possible implementations for p                    */
3      I ← getImplementations(p);
4      forall i ∈ I do
5          p' ← clone(p);
6          p'_{impl ← i};
7          p'_{isGoal ← true};
8          P' ← P' ∪ {p'};
9      end
10     return P';
11 end
12 else
13     P ← ∅;
14     if Goal.size() > 1 then
           /* Determine the total number of input operators for the
              current implementation of p                              */
15         noComb ← getNoChildren(p_impl);
16         if noComb = 1 then
17             P'' ← Goal.pop();
18             forall p ∈ P'' do
19                 P ← P ∪ {{p}};
20             end
21         end
22         else
               /* Create all possible combinations of input plans
                  (different implementations), which are residing on
                  stack Goal; P is a set of plan sets                  */
23             P ← combine(noComb, Goal);
24         end
           /* For each input combination in P, create a new plan using p
              as root and P ∈ P as input plans                         */
25         P''' ← createGoal(p, P);
26         return P''';
27     end
28     else if Goal.size() = 1 then
29         P''' ← createGoal(p, Goal);
30         return P''';
31     end
32     else
33         return ∅;
34     end
35 end
```

Function 9.3: expand(p)

Input: A set of plans $\mathcal{P}$
Output: A set of plans $\mathcal{P}'$

```
1  P' ← ∅;
   /* Get the set of all structural variation rules (T) from the
      metadata catalog, for example, SJ reordering or join fusion     */
2  T ← getStructuralVariationRules();
3  forall p ∈ P do
4      P' ← P' ∪ {p};
5      forall t ∈ T do
           /* Try to apply t on p, if possible, derive a new semantically
              equivalent plan p'                                       */
6          p' ← apply(t, p);
7          if p' ≠ NIL then
8              p'_{isGoal ← true};
9              P' ← P' ∪ {p'};
10         end
11     end
12 end
13 return P';
```

Function 9.4: transform($\mathcal{P}$)

22), we use the combine function that pops noComb plan sets from *Goal* and combines them in such a way, that all permutations of plans from the plan sets are derived. For example, let us assume that noComb $= 3$ and we popped the following plan sets[3]: $\mathcal{P}_1 = \{a_1, a_2\}$, $\mathcal{P}_2 = \{b_1, b_2\}$, and $\mathcal{P}_3 = \{c_1, c_2, c_3\}$. As a result, combine returns $P = \Big\{\{a_1, b_1, c_1\} \ldots \{a_2, b_2, c_3\}\Big\}$. In line 25, we call the createGoal function. For every implementation of p, and every input candidate $\mathcal{P}_i \in P$, we create a new plan p' that uses $\mathcal{P}_i$ as input ($p'_{\text{children} \leftarrow \mathcal{P}_i}$).

Finally, in line 28, we handle the trivial case that there is only a single goal plan. Therefore, we can skip the rather expensive combine step and immediately derive the output.

After applying implementation variation to the input plan, we can now use the structural variation rules to further enlarge the search space. Function 9.4 (transform) shows how these rules are applied. First, we fetch all structural variation rules from the metadata catalog (line 2). For each plan $p \in \mathcal{P}$, which was previously formed by Function 9.3, we try to apply each structural variation rule $t \in \mathcal{T}$ (line 6). If the application was successful, we simply collect the new alternative in the result set $\mathcal{P}'$, which also contains the original plan p.

[3]Please note, each plan set $\mathcal{P}_i$ contains only semantically equivalent plans that differ only in p_{impl}.

9.2.3 Top-Down Plan Generation

In Section 2.1.2 on page 15, we already introduced the basic principles of three top-down plan generation strategies: Iterative Improvement, Simulated Annealing, and 2-Phase Optimization. In contrast to bottom-up approaches, which can potentially traverse the entire search space, top-down approaches take random walks through parts of the search space (compare Section 2.1.2 on page 14). Even though we do not assume that bottom-up search will meet its limits for standard queries, we also implemented top-down strategies for allowing to compare both classes of algorithms with respect to the quality and efficiency of their optimization results.

For the sake of brevity, we will only have a look at our implementation of Simulated Annealing, as the implementation of Iterative Improvement and 2-Phase Optimization is very similar to it. Especially, for selecting a neighbor plan in the search space, our implementations of Iterative Improvement, Simulated Annealing, and 2-Phase Optimization use the same approach. Moreover, the principles of Iterative Improvement and 2-Phase Optimization were discussed adequately before.

As a starting point for query optimization, we use a complete plan graph, where each plan is assigned its default implementation. To improve optimization results, we already do cost-based optimization at the initialization stage by assigning the cheapest access path to each Access plan in the initial graph. Thereafter, the plan is passed on to the actual search algorithm.

Algorithm 9.2 depicts our implementation of Simulated Annealing. First, we estimate the cost of the initial plan as a reference point for further optimization steps (line 3). The initial temperature t is calculated using the runtime constant *INIT_TEMPERATURE* (line 6). Global optimization continues as long as the condition[4] of function stop() is not satisfied (line 7). Local optimization stops (line 9) if an equilibrium is reached, that is, n local optimizations were performed, where n is proportional to the total number of join operators involved in the query.

In line 10, we choose a neighbor plan in the search space using function getNeighbor(p_c). This function traverses p_c in random order: At each subtree, this function randomly chooses either (1) an implementation variation rule or (2) a structural variation rule. If the application of the rule selected in (1) or (2) was successful, the newly derived plan is returned. Otherwise, random traversal continues until a variation was applied. Consequently, the neighbor plan derived using this function differs from p_c either in a different

[4] Global optimization stops, if $t <$ FROZEN_TEMPERATURE or costUnchangedCount $=$ FROZEN_COUNT.

Input: An initial plan p
Output: The cheapest plan p' in the considered part of the search space

```
1  p_min ← p;
2  p_c ← p;
3  minCost ← getCostEstimate(p);
4  costUnchangedCount ← 0;
5  localOptCount ← 0;
6  t ← INIT_TEMPERATURE · minCost;
7  while ¬stop() do
8      noLocalOpt ← 0;
       /* Continue until a sufficient number of alternatives was
          inspected (localOptCount = (EQUIL_FACTOR · getNoJoins(p)))   */
9      while ¬equilibrium() do
           /* Randomly choose a neighbor plan by applying impl.
              variation or structural variation to p_c                 */
10         p_l ← getNeighbor(p_c);
11         localCost ← getCostEstimate(p_l);
12         currCost ← getCostEstimate(p_c);
13         Δ ← (localCost − currCost);
14         if Δ ≤ 0 then
15             p_c ← p_l;
16         end
17         else
18             prob ← e^(−(Δ/t));
19             if prob ≥ PROB_THRESHOLD then
20                 p_c ← p_l;
21             end
22         end
23         cost ← getCostEstimate(p_c);
24         if cost < minCost then
25             p_min ← p_c;
26             minCost ← cost;
27             costUnchangedCount ← 0;
28         end
29         else
30             costUnchangedCount++;
31         end
32         localOptCount++;
33     end
34     t ← t · TEMP_REDUCTION;
35 end
36 return p_min;
```

Algorithm 9.2: Simulated Annealing

implementation for a subplan or in an alternative structural composition (e. g., rearranged join orders).

As already argued in Section 2.1.2, the quality and the efficiency of the optimization results of top-down search algorithms is critically dependent on how the runtime constants (e. g., *PROB_THRESHOLD*) are chosen.

9.3 Related Work

In Section 9.2, we outlined the top-down and bottom-up search strategies currently implemented in our framework. Fairly, there are many more plan generation techniques than covered by this thesis. Nevertheless, we believe that our query optimization framework is general enough to integrate further bottom-up and top-down plan generation strategies.

Research Papers In Section 2.3.1, we already discussed the Volcano approach (Graefe and McKenna, 1993) for plan generation, which restricts the search space using branch-and-bound pruning.

McKenna et al. (1996) specify a toolkit called *EROC* (Extensible, Reusable Optimization Components) for developing tailor-made variants of Teradata's *NEATO* query optimizer. Instead of exclusively relying on bottom-up or top-down plan generation, they combine both approaches: For implementation variation, they use top-down search, whereas for structural variation, they apply bottom-up search.

In Moerkotte and Neumann (2006), the authors propose a novel algorithm for optimally reordering bushy trees without cross products. If we would only consider SJ reordering in our system, we could implement this algorithm as yet another bottom-up plan generation strategy.

DeHaan and Tompa (2007) introduce a novel class of join enumeration algorithms, which is neither based on System R's optimization approach nor a mere extension of Volcano's top-down enumeration algorithm. Their top-down algorithm relies on branch-and-bound pruning and provides two pruning heuristics that are called *accumulated-cost bounding* and *predicted-cost bounding* (originally introduced by Shapiro et al., 2001). Accumulated-cost bounding assigns a cost budget to the plan generation function. If the cost budget is higher or equal to the estimated cost for a subtree, it is considered optimal. Otherwise, if the budget is exhausted, the top-down traversal stops for the considered recursive step. On the other hand, predicted-cost bounding is a look-ahead approach that tries to determine a lower cost bound of a subtree in the search tree even before actually touching it in a recursive decent.

As we have outlined in Section 5.2, even though most XQGM instances are DAGs, we transform them to trees to simplify query optimization. The approach of Neumann and Moerkotte (2009) discusses the optimization of DAG-structured QEPs. We believe that it can also be used in our system to speed-up plan generation, especially when we start focusing on very large XQuery expressions in the future.

Neumann (2009) proposes a technique for optimizing very large star joins, whose optimization is in general computational intractable, that simplifies queries as long as their optimization is possible within a given period of time. Even though XQuery optimization is more or less restricted to chain queries, which can be optimized more easily, we can apply this idea, if minimal optimization time—rather than minimal execution cost—is our optimization criteria and we are willing to accept sub-optimal QEPs.

Textbooks Besides research papers, we consider two textbooks as very useful for getting a deeper understanding of query processing.

Many ideas developed in this thesis are inspired by Mitschang (1995), which looks at query processing in relational database systems from a research perspective as well as from an engineering perspective.

The comprehensive work of Moerkotte (2009), which is currently under development, focuses on all aspects of query optimization in database systems and claims to become a standard textbook for query optimizer architects. In addition to Mitschang (1995), it covers the research results that haven been contributed since 1995.

9.4 Summary and Conclusions

This chapter concludes Part II. In Section 9.1, we had a look at potential search space sizes and showed that search spaces for XQuery expressions—due to an additional dimension (structural relationships)—can become even larger than search spaces for similar SQL queries.

In Section 9.2, we described two of our plan generation algorithms. At an abstract level, we discussed how the key concepts developed in Chapters 5–8 can be entangled in such a way, that we can derive the most promising QEP—according to the cost model—for a given XQuery expression. First, we had a look at our bottom-up plan generation strategy that is based on dynamic programming. Next, we exemplified top-down plan generation using our implementation of the Simulated Annealing algorithm.

Now, as we have reached the end of Part II, we can dispose of the theoretical background to build a cost-based XQuery optimizer:

- In Chapter 4, we specified logical query rewrite rules that supplement the rules already introduced by Mathis (2009) and allow for pushing up *fn:text()* accesses. This heuristics has a similar effect on query performance as selection push-down in RDBMSs.

- Chapter 5 introduced a neat representation of query plans. Plans are the main data structures, which can be easily derived from XQGM instances, and are manipulated by the query optimizer.

- Query transformations, which are discussed in Chapter 6, are key for generating alternative QEPs. First, we introduced *implementation variation* rules that allow to choose between alternative physical implementations of logical operators. Next, we added another dimension to the search space by formulating *structural variation* rules that allow for rearranging the compositional structure of QEPs by the query optimizer.

- As our approach relies on cost information that help to restrict the search space to promising QEPs, cardinality information are mandatory for cost estimation. Hence, Chapter 7 discussed an XQuery cardinality estimation approach that estimates the cardinalities of tuple sequence items at runtime using abstract domain identifiers.

- Based on functions provided by the cardinality estimation framework, which was derived in Chapter 7, we specified the cost formulæ in Chapter 8 that finally support the query optimizer in calculating the costs of alternative plans and help to identify the plan with the lowest cost amongst all n alternatives.

In Part III, we will discuss the challenges of cost-based query optimization from a software engineering perspective. Moreover, we will demonstrate—using an extensive empirical evaluation—the strengths and weaknesses of our system.

III

Implementation and Empirical Evaluation

10 Optimizer Architecture

"Architecture begins where engineering ends."

(Walter Gropius)

After providing the theoretical background of our cost-based XQuery optimization approach in Part II, we will now look at the optimizer's system architecture.

In Section 10.1, we discuss the different components of our cost-based optimizer. Thereafter, Section 10.2 reviews related work on optimizer architectures. Finally, Section 10.3 summarizes this chapter and leads over to Chapter 11.

10.1 System Architecture

In this section, we will have a closer look at the architecture and implementation aspects of our query optimizer. In Figure 10.1, we provide an overview of the system components. We will focus only on the the query optimization framework itself and do not discuss the surrounding infrastructure, that is, (1) the *Physical Algebra* (PAL), (2) the *XQGM* implementation, (3) the *Execution Engine*—because these components were already discussed comprehensively in Mathis (2009)—, and, finally, (4) the *Statistics* component, which would require a deeper understanding of XTC's implementation being out of the scope of this work.

The *Plan Generator* component provides the glue between the optimization framework and XTC. Consequently, its interface is rather simple: It only provides a method called getBestPlan, that receives an XQGM instance as input and returns one or more QEPs as result—depending on the currently chosen plan enumeration strategy. Beyond that, it allows to influence the overall setup of the query optimizer by selecting the search strategy and the physical operators that shall be considered during plan generation[1].

[1] More precisely, this configuration can be modified on-the-fly using the XTC Universal GUI (see Chapter 11).

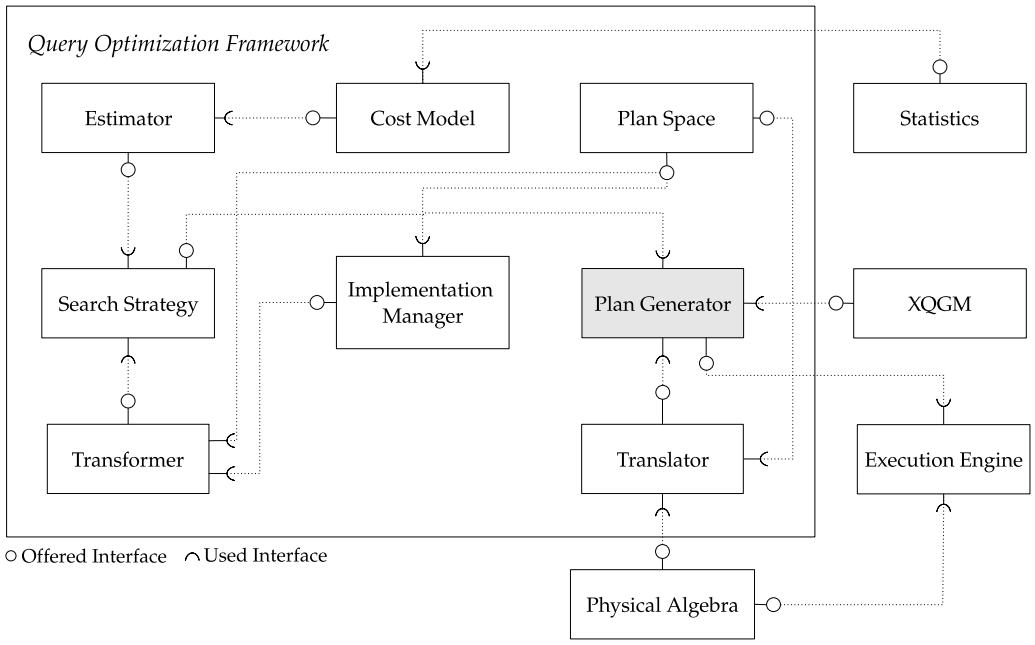


Figure 10.1: Optimizer components (taken from Weiner and Härder, 2010a)

10.1.1 Plan Space

The *Plan Space* component encapsulates the knowledge on the different plan types. It provides a PlanFactory that maps XQGM operators onto Plan classes.

Figure 10.2 shows the common interface of Plan classes[2]. The isGoal method helps to determine whether the current plan was already optimized or not. Using method getImplementation, the optimizer can retrieve the specification of the currently assigned physical operator. When query optimization is finished and plan translation begins, the Translator (see Section 10.1.7) uses the isTranslated method to verify whether the current plan has already been mapped onto a corresponding physical operator. Cost estimates and the output cardinality of the current plan can be derived using getIoCost, getCpuCost, and getpOutputCard, respectively.

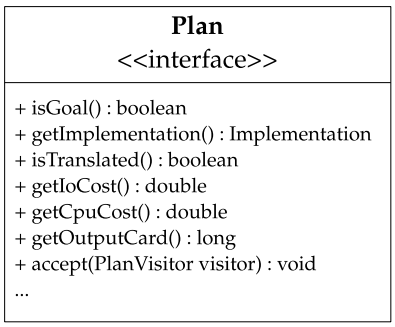


Figure 10.2: Plan interface

[2]Note, for the sake of readability, we only show the most important get methods and omit simple manipulation methods, for example, for adding or removing child plans.

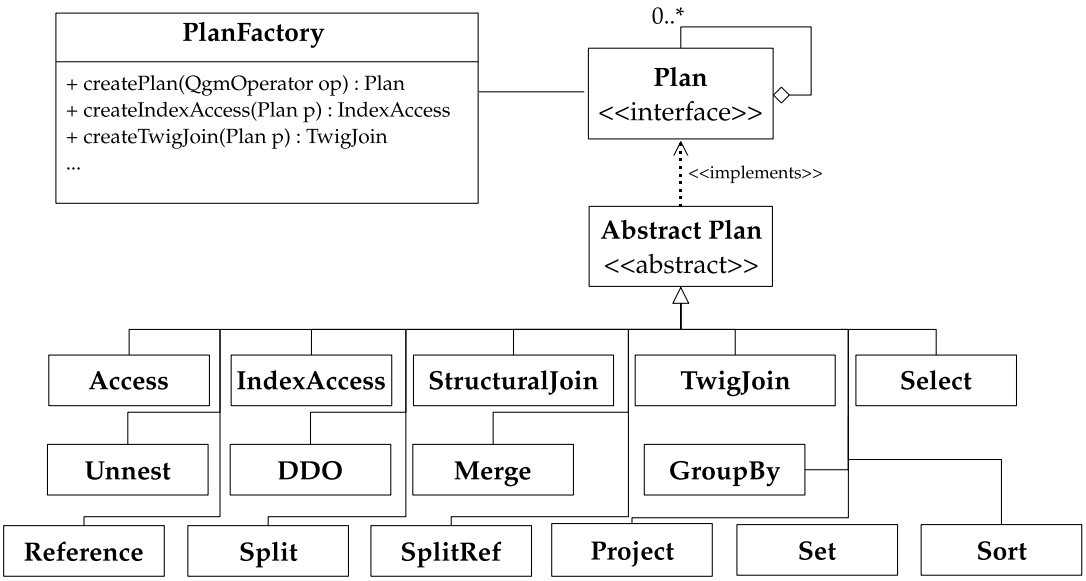


Figure 10.3: Plan Space component

There are some tasks in query optimization that make a full traversal of plans necessary, for example, cost estimation. The *Visitor* design pattern (Gamma et al., 1995) provides a neat recipe for extensible traversal algorithms. Based on this pattern, the accept method allows to register PlanVisitor objects that are notified during plan inspection.

Figure 10.3 depicts the Plan class hierarchy as well as the PlanFactory class. Specific implementations of the Plan interface, for example, a specialized class for handling SJs (StructuralJoin objects), are only created within the Plan Space component. Using the famous *Factory* design pattern (Gamma et al., 1995), the component can expose Plan objects to other components. The abstract subclass AbstractPlan provides basic implementations for common functionality, for example, adding and removing of subplans. The subclasses of AbstractPlan realize the functionality of the corresponding Plan subtypes, which we already introduced in Chapter 5.

In essence, the so-called PlanFactory provides two categories of methods: (1) getPlan methods that return a Plan object for a corresponding XQGM operator (e. g., createPlan) and (2) methods that replace a Plan object with an alternative, for example, a cascade of SJs is replaced by an IndexAccess plan if possible (method createIndexAccess). The first category of methods is only used during the initialization phase to bootstrap the query optimizer. In contrast, methods of the second category are used during plan generation.

10.1.2 Implementation Manager

It is worth noting that, conceptually, Plan objects are completely separated from their actual physical implementation—as already indicated by the getImplementation method. The ImplementationManager encapsulates all knowledge about the currently available physical operator. It is directly connected to XTC's metadata catalog and knows everything about the implemented physical algebra. Externally, the ImplementationManager is exposed using the corresponding ImplementationManagerFactory.

Figure 10.4 shows the interface of the ImplementationManager. Using the getAlternatives method, we can retrieve a list of all valid implementations for a given plan. The ImplementationManager assures that getAlternatives returns exactly those implementations that do not contradict the conditions of the respective implementation variation rules as specified in Section 6.2. To support this important task, each Implementation object describes the metadata of a physical algebra operator. For example, each Implementation object specifies whether the operator requires sorted (duplicate-free) input or provides sorted (duplicate-free) output, respectively. Moreover, the Implementation can tell the Plan Generator how many input streams it needs. Additionally, we can use method isIndexAvailable to verify whether there exists a specific index for a well-defined path.

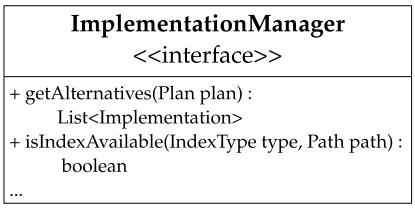


Figure 10.4: Implementation Manager interface

10.1.3 Search Strategy

The *Search Strategy* component is probably the most important component of the query optimizer, because it contains the essential plan generation algorithms.

Figure 10.5 shows the different classes and interfaces of the Search Strategy component. Search strategies are exposed to the Plan Generator using the SearchStrategyFactory. Search strategies—no matter whether they are bottom-up or top-down strategies—have a common interface called SearchStrategy. This interface provides the basic method definitions for search-space exploration. The Plan Generator calls the initialize method for deriving an initial plan for a corresponding XQGM graph. Next, it employs the search method to find an optimal plan. The stop methods informs the concrete

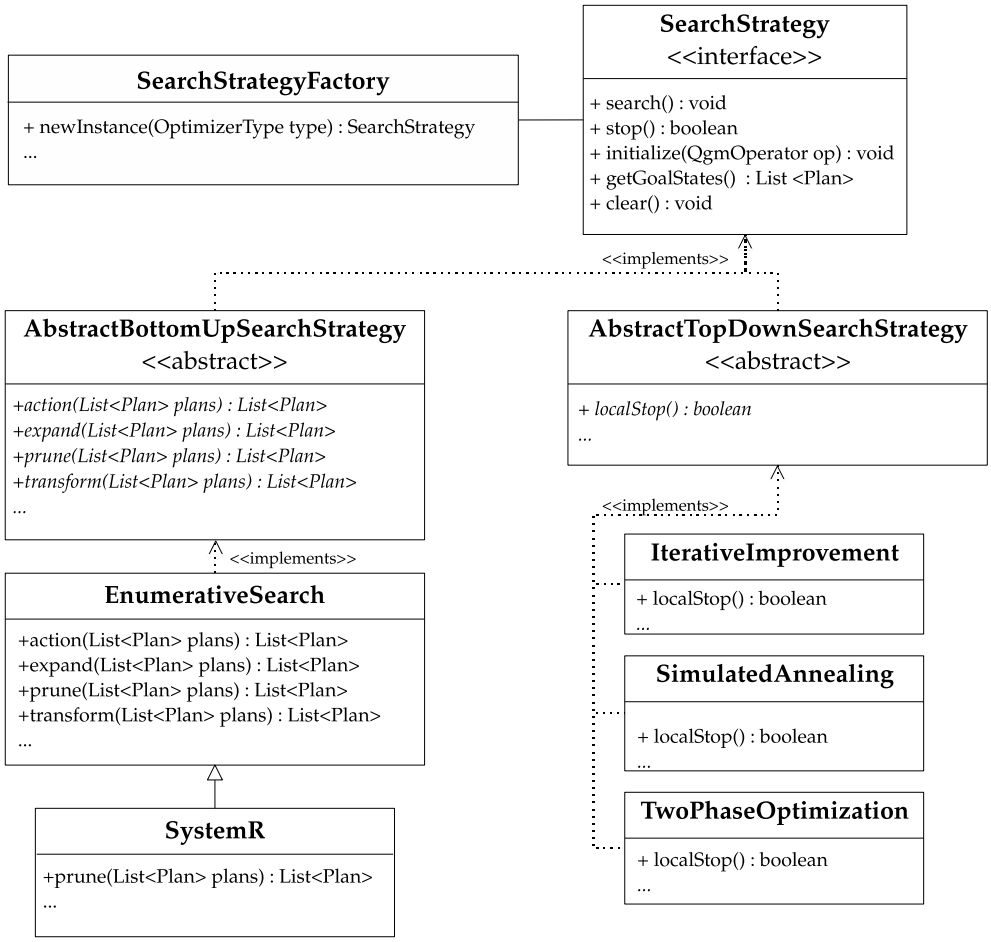


Figure 10.5: Search Strategy component

search strategy whether the termination criteria is met (e. g., no more non-goal plans or local cost minimum). Finally, the optimal plan can be retrieved via the getGoalState method. Bottom-up search strategies inherit from class AbstractBottomUpSearchStrategy several utility methods. Additionally, this class provides abstract method definitions for the implementation of enumerative search algorithms. In class EnumerativeSearch, we realize a generalized version of our exhaustive bottom-up search algorithm (Algorithm 9.1 on page 147), which performs a full enumeration of the search space, that is, the prune function has only a trivial implementation. This search strategy is only used for experiments, because it is only applicable for small search spaces. Nevertheless, it helps to determine each possible plan for a given query and allows

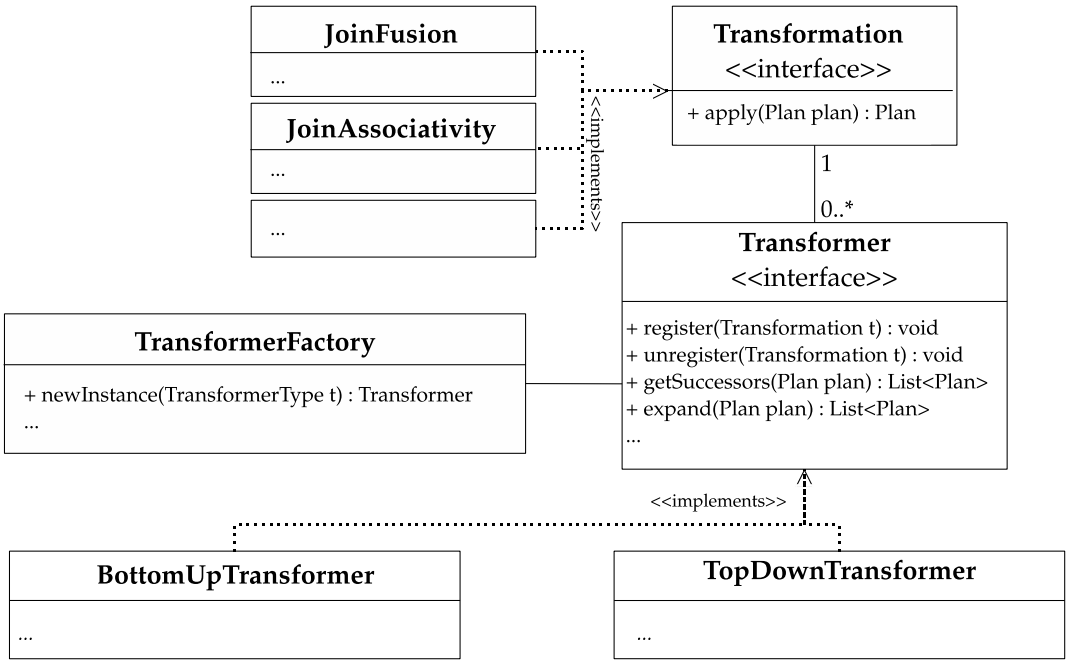


Figure 10.6: Transformer component

to record their respective execution times. As full enumeration is not feasible for medium to large search spaces, its subclass SystemR overrides the prune method in such a way, that only the cheapest alternative of each optimization stage is retained (compare Function 9.2 on page 148).

Implementations of top-down search strategies are based on the abstract class AbstractTopDownSearchStrategy that additionally provides method localStop, which helps to implement local stop conditions. Currently, we implemented three top-down strategies using three subclasses of the abstract base class: IterativeImprovement, SimulatedAnnealing, and TwoPhaseOptimization. Algorithm 9.2 on page 153 already showed our implementation of SimluatedAnnealing's search method.

10.1.4 Transformer

In Chapter 6, we discussed implementation variation rules and structural variation rules that allow to derive semantically equivalent plans. In our system, the *Transformer* component provides the infrastructure for implementing these rules. In Figure 10.6, we illustrate the different classes and interfaces belonging to the Transformer component. Every structural variation rule is

represented as a Transformation object. For example, implementations of the Transformation interface, such as JoinFusion or JoinAssociativity provide an apply method that allows to transform a plan into an equivalent alternative according to the implemented rule.

The TransformerFactory exposes concrete Transformer objects to the plan generator's search strategy. The Transformer interface is implemented by two classes: BottomUpTransformer and TopDownTransformer, whereupon the former one is associated with bottom-up search strategies and the latter one is used by top-down search algorithms. To allow for runtime adaptivity, every transformer does not have a fixed set of transformations. Instead, the framework is open for novel transformation rules that can be switched on (method register) and off (method unregister) using the optimizer configuration tool (see Chapter 11).

In Section 9.2.2 on pages 150–151, we detailed our bottom-up plan generation approach. The BottomUpTransformer implements the expand function (method expand) and the transform function (method getSuccessors) of Function 9.3 and Function 9.4, respectively[3].

In contrast to BottomUpTransformer, TopDownTransformer does not implement the expand method. Instead, implementation variation and structural variation are only applied in the getSuccessors method. Algorithm 9.2 on page 153 shows our implementation of Simulated Annealing. There, the call to function getNeighbor (line 10) internally invokes method getSuccessors of the corresponding TopDownTransformer. Our optimization framework provides two implementations of the getSuccessors method. The first implementation is called *linear top-down* and the second approach is referred to as *random top-down*. Linear top-down transformation works as follows: We choose a complete path from the plan's root node to a leaf node and try to apply implementation variation or structural variation to each node on this path. A successor is found, if a structural variation or an implementation variation was successfully applied. If the leaf node was reached without any modification of the plan, a new path is chosen and the process starts again. The second approach (random top-down) works similarly, except the fact, that it does not take a linear route through the plan, but, instead, it chooses at each operator its future direction randomly.

Now you might ask why we put both types of variations into a single method, instead of handling them separately as done in the BottomUpTransformer. The rationale is very simple: In top-down plan generation, we start

[3] Please recall, function expand applied implementation variation rules and generated all possible combinations of already optimized plans, whereas function transform applied only structural variation rules to the input plan.

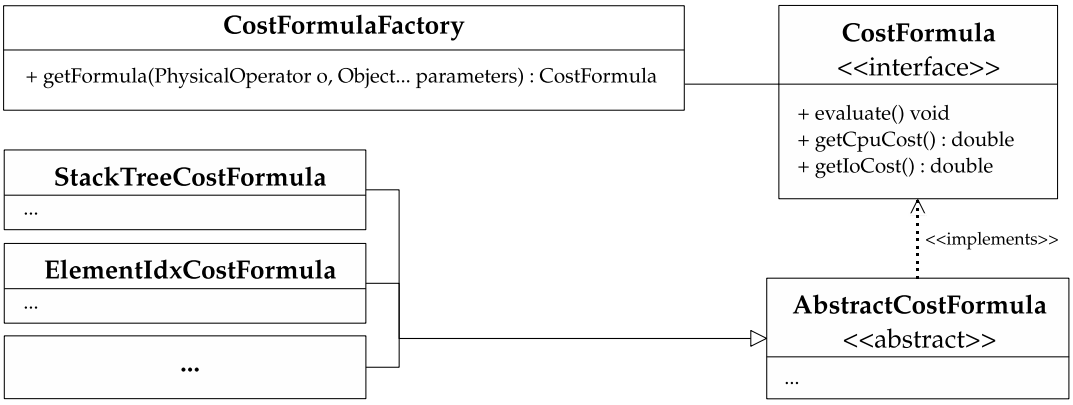


Figure 10.7: Cost Model component

with a complete plan graph and derive an equivalent plan by changing the implementation or the structure, that is, we do not perform both operations at the same time. In contrast, bottom-up plan generation always applies implementation variation first and performs structural variation on all permutations derived in the previous step. Hence, bottom-up plan generation results in a strict separation with respect to rule application, whereas top-down search does not.

10.1.5 Cost Model

The cost formulæ of our cost model are implemented in the *Cost Model* component. Figure 10.7 depicts the corresponding UML diagram. The Estimator component (Section 10.1.6) uses the CostFormulaFactory to get the cost formula for the currently visited physical operator[4]. All cost formulæ share a common interface (CostFormula), where class AbstractCostFormula provides a skeleton implementation of utility functions shared by all specific cost formulæ, for example, StackTreeCostFormula.

10.1.6 Estimator

The *Estimator* component, whose interface and classes are depicted in Figure 10.8, offers an infrastructure for cost and cardinality estimation.

[4]Please note, the Implementation interface (see Figure 10.4) provides a method getImplementation that returns a PhysicalOperator object. This object can be used as a parameter for method getCostFormula to fetch the corresponding formula.

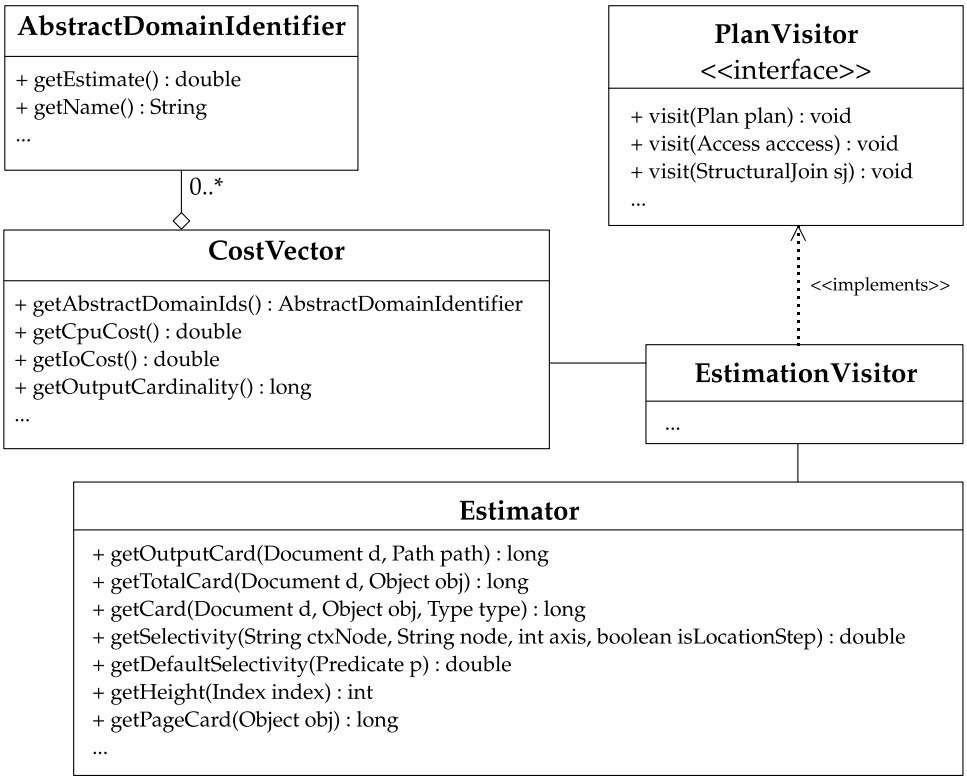


Figure 10.8: Estimator component

For cost estimation, we have to traverse plan graphs. To accomplish this standard task, we rely on an implementation of the classical *Visitor* design pattern (Gamma et al., 1995). The PlanVisitor interface provides a visit method for all Plan classes specified in Section 10.1.1. The actual implementation (EstimationVisitor) allows to inspect a plan graph in left-most depth-first traversal order. A CostVector provides the current estimation context and helps to collect the cost and cardinality information of previously visited plans. Besides cost and cardinality information, the CostVector also manages the current set of AbstractDomainIdentifier objects, which are used for cardinality inference (as already introduced in Chapter 7). All cardinality inference rules are implemented in the visit method of the respective plan type. For example, Cardinality Inference Set 7.1 on page 115 is implemented in method visit(Access access). For cost estimation, which is also done in the visit methods, we use the CostFormulaFactory (Section 10.1.5) to get the cost formula for the currently assigned physical operator.

Cost Model	Estimator
TotalCard(x)	getTotalCard
Card$_x(y)$	getCard
PageCard(x)	getPageCard
Selectivity(p)	getDefault Selectivity
PathCard(p)	getOutputCard
$h(i)$	getHeight

(a)

Cardinality Inference	Estimator
$\|e\|$	getTotalCard
$\sigma(a[\theta\, b])$ $\sigma(a\, \theta\, b)$	getSelectivity
$\sigma(p)$	getDefault Selectivity

(b)

Table 10.1: Mapping of Estimator methods to cost model and cardinality inference rule functions

To bootstrap cost and cardinality estimation, the EstimationVisitor uses the Estimator class which serves as façade for XTC's metadata catalog and its cardinality estimation framework EXsum (see Section 3.1.5 on page 52). Method getOutputCard is used to estimate the output cardinality of complete paths. Using getTotalCard, we can derive the exact number of occurrences of a specific object, for example, an element.

For the cardinality estimation of SJs, we must calculate the selectivity of the structural predicate. This task is done using getSelectivity. For estimating the selectivity of value-based predicates, we use method getDefaultSelectivity, which returns the classical default values well-known from the relational context, for example, for a simple selection predicate, we apply the classical 10% rule[5]. To determine the height of an index, we use method getHeight. For retrieving the total number of data pages consumed by an index, we rely on method getPageCard.

To emphasize the correspondence between the Estimator and the cardinality inference rules on one hand and with the cost formulæ on the other hand, we summarize the mappings between Estimator methods and the parameters of the inference rules as well as the cost formulæ in Table 10.1. Table 10.1(a) shows the relationships between Estimator methods and the cost model functions, whereas we can see in Table 10.1(b), how the expressions used for the formulation of the cardinality inference rules relate to corresponding Estimator methods.

[5] As already mentioned in Section 2.1.3 on page 19, the relational default selectivities can be error-prone and might be improved using histograms. Nevertheless, we have not yet implemented histograms for value-based predicates.

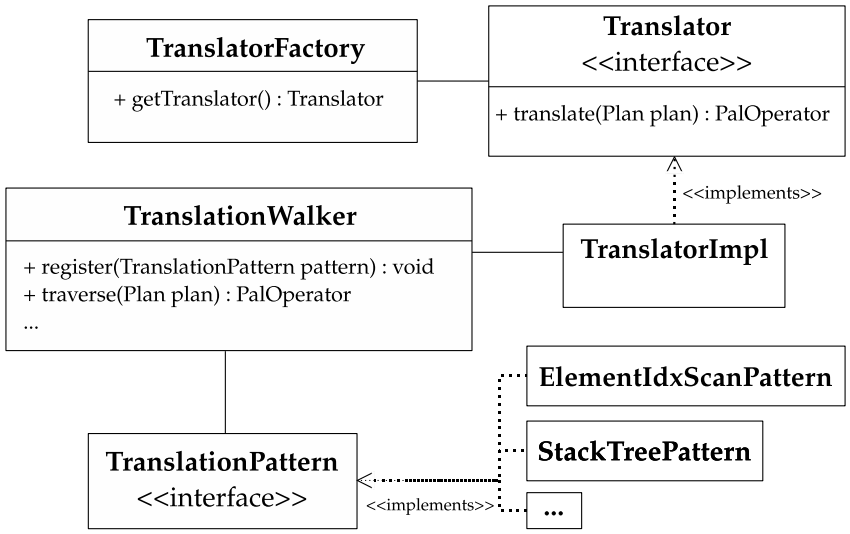


Figure 10.9: Translator component

10.1.7 Translator

Finally, after an optimal QEP is found by the query optimizer, we have to map it onto a corresponding physical plan. For the generation of physical plans, we retain the principles introduced in Mathis (2009): We use a pattern-based mapping approach. For each plan implementation, there exists a TranslationPattern that describes the parameters for instantiating a concrete physical operator. Every TranslationPattern is registered at the TranslationWalker, which completely traverses the plan graph and collects the already translated subgraphs. It uses the *Observer* design pattern (Gamma et al., 1995) to notify TranslationPattern objects, for example, an instance of ElementIdxScanPattern, about the current position in the plan graph. If a pattern matches, the corresponding translation code is executed and the resulting subgraph is collected by the TranslationWalker. Whenever a plan is successfully translated, it is marked as *translated*. To prevent conflicts during matching, we claim that all conditions of TranslationPattern implementations must be disjoint, that is, there may not exist two or more patterns that match for an arbitrary but consistently chosen plan.

10.2 Related Work

Selinger et al. (1979) introduced the fist cost-based query optimizer (see Section 2.1). Our bottom-up search strategy is inspired by System R's seminal plan generation algorithm. Though, our algorithm is not restricted to left-deep query graphs. Instead, our plan generation algorithm is capable of also generating right-deep and bushy query graphs. Nevertheless, to honor this seminal approach and for emphasizing the relationship of our implementation to it, we called our bottom-up plan generation algorithm *SystemR* (see Figure 10.5)[6].

Rosenthal and Reiner (1982) described a query optimization infrastructure for relational and CODASYL-based databases. Though they rely on a different storage mechanism, their overall goal is congruent with ours: evaluating the effectiveness of different query optimization approaches under equal and fair conditions using a single system. They rely on a query optimization technique that incorporates three phases: the first and second phase serve as preparation for cost-based query optimization, which is actually performed in the third step. In the first stage, they create so-called *join templates*, which are alternative join order graphs (similar to search trees formed in System R's optimization methodology). The second phase, which is optional, enriches the join templates with additional knowledge on possible access methods (so-called *direct access structures*). Finally, the third step derives a QEP using cost-based optimization.

Lanzelotte and Valduriez (1991) contributed an extensible framework for query optimization that models the search space without having a concrete search strategy in mind. The architecture of our optimizer implementation is strongly influenced by this paper. Developing an extensible query optimization infrastructure that is open to different types of plan generation strategies is hard, because, for example, bottom-up search strategies and top-down search strategies differ significantly in the way how they traverse the search space and not only in their direction of action (see Section 10.1.3): Bottom-up strategies exhaustively generate all permutations of join orders, whereas top-down strategies transit from one plan to another one by simply modifying the implementation or structure of a single (sub-)graph per optimization step.

Das and Batory (1995) propose a flexible rule specification framework called *Prairie* that extends the rule-based optimizer generator *Volcano* (see Section 2.3.1 on page 30). The way how we specify the implementation variation and structural variation rules are very close to the Prairie approach. Using

[6] A detailed discussion of the various search strategies can be found in Section 2.1.2 on page 14.

Prairie's nomenclature, our implementation variation rules and structural variation rules are more or less equal to their *transformation rules*, whereas our translation patterns correspond to Prairie's *implementation rules*.

Kabra and DeWitt (1999) specify *OPT++* as an object-oriented architecture for extensible query optimization, which extends the work of Lanzelotte and Valduriez (1991). Their approach combines a flexible search component with an extensible logical and physical algebra representation.

Bruno et al. (2009) present a hinting framework called *Phints* that allows to influence the optimizer's choice of the best plan by injecting "external knowledge", for example, constraints that forbid the generation of certain query shapes or always prefer certain implementation variants over equivalent ones—no matter whether the cost estimator considers the first one more or less efficient than the latter one. At the moment, our optimizer does not provide such a hinting mechanism. Though, we could easily integrate it in the future.

10.3 Summary

This chapter looked at the architecture of our query optimization framework to give you a better understanding how the concepts developed in Part II are realized in our system prototype.

First, in Section 10.1, we introduced the optimizer architecture from a bird eye's perspective. Thereafter, we had a glimpse at at the architecture of each individual component. Next, in Section 10.2, we discussed how other authors realized extensible query optimization architectures in the past.

By finishing this chapter, we end the discussion of the query optimization framework. Now, we are familiar with the theoretical background and have a decent impression of the actual implementation of our system. Before we step on to empirically evaluating the query optimization framework in Chapter 12, we will discuss the *XTC Universal GUI* (Chapter 11) that (1) helps to visualize the different outputs of our optimization pipeline (i. e., XQGM instances and QEPs) and (2) allows for on-the-fly configuration of the query optimizer with different search strategies and structural variation rules.

11 Optimization Visualization

"Try out your ideas by visualizing them in action."

(David Seabury)

In this chapter, we introduce *XTC Universal GUI* (XUG) that allows to configure and control our query optimizer at runtime. Moreover, it helps to get a deeper understanding of cost-based XML query optimization by visualizing every stage of the overall query optimization process—that is, from the translation of an XQuery expression into XQGM up until the final QEP.

In Section 11.1, we describe the features of XUG. Thereafter, we summarize this chapter in Section 11.2.

11.1 Visualizing Query Optimization

When we started with the implementation of the query processor, we needed a visual explain tool that helped to debug the complex XQuery-to-XQGM mappings of XTCcmp (see Section 3.2.3 on 57). Notably, we demonstrated the initial version of our visual explain tool at *VLDB* (Mathis et al., 2008) as well as at *BTW* (Weiner et al., 2009).

Even in the first phase of implementation, we immediately became aware that the initial visual explain tool had to be enhanced to support the development of the query optimizer in the same fruitful way as it did before in the context of XTCcmp. Besides that, we performed a technological shift, that is, we reimplemented the initial visual explain tool as an *Eclipse* plug-in[1] and added new features for configuring the query optimizer. Finally, in 2010, we successfully presented our query optimization approach using the novel *XTC Universal GUI* (XUG) at *ICDE* (Weiner et al., 2010).

[1] For more information on Eclipse, visit `http://www.eclipse.org`.

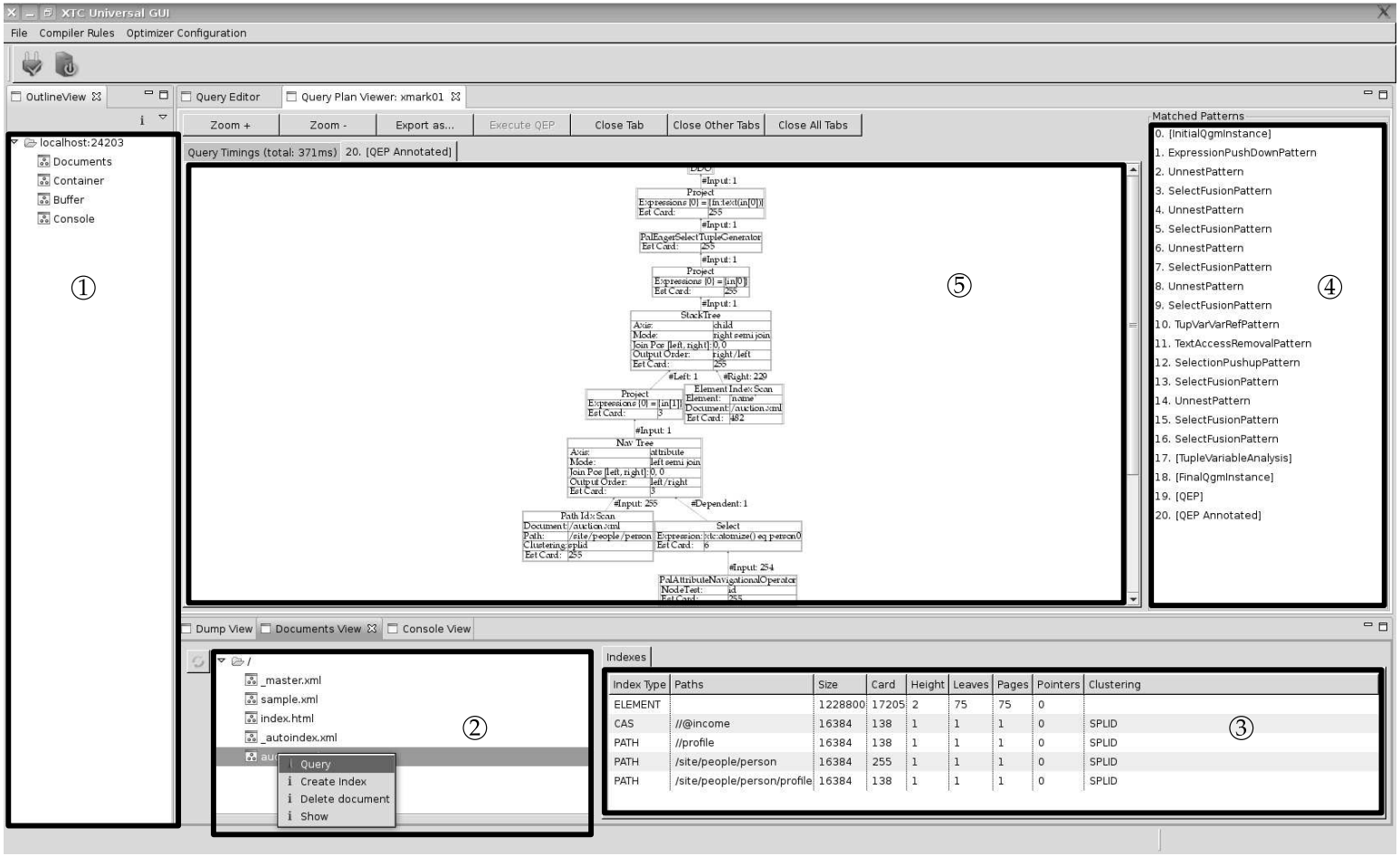


Figure 11.1: XTC Universal GUI

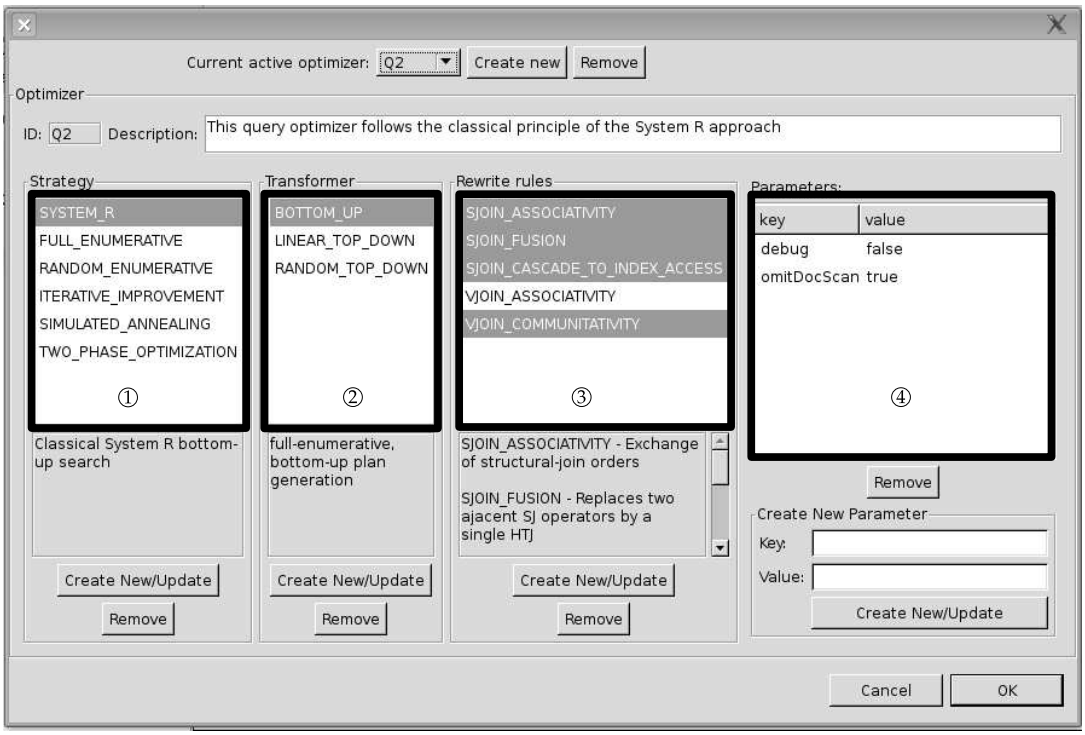


Figure 11.2: XUG configuration dialog

11.1.1 Overview

The name of XUG indicates that our aim was providing a complete control center allowing to interact with XTC in a graphical manner; in addition to the actual optimizer configuration that we are focusing on in this work.

Figure 11.1 shows an overview of XUG. In box ①, you can have a look at the metadata of documents, database containers, and the database buffer. Additionally, you can access the console for interacting with the XTC server in a textual way. Box ② shows the list of all documents currently stored in a document collection. For each document, you can open a dialog for querying the document using an XQuery statement as well as (1) creating index definitions, (2) deleting documents, or (3) simply listing the complete document. Index statistics, that is, (1) the index size, (2) the total number of indexed elements (Card), (3) the height of the index, (4) the total number of leave nodes and inner nodes, and, finally, (5) the clustering type are illustrated in box ③.

The original explain tool of XTCcmp allowed to track the complete query evaluation process from the beginning (translation of an XQuery expression

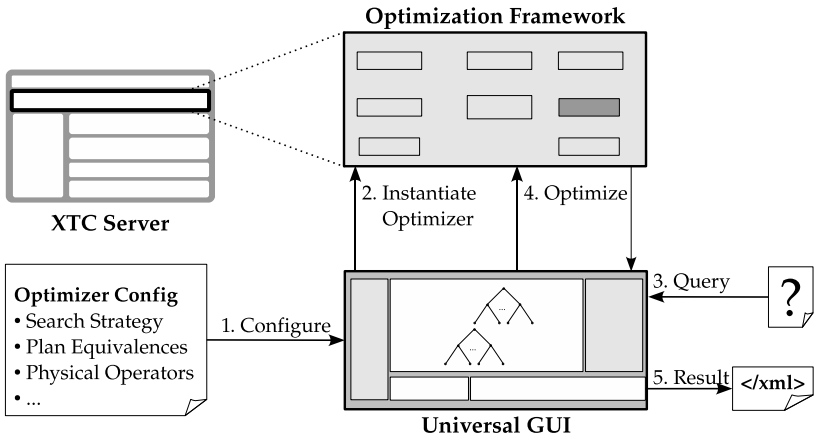


Figure 11.3: XUG in action

into an XQGM instance), over query rewrite (e. g., query unnesting), to its very end (plan of physical algebra operators). In XUG, we retain this feature and create for each modification of an XQGM instance a new graphical representation, which can be selected for visualization in box ④. The main canvas of XUG is represented by box ⑤, where we display either (1) the query result, (2) the query graph previously selected in box ④, or (3) statistics provided by the query optimizer, for example, the estimated IO costs.

11.1.2 Optimizer Configuration

In Figure 11.2 on page 175, we illustrate one of the novel features of XUG: the optimizer configuration dialog. It allows for an on-the-fly reconfiguration of the query optimizer. In box ①, we can select a search strategy, for example, *SystemR*. If there are multiple Transformers that are compatible with the previously chosen search strategy—for example, top-down search strategies can be combined with a linear or a random top-down Transformer (see Section 10.1.4)—, box ② allows to select one of them. In box ③, we can enlarge or restrict the search space by adding or removing structural variation rules, for example, join fusion (in XUG, they are simply called *rewrite rules*). Finally, using box ④, we can assign search-strategy-specific parameters, for example, the temperature reduction factor for Simulated Annealing (see Section 9.2.3 on page 152). XUG's optimizer configuration file is specified in XML. In Appendix A on page 219, you will find a sample configuration file.

Furthermore, we enhanced XUG in such a way that it can output all possible QEPs for a single query. This can be easily reached by choosing *Full Enumerative* as search strategy in box ①; this strategy is basically the same as *System R*, except the fact that it does not prune expensive subtrees[2].

11.1.3 XTC Universal GUI in Action

So far, we have shown some features of XUG, but neglected the interaction of users with it. As XTC and XUG are implemented using the Java programming language, we use the Java RMI API for interaction. Figure 11.3 shows how developers can work with XUG to configure the query optimizer.

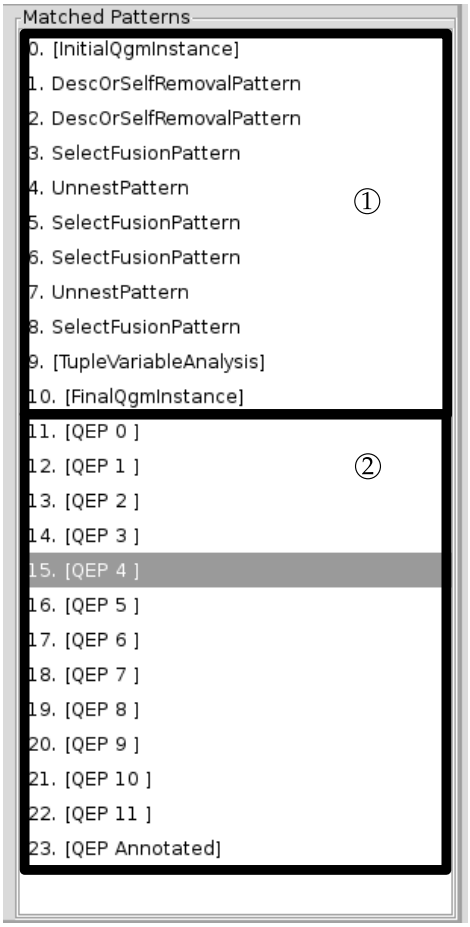


Figure 11.4: Overview of XQGM instances and QEPs available for visualization

In the first step, we can configure a specific query optimizer using the dialog shown in Section 11.1.2. Thereafter, XUG interacts with XTC's query optimization framework and instantiates a new query optimizer (second step). Now, a user can expose a query in XUG and, hence, initiate query optimization. The query optimizer receives the query and processes it according to XTC's query evaluation process (Figure 3.11 on page 57). When optimization is finished, the optimal physical plan is executed and the result is returned to the user. During query optimization, we recorded statistics (e. g., query optimization and execution time as well as estimated costs and cardinalities) and generated textual representations—so-called *dot graphs*—of each modified XQGM graph and of each QEP (one or more—depending on the chosen search strategy). Statistics, dot graphs, and the query result are sent back to XUG. XUG employs the *GraphViz* visualization framework

[2] As we have pointed out in Section 9.1 on page 143, this strategy can only be used for simple queries with few SJs. Nevertheless, it provides a good means for evaluating the quality of the cardinality inference rules and the cost model as well as their influence on effective SJ reordering.

(Ellson et al., 2003) for layouting the dot plans: All plans are translated into *Scalable Vector Graphic* (SVG) instances and are rendered using the *Apache Batik SVG Toolkit*[3]. Additionally, the actually executed QEP is annotated with numerous runtime statistics, for example, the total number of tuples received via a specific access path. Moreover, we generate a pie chart that shows the relative time of different optimization stages (e. g., query optimization time for plan generation) with respect to the total query execution time. Finally, we generate a list of all XQGM instances and QEPs (Figure 11.1, box ④) that can be visualized in XUG's main canvas (Figure 11.1, box ⑤).

Figure 11.4 depicts the plan list for a sample query. In this situation, XUG exposed 10 modifications of the original XQGM during logical query optimization (box ①). Additionally, a full enumeration of the search space resulted in 12 different QEPs (box ②), where the last entry in the list represents the visualization of the actually executed QEP that was annotated with runtime statistics by the the Execution Engine. In the case of Figure 11.4, the main canvas would show the SVG graph of QEP 4.

11.2 Summary

In this chapter, we briefly discussed a helpful tool for developing and evaluating our query optimization framework. During the development of the optimizer, it served as a utility for finding bugs in structural rewrite rules as well as in the different plan generation strategies. Beyond that, it was a precious tool for easily deriving camera-ready illustrations of XQGM instances and QEPs in equal measure (In fact, XUG permits exporting the SVG visualizations into numerous graphics formats).

The subtitle of this thesis mentions three aspects for looking at cost-based query optimization in XML databases: theoretical concepts, implementation, and an empirical evaluation. Until now, we have discussed the first and the second aspect comprehensively. Now, we can move on to the final step, where we will empirically evaluate our concepts and our framework in Chapter 12.

[3]More information on Apache Batik is available at: `http://xmlgraphics.apache.org/batik`

12 Empirical Evaluation

"Doubt the conventional wisdom unless you can verify it with reason and experiment."

(Steve Albini)

So far, this thesis outlined the theoretical background of cost-based XQuery optimization and discussed its integration into the XTC prototype. Now, we are ready to assess the system using a comprehensive empirical evaluation.

In Section 12.1, we describe the experimental setup (hardware, software, and the different query sets) that is used for the empirical evaluation. Section 12.2 looks at the effectiveness of the push-up query rewrite rules as defined in Chapter 4. The reliability of the cardinality inference rules, which were proposed in Chapter 7, is studied in Section 12.3. Thereafter, Section 12.4 verifies the correctness of the cost model. Next, Section 12.5 assesses the quality of our plan generation approach, for example, we perform a scalability test and look at search space sizes. Afterwards, Section 12.6 discusses related work. Finally, Section 12.7 concludes this chapter with a short summary.

12.1 Experimental Setup

To perform all experiments under equal and fair conditions, we rely on a common hardware and software setup and use documents and query sets that are well-known in the XML query optimization community.

12.1.1 Hardware and Software

We conducted all experiments on an Intel XEON quad core (3350) computer (2.66 GHz CPUs, 4 GB of main memory, 500 GB of external memory) running Linux with kernel version 2.6.14. Our native XDBMS server XTC and the query optimization framework were implemented using Java version 1.6.0_07. We started the Java Virtual Machine (JVM) with 512 MB of main memory and we permitted it to consume up to 2 GB. XTC's storage system was initialized with a page size of 16 KB and its buffer was able to hold 256 16-KB frames. If not mentioned otherwise, we used the bottom-up algorithm (Algorithm 9.1 on

page 147) for plan generation and switched on all rules for implementation variation and structural variation that were introduced in Chapter 6.

Every experiment was executed on a cold database buffer, that is, after storing a document and creating PAPs, SAPs, or TAPs, we performed a complete shutdown of the server and restarted it for running the queries. Moreover, before executing a query, we completely emptied the database buffer. The results presented in this section reflect the arithmetic mean of four runs per experiment[1], whereupon we made sure that the standard deviation among these four runs did not exceed 5%.

12.1.2 Query Sets

For the experiments, we rely on a standard set of documents and queries that are widely used in the context of XML query optimizers. First, we use the XMark benchmark (Schmidt et al., 2002) that provides a document generator and a set of 20 simple to complex XQuery expressions (Appendix C.1 on page 223). In our opinion, XMark serves very well to test the effectiveness and the ability of an XQuery processor to derive scalable QEPs. XMark's document generator produces XML documents whose sizes can be specified by a linear scaling factor f. For example, if $f = 1.0$ or $f = 10.0$, then the document generator creates a document that has an approximate size of 110 MB or 1.1 GB, respectively.

In some situations, the conventional XMark benchmark queries are too complex to assess certain aspects of query processing. Therefore, we retain the original XMark documents, but replace the query set by XPath queries (Appendix C.2) provided by the XPathMark-A[2] benchmark.

12.2 Push-Up of Text Accesses

Before we start to assess the cost-based query optimizer, we will have a look at the query rewrite rules introduced in Section 4. There, we defined several rules that help to push-up accesses to the *text()* function as far as possible.

For verifying the effectiveness of the push-up rules, we used Query 4.1, which we already introduced on page 63. We executed the query on an XMark document with $f = 4.0$ and varied the selectivity of the value-based

[1] Actually, we executed five runs. But we did not consider the first run for the calculation of the average, due to skews caused by "warming-up" the JVM that would have tampered the average value.

[2] More information on the benchmark is available at: `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html`

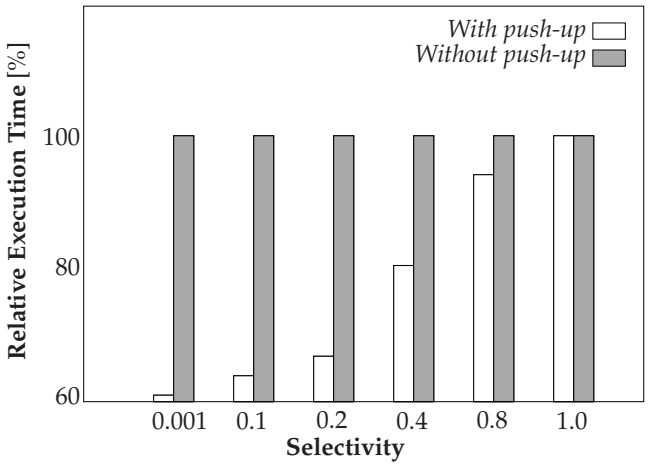


Figure 12.1: Effectiveness of *fn:text()* push-ups

predicate in the range between 0.001 and 1.0 (x-axis). On the y-axis, we put the relative execution time of scenario *with push-up* with respect to the baseline provided by *without push-up*.

Figure 12.1 illustrates the results we recorded for the aforementioned query with respect to multiple selectivities. In the best case (selectivity=0.001), we can reduce the evaluation time by 40%. For a moderate selectivity of 0.2, we still get an reduction of the execution time by 20%. Even if the predicate is not selective at all, the rewritten XQGM is as good as the original one.

12.3 Cardinality Estimation

In Chapter 7, we defined a set of inference rules, which are based on the concept of abstract domain identifiers. These rules are used for cardinality estimation in our optimization framework.

Three questions arise when we try to appraise the quality of the cardinality estimator:

1. Do the rules provide precise estimates for intermediate operators?
2. If there are estimation errors, can the framework recover from them?
3. How close to reality is the estimated cardinality of the complete query?

The first and the second question affect the actual plan generation phase (Section 12.3.1). For example, the bottom-up plan generation algorithm makes a local optimality assumption. If it is violated, for example, due to a cardinality estimation error, consecutive optimization steps may use suboptimal plans. In

contrast, the third question (Section 12.3.2) is almost only important for query translation, as the output cardinality of the query might call for a different materialization strategy.

12.3.1 Cardinality Estimates for Intermediate Operators

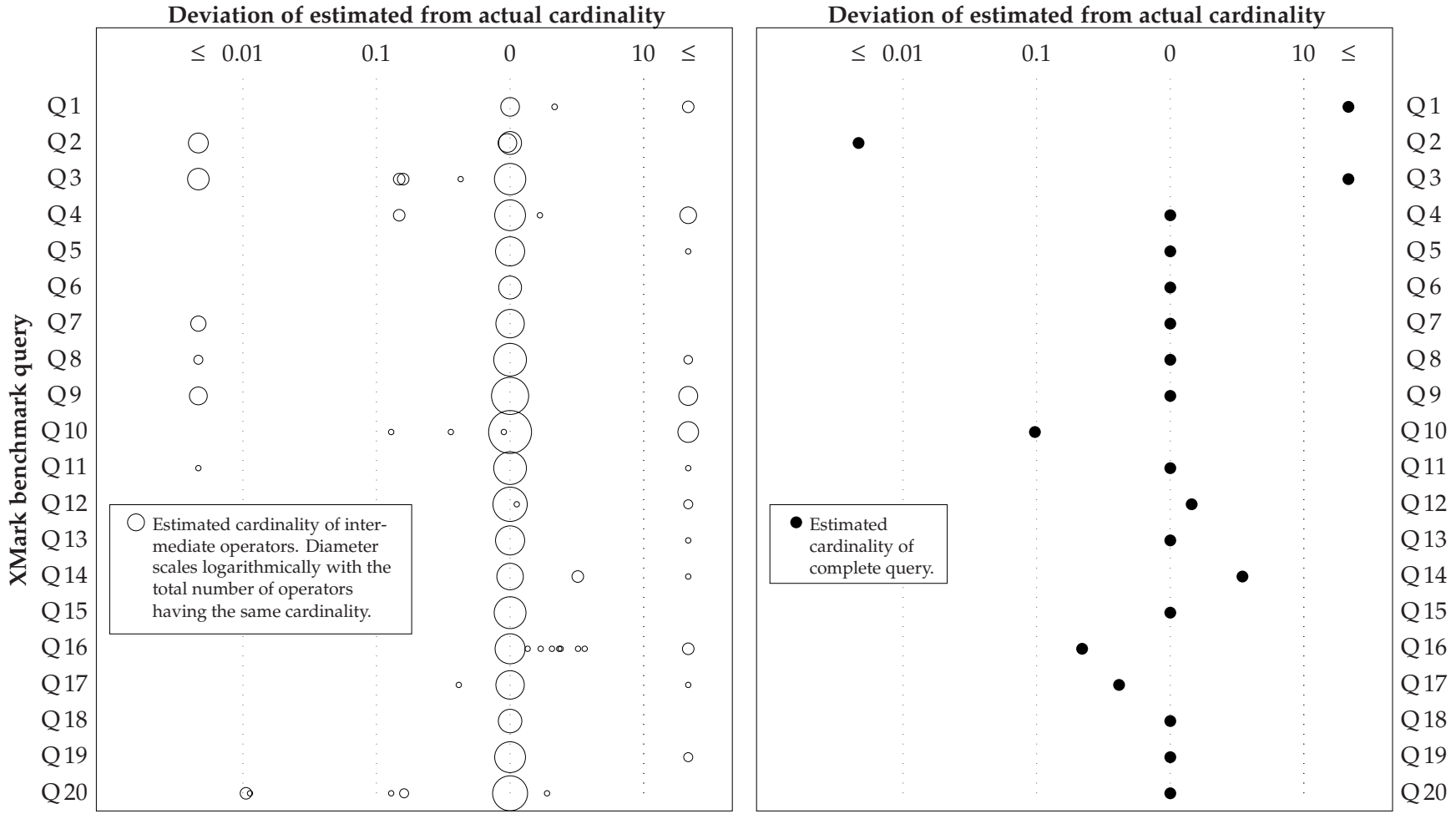
The initial experiment focuses on the first and second question. We executed the XMark benchmark on a document with scaling factor $f = 1.0$. For each query, we recorded the deviation of the estimated cardinality from the actual cardinality of each intermediate operator.

Figure 12.2(a) illustrates the results. The y-axis is labeled with the identifiers of the 20 XMark benchmark queries. On the x-axis, you can see the deviation of the estimated value from the actual value. The interpretation of the scale is: a circle drawn at position 0 means that there was no deviation between the estimated and the actual cardinality at all. In contrast, if a circle is drawn at 10 or 0.1, then we faced a 10-fold overestimate or a 10-fold underestimate, respectively. All circles drawn at the left-hand side of 0.01 and on the right-hand of 10.0 summarize overestimates and underestimates beyond these limits. Some XMark benchmark queries are very complex and contain numerous intermediate operators. Hence, for improving the readability of the results, we scaled the diameters logarithmically with respect to the total number of operators having the same deviation ratio. For example, for the majority of operators in query Q 10, the inference rules estimated the correct output cardinality (here, 88% of the inferred cardinalities were correct). Consequently, the largest circle is drawn at position 0. Additionally, the tiny circles between 0.1 and 0 and the medium-sized circles on the right-hand side of position 10 represent outliers, from which the estimation rules were able to recover successfully.

To sum up, the inference rules provided exact estimates for the majority of queries and associated operators. Though the estimation procedure caused some outliers, their effect on the structure of the QEP is negligible, because the errors mostly affect operators like GroupBy and Unnest that cannot be removed without violating the query semantics and for which we do not provide physical alternatives at the moment.

12.3.2 Output Cardinality Estimation

Besides reliable cardinality estimation for intermediate results, we also want to get decent results for the whole query. In Figure 12.2(b), we depict the actual and the estimated values for the XMark queries' output cardinalities. Once again, we record the deviation of the estimated value from the actual



(a) Actual and estimated cardinalities of intermediate operators

(b) Actual and estimated output cardinalities

Figure 12.2: Results for cardinality inference (Weiner, 2011)

cardinality on the x-axis. The semantics of the circle positions with respect to the x-axis remains the same as before in Section 12.3.1. But there exists a subtle difference: we do not need different diameters anymore, because there is only a single output cardinality for each query. Once again, the y-axis is labeled with the XMark query identifiers.

After clarifying the semantics of the axes, we can now analyze the empirical results. The cardinality estimates for 14 out of 20 queries are close or equal to the actual cardinality gained after executing the query.

We faced a significant deviation between the estimates and the actual values for queries Q 1–Q 3. In the case of Q 1, the predicate selectivity is much lower than expected according to the 10% heuristics. This is not an XML-specific problem and could be easily overcome, if we would use more refined statistics on value distribution, for example, histograms. For queries Q 2 and Q 3, the estimation error, which was unfortunately propagated up to the estimate of the final query result, is due to the incorrect selectivity estimation of positional predicates. Nevertheless, the cardinalities for all performance-critical operators, such as access paths and SJs were estimated correctly. Therefore, the query optimizer's ability to provide scalable QEPs is not hindered in these situations (see Section 12.5).

12.4 Cost Estimation

After evaluating the cardinality inference rules in the previous section, we can now step on to verifying the correctness of the cost model.

First and foremost, we evaluate the cost formulæ for the various access paths in Section 12.4.1. Next, Section 12.4.2 describes how we can derive parameter values for EvaluationCost(p), which helps the plan generator to prefer SJs over HTJs and vice versa. In Section 12.4.3, we look at cost estimation for simple path expressions. Thereafter, we evaluate the quality of cost estimation for value-based path expressions. Finally, we assess cost estimation for more complex XQuery expressions in Section 12.4.5.

12.4.1 Access Paths

In Section 8.2.1 on page 131, we specified the cost formulæ for the numerous access paths. Each cost formula uses a constant factor $PageFetchCost_{IO}$ that represents the cost for loading and transferring a single page from hard disk to main memory. In the first experiment, we performed complete scans over PAPs, SAPs, and TAPs that were created for XMark documents with $f = 2.0$

Access path	Query	f	Est. IO [ms]	Act. IO [ms]	Error [%]
Docum. index	Full scan	2.0	9,225	9,133	+1.007
	Full scan	10.0	46,260	46,451	−0.411
Element index	//`text`	2.0	329.84	320.20	+3.010
	//`listitem`	10.0	828.32	894.80	−7.430
	//`bidder`	10.0	748.96	721.80	+3.762
Path index	Path p_1	10.0	115.15	125.80	−8.465
	Path p_2	10.0	303.80	293.40	+3.544
CAS index	Full, `income`	10.0	303.40	307.60	−1.365
	Point, `income`	10.0	47.68	44.60	+6.095
	Range, `income`	10.0	199.89	219.60	−8.975

Table 12.1: Estimated versus actual IO for access path scans (Weiner and Härder, 2010b)

and $f = 10.0$. We recorded the timings for the complete scans and related them to total number of pages consumed by the indexes. Table A.1 (Appendix A.1) summarizes the average timings per page. Fairly, this is only a rough approximation of the actual page fetch costs that could be looked at a more fine-granular resolution (for example, the costs could be split up into transfer costs to the buffer and costs for reading a single page).

Nevertheless, for our purposes, this approximation is good enough to provide reliable cost estimates for access paths. Even though the cost model treats IO costs and CPU costs separately, our main concern in query optimization is minimizing IO. In our query optimizer, the plan generator only looks at the CPU costs of access paths if their IO costs do not differ. To verify that the cost formulæ are correct and reliable, we carried out a number of experiments. Table 12.1 shows the empirical results we gained for PAPs, SAPs, and TAPs.

We performed full scans over the document index with scaling factors $f = 2.0$ and $f = 10.0$, respectively. For testing the cost estimation of the element index, we scanned three node-reference indexes of varying sizes, which were created for elements *text*, *listitem*, and *bidder*. Moreover, we defined and scanned path indexes p_1 and p_2 (see Appendix C.4.1 for their definitions) on an XMark document with $f = 10.0$.

Finally, we looked at CAS index scans. Besides a full scan over the content of all *income* attribute nodes (p_3 in Appendix C.4.1), we also exposed a point query (income=9,876), and a range query (20,000 ≤ income ≤ 80,000).

Let us consider the simple operation of finding all *text* element nodes. Using a document index scan on an XMark document with $f = 2.0$, we need approximately 9 seconds, whereas using a scan over a corresponding element index, only around 0.3 seconds are necessary. Hence, the second variant is

30 times faster than the first one. This stark disproportion holds in general, because (1) for fetching element nodes from the document index, a complete scan is always necessary and (2) the element index is always much smaller (i. e., consumes fewer pages) than the document index. Consequently, the estimation error for element index scans, which is higher than for document index scans, does not affect the query optimizer in selecting the right access path.

As we have learned, for example, in Chapter 6, path indexes can substitute cascades of SJs that, in turn, use document index or element index scans as access paths. Recall, path index records contain PCR numbers that encode the path from the document root to the leave node. Let us consider path query p_1. Using a path index on p_1, a complete scan would take around 0.3 seconds. For retrieving the same result using SJs that are fed by document index or element index scans, we would need to execute three scans; not to mention the overhead for evaluating each path step using an SJ. Even if we would perform a shared scan over the document index, that is, a scan that collects all *site*, *closed_auctions*, and *closed_auction* elements in a single scan, this would take 46 seconds. In fact, in this situation, an evaluation of p_1 with a corresponding path index scan is more than two orders of magnitude faster than using SJ cascades.

For full CAS index scans, we observed only a slight estimation error (around 1%). We have mentioned before that we use the standard heuristics for predicate selectivity estimation and do not rely on histograms or more refined techniques. By looking at the results for point and range scans over the CAS index, we can see that these heuristics trade estimation accuracy for cost model simplicity. Nevertheless, the errors are still tolerable.

12.4.2 Calibration of EvaluationCost(p)

In the heydays of XML query processing, there was an argument which kind of join operator (SJ or HTJ) would become the first-class citizen for structural predicate evaluation.

Influenced by this debate, Weiner and Härder (2009) compared the performance of StackTree (Al-Khalifa et al., 2002) and TwigOptimal (Bruno et al., 2002) with respect to their relative performance in different selectivity and cardinality scenarios. We introduced a measure called the *relative performance gain* (RPG) that correlates the execution times of both classes of operators for the evaluation of an XPath location step. For example, let us denote the execution times for StackTree and TwigOptimal by t_s and t_t, respectively. Accordingly, the RPG of StackTree is defined as $\text{RPG}_s = t_t/t_s$. If $\text{RPG}_s > 1.0$, then StackTree

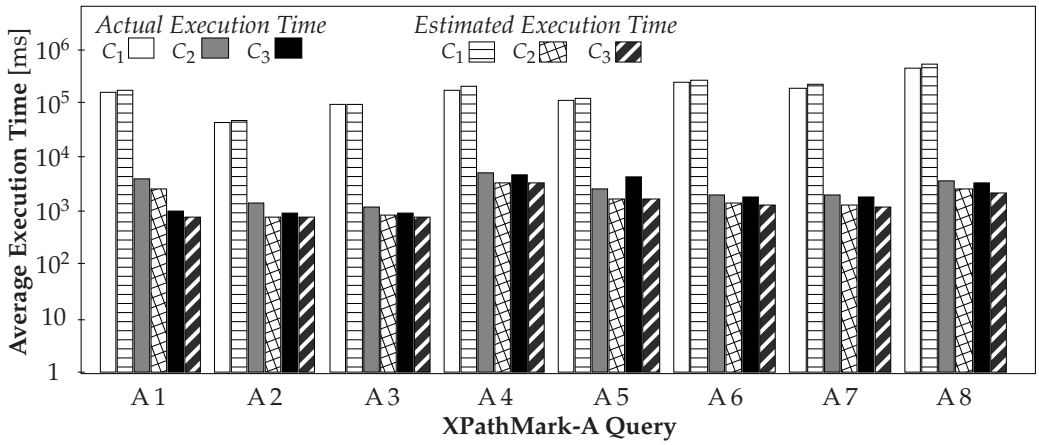


Figure 12.3: Actual vs. estimated execution times on XPathMark-A benchmark

outperforms TwigOptimal and vice versa. Based on this analysis, we derived a set of linear functions that approximate the RPG for StackTree and TwigOptimal. These functions provide a decision criteria for EvaluationCost(p), which we introduced in Section 8.2.2.

Mathis (2009) showed that SJs and HTJs do not differ much in their significance for efficient query evaluation. Moreover, this claim is supported by forthcoming experiments in this chapter. Hence, we do not have a more detailed look at RPG functions in this thesis. Nevertheless, we reperformed the experiments using a novel hardware setup (Section 12.1) that differed from the one in Weiner and Härder (2009) and adjusted the RPG functions accordingly.

12.4.3 Cost Estimation for Path Expressions

In the previous section, we only looked at the prediction quality of cost formulæ for access paths. Now, we will step on to complete queries. Figure 12.3 shows the results for the XPathMark-A queries that were executed on an XMark document with $f = 5.0$. For this experiment, we considered three optimizer configurations: C_1, C_2, and C_3. All configurations were able to apply implementation variation and structural variation (see Chapter 6). In configuration C_1, the optimizer was only permitted to use the document index as access path. Moreover, in configuration C_2, we created an element index as alternative access path. Finally, in configuration C_3, we additionally provided three path indexes (p_1, p_2, and p_4 in Appendix C.4.1).

For each configuration, we recorded the actual execution times (solid boxes) and the estimated execution times (hatched boxes), which we determined

using the IO cost estimates of the cost model. On the x-axis, we depict the identifiers of the eight benchmark queries. In contrast, the y-axis is labeled with the average execution times on a logarithmic scale.

Let us first have a look at the individual configurations: For configuration C_1, the estimated and actual values differ only insignificantly, that is, we observed a minimal overestimation of the execution times. Compared to C_1, we can see a slightly higher difference between estimated and actual values in configurations C_2 and C_3. In both cases, the actual execution times are slightly underestimated by the cost model.

If we perform an inter-configuration analysis for C_1 and C_2, we observe a harsh reduction of the actual execution times. For example, for query A 6, the optimizer reduces the execution time by more than two orders of magnitude. Moreover, we see a strong correlation between decreasing estimated and actual execution times. By comparing C_2 and C_3, we realize that additional path indexes can be beneficial in some situations, for example, for query A 1, where we noticed a further cost reduction. Nevertheless, in general, for C_3, we did not observe such a strong reduction compared to the first two configurations.

12.4.4 Cost Estimation for Value-Based Path Expressions

In Section 12.4.3, we had only a look at simple path expressions. In this section, we will look at XPath queries with value-based predicates. In Appendix C.3 on page 229, you find the definition of queries B 1–B 4, which we will use in this section to verify whether the query optimizer recognizes the ability for exploiting CAS indexes[3]. For this experiment, we reused configurations C_1–C_3 as specified in Section 12.4.3. For configuration C_3, we created the CAS indexes defined in Appendix C.4.2.

Figure 12.4 shows the experimental results for an XMark document with scaling factor $f = 5.0$. Again, on the y-axis, we show the execution times in milliseconds on a logarithmic scale, whereas the x-axis depicts the query identifiers. Bars drawn in solid colors represent the actual execution times, whereas hatched bars illustrate the estimated execution costs.

Obviously, there are gaps between estimated and actual costs in all three configurations. This is mainly due to simplifying assumptions, for example, uniform distribution of values (see also Section 2.1.3). Moreover, for B 3, we can see a huge gap between estimated and actual costs in configuration C_3. Though the optimizer used the corresponding CAS index, which is indicated by an obvious speed-up factor of ~5, the cost did not decrease as estimated.

[3] In Section 3.1.3 on page 45, we already discussed the benefits of CAS indexes for efficiently evaluating value-based predicates.

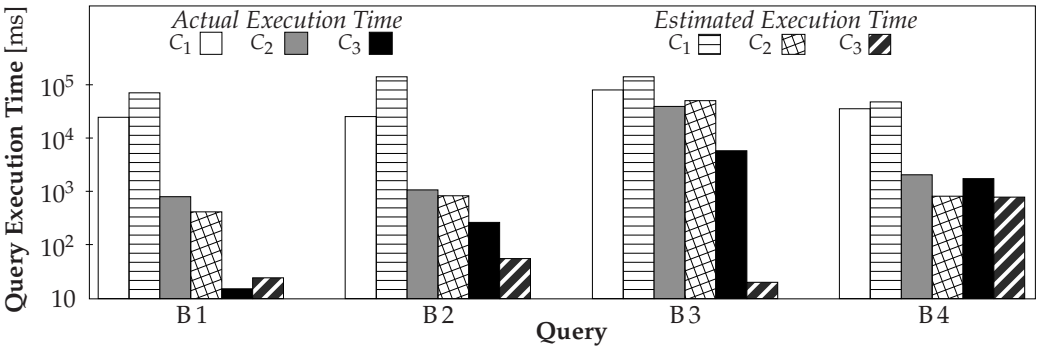


Figure 12.4: Actual vs. estimated execution times on value-based queries

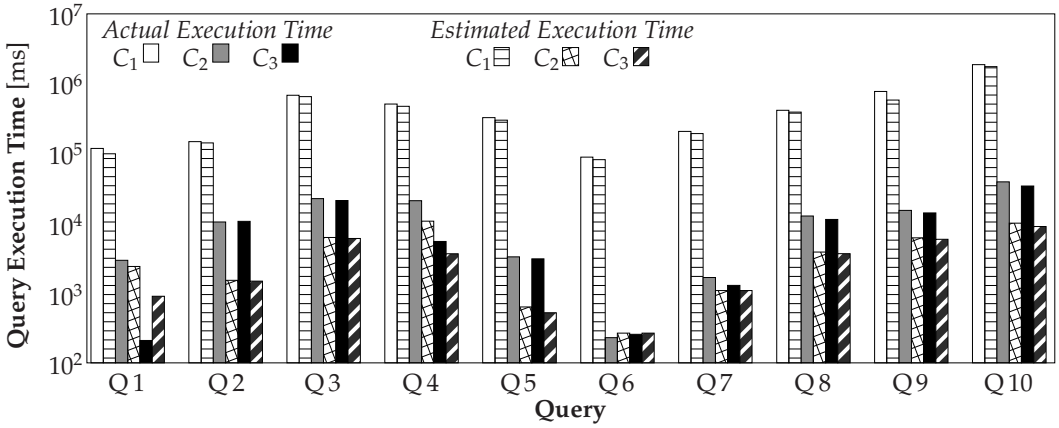
The main reason for this behavior is due to a ParentResolution plan, which causes additional CPU costs, that must be injected by the plan generator to retrieve the parent nodes of *keyword* nodes that actually form the query result (see also Section 5.1).

In configurations C_1 and C_2, all values must be fetched using further accesses to the document index. Only in configuration C_3, the optimizer has the ability to exploit CAS indexes. We observed for all four queries that the optimizer selected the appropriate indexes. This fact is also indicated by the actual execution times for C_3. For example, if we compare the average speed-up between C_1 and C_3, we observed a factor of ~446. Even more impressing, for query B 1, the plan derived in C_3 is more than three orders of magnitude faster than the equivalent plan in C_1. Even for query B 4, we still noticed a speed-up factor of ~1.2 (compared to configuration C_2).

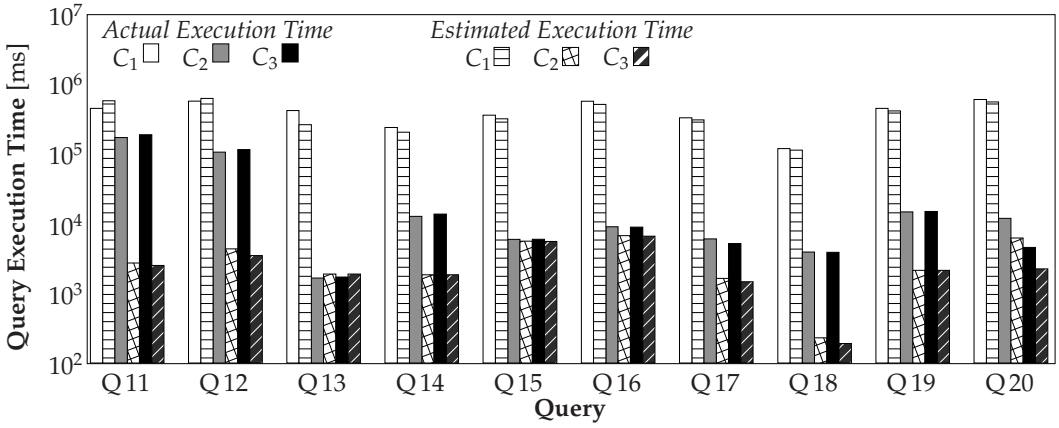
Though the estimates are coarse-grained on the intra-configuration level, they are precise enough on the inter-configuration level, that is, the query optimizer does not miss chances to exploit CAS indexes.

12.4.5 Cost Estimation for XQuery Expressions

Before, we exclusively looked at XPath queries. In this section, we appraise whether the cost estimates are still valid, if we consider XQuery expressions instead of simple XPath queries. For this experiment, we used the XMark benchmark queries and executed them on a document with scaling factor $f = 6.0$ ($\approx 600\,MB$). We reconsidered the three configurations C_1, C_2, and C_3 proposed in Section 12.4.3. For configuration C_3, we used XTC's autonomous index advisor (Schmidt and Härder, 2010) for path index and CAS index creation.



(a) Queries Q1–Q10



(b) Queries Q11–Q20

Figure 12.5: Actual vs. estimated execution times on XMark benchmark

Figure 12.5(a) and Figure 12.5(b) depict the empirical results for queries Q1–Q10 and Q11–Q20, respectively. We scaled the *y*-axis—which shows the average execution time in milliseconds—logarithmically and annotated the *x*-axis with the query identifiers. Again, solid bars show the actual execution times, whereas hatched bars depict the estimated execution times according to the cost model.

By comparing the estimated and actual times for C_1, we observed that the cost model slightly underestimated the actual execution times for all queries. In contrast, by considering configurations C_2 and C_3, the gap between actual and estimated execution times increases. Now, you might wonder about the reason for this gap: Recall, our cost model mainly considers IO costs and takes CPU costs only into account, if two alternatives do not have different IO

costs[4]. In C_1, accessing the document index is very expensive. Hence, almost the whole execution time is spent for IO operations, leading only to tiny gaps between actual and estimated execution times. As the share of IO operations with respect to the total execution time decreases from C_2 to C_3 significantly, the gap between the estimated costs increases accordingly.

In the inter-configuration analysis, we can see that the actual execution times as well as the estimated execution times drop by up to more than two orders of magnitude. For example, for queries Q6 and Q13, we observed speed-up factors of ~380 and ~250, respectively. If we transit from configuration C_2 to C_3, we can still accelerate query processing by more than one order of magnitude. For example, by exploiting a matching CAS index for Q1, the optimizer can derive a QEP with speed-up factor ~14. The reason why the actual execution times improve only slightly is due to: (1) most QEPs involve blocking operators, which cannot be removed (e. g., unnesting and merging), that do not profit from IO reduction by path indexes or CAS indexes and (2) most paths are not very selective. Hence, the overhead for evaluating a path expression using a cascade of SJs is not extremely high.

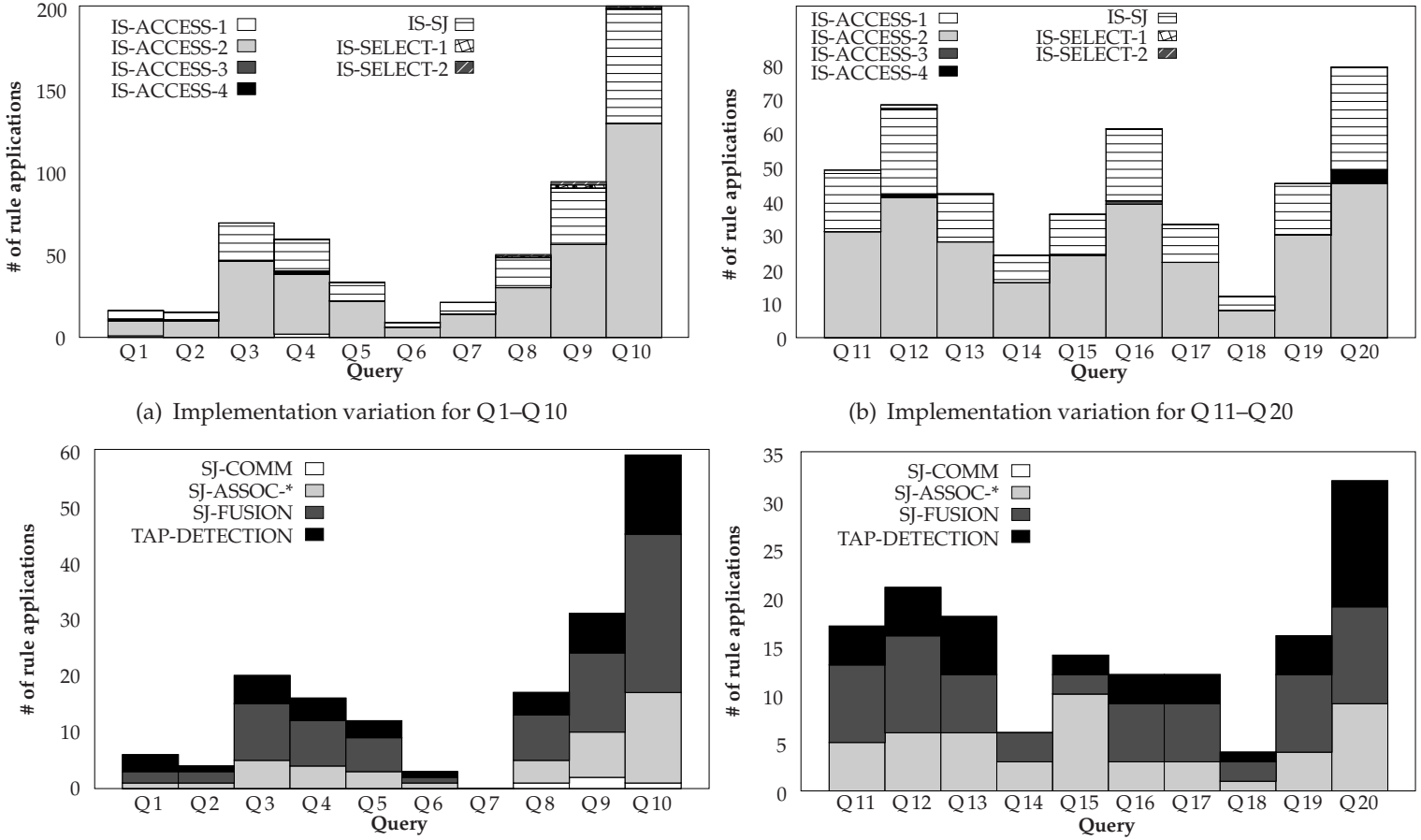
If we juxtapose the actual execution times for C_1 and C_3, we can see that QEPs derived in the latter configuration are, on average, by two orders of magnitude faster than QEPs generated by the optimizer in the former one. By considering the query with the lowest (Q11) and highest (Q1) speed-up, we can see that speed-up factors range between ~3 and ~556, respectively.

As conclusion, a query optimizer, which can only dispose of a document index, is not able to keep up with relational XQuery processors (e. g., MonetDB/XQuery that was introduced in Section 2.3.2). In fact, scalable QEPs can only be derived in configuration C_3—which makes excessively use of the element index and can additionally profit from tailored path indexes and CAS indexes—as indicated by further experiments in Section 12.5.2.

12.5 Plan Generation

After looking at cardinality and cost estimation in Section 12.3 and Section 12.4, respectively, we will now focus on aspects of plan generation.
In Section 12.5.1, we analyze how often the various transformation rules are used. Moreover, we will derive a minimal transformation rule set for the XMark workload. Based on the empirical results gained so far, we perform

[4] By mainly considering IO costs for cost estimation, we do not risk to omit an optimal plan during query optimization. Most operators have only a single implementation, for example, grouping and unnesting, and cannot be removed. Only for value-based join and SJ reordering, the CPU costs are important and must be considered by the optimizer to prevent bad join orders.



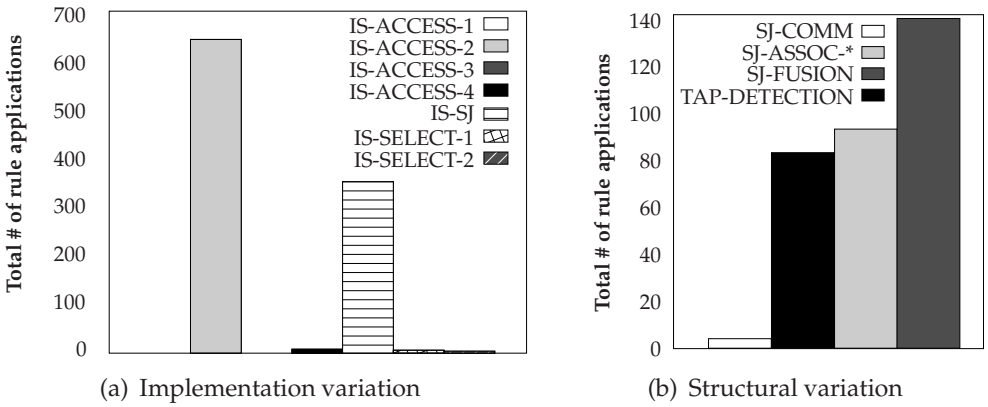
(a) Implementation variation for Q1–Q10

(b) Implementation variation for Q11–Q20

(c) Structural variation for Q1–Q10

(d) Structural variation for Q11–Q20

Figure 12.6: Application of transformation rules



(a) Implementation variation (b) Structural variation

Figure 12.7: Transformation rules on XMark

a scalability test in Section 12.5.2. Next, we look at the runtime overhead raised by query optimization in Section 12.5.3. In Section 12.5.4, we compare the four plan generation strategies currently implemented in our framework with respect to optimization time and QEP quality. Thereafter, Section 12.5.5 empirically analyzes and discusses the complexity of bottom-up plan generation. Finally, we rank and analyze the estimated costs and actual costs of the top five plans in Section 12.5.6.

12.5.1 Minimal Set of Transformation Rules

In the initial plan generation experiment, we assess the usage of the implementation variation and structural variation rules. The goal of this analysis is to identify a minimal set of transformation rules—a so-called *essential set* (see Section 12.6)—that is absolutely necessary for effective query optimization, that is, it contains only those rules that still allow to find an optimal plan.

For this experiment, we use an XMark document with scaling factor $f = 1.0$ and created all indexes specified in Appendix C.4.3 on page 230.

Analysis of XMark Workload

Now, we will look at each query of the XMark workload and identify the rules that strongly influence the size of the search space.

First, we will look at the implementation variation rules. Figure 12.6(a) and Figure 12.6(b) show the results for queries Q 1–Q 10 and Q 11–Q 20, respectively. For every query, rule IS-ACCESS-2 (as defined in Section 6.2)

		S_1	S_2	S_3
Implementation variation rules	IS-ACCESS-2	✓	✓	✓
	IS-SJ	✓	✓	✓
Structural variation rules	SJ-FUSION-*	✓		
	SJ-ASSOC-*	✓	✓	

Table 12.2: Transformation rule sets

is applied[5]. Moreover, in almost all cases, rule IS-SJ (Section 6.2) is used as well[6]. In Figure 12.7(a), we aggregated the application of implementation variation rules for the whole XMark workload. Rules IS-ACCESS-2 and IS-SJ are obviously dominating; the remaining rules are rarely used—if at all.

Figures 12.6(c) and 12.6(d) illustrate the results for the transformation rules. In contrast to implementation rules, where some rules have not been used at all, all transformation rules are applied—at least for some queries. To improve readability, we aggregated the results for TAP-1 and TAP-2 (see Section 6.3.3) as *TAP-DETECTION*, the results for the various join associativity rules (Section 6.3.2) as *SJ-ASSOC* as well as the results for the join fusion rules SJ-FUSION-1 and SJ-FUSION-2 (compare Section 6.3.3) as *JOIN-FUSION*. These results are dominating structural variation rule application. Figure 12.7(b) shows the aggregated results for the XMark workload. For XMark, join fusion is the most applied structural variation rule.

Towards a Minimal Set of Transformation Rules

Now, we identify which rules must be part of the essential set for XMark.

We already saw in Section 12.4.5—where we observed a dramatic performance increase after switching from configuration C_1 to C_2—that rule IS-ACCESS-2 must be in the essential set, otherwise element index scans are not used by the optimizer. Moreover, rules TAP-1 and TAP-2 are also budding candidates for the essential set, because we observed in Figure 12.5 further performance increase if the optimizer was able to exploit TAPs, for example, compare the results for configurations C_2 and C_3 for query Q20 in Figure 12.5(b).

For the remaining transformation rules—for which we are currently not sure whether they have an impact on query optimization performance—, we will evaluate whether we can switch them off without observing a dramatic performance decrease for the workload. For the sake of simplicity, we do not

[5]Please recall, this rule tries to replace a document index scan by an element index scan.
[6]Note, this rule changes the implementation of an SJ.

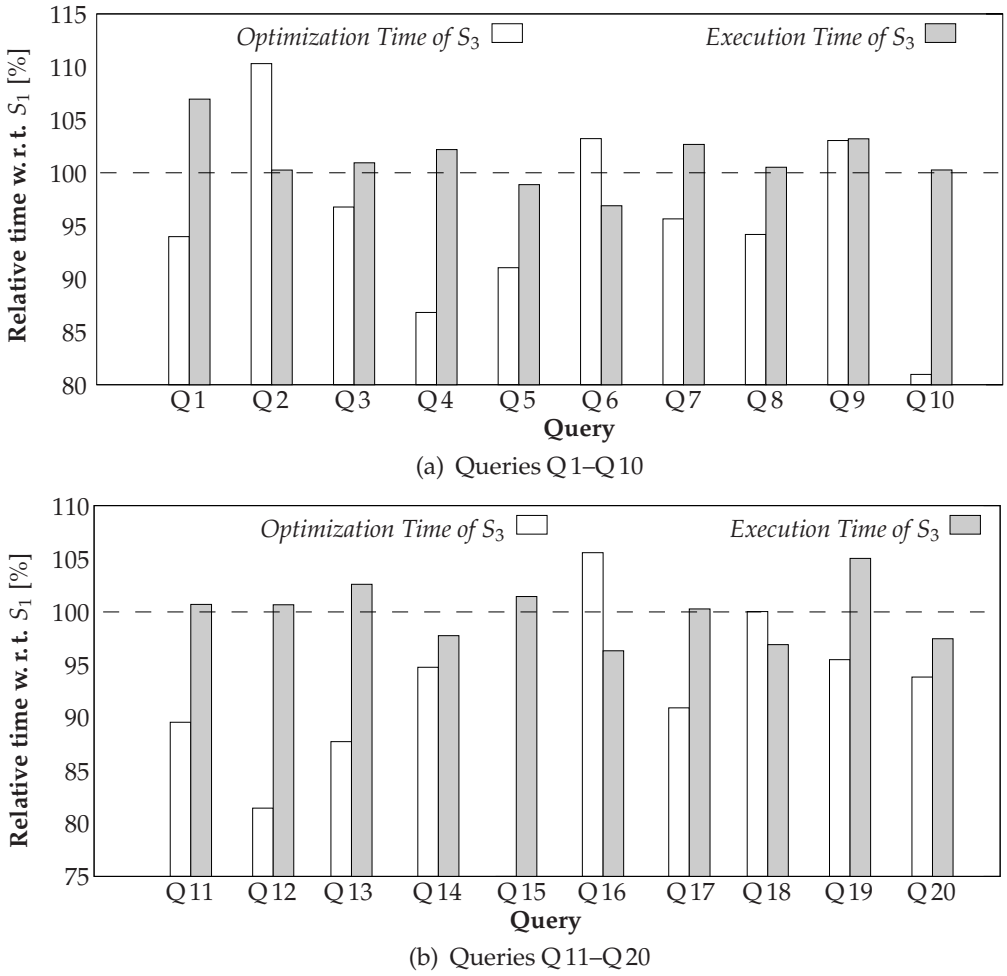


Figure 12.8: Comparison of optimization time and execution for S_1 and S_3

consider rule IS-SJ, but only apply rules J-ASSOC-*, SJ-FUSION-1, and SJ-FUSION-2. For this experiment, we once again rely on an XMark document with scaling factor $f = 1.0$. Furthermore, we only created an element index as additional access path[7].

Table 12.2 on 194 depicts three transformation rule sets[8] S_1–S_3. In S_1, we did not switch off any structural variation rule. In S_2, we turned off SJ-FUSION-1 and SJ-FUSION-2. Finally, we also deactivated the join associativity rules (SJ-ASSOC-*) in S_3.

[7] In this situation, we do not consider TAP detection, because appropriate path or CAS indexes would complicate the analysis.

[8] Rule IS-SJ is not switched on and off, because we observed in a previous experiment that this rule has no influence on optimization time.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Impl. variation rules	IS-ACCESS-2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	IS-SJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struct. variation rules	SJ-FUSION-*		✓	✓		(✓)	✓	(✓)	(✓)	✓	
	SJ-ASSOC-*		✓	✓		(✓)	✓	(✓)	(✓)	✓	
	TAP-*	✓			✓			✓			

		Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
Impl. variation rules	IS-ACCESS-2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	IS-SJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struct. variation rules	SJ-FUSION-*	(✓)		(✓)	✓		✓	✓	✓	(✓)	(✓)
	SJ-ASSOC-*	(✓)		(✓)	✓		✓	✓	✓	(✓)	(✓)
	TAP-*										✓

Table 12.3: Tailor-made minimal transformation rule sets for XMark queries

After performing the experiments, we observed only minimal differences between the results of S_2 and S_3, that is, in general, S_2 shows the same correlations between optimization time and execution time as observed for S_3. Therefore, we will only look at the differences between S_1 and S_3. In Figure 12.8, we depict the results for average optimization time and the average query execution time on the y-axis. Figures 12.8(a) and 12.8(b) illustrate the results for queries Q1–Q10 and Q11–Q20, respectively. The results depict the relative increase or decrease for both properties with respect to the results gained using set S_1. For example, for query Q1, by preferring set S_3 over S_1, we can decrease the optimization time by around 5% and will observe a more or less equal share of performance decrease.

Let us assume that our optimization goal is less optimization time while not observing a dramatic performance decrease. If we look at Figure 12.8, we see that we will need J-FUSION-* and J-ASSOC-* for queries Q2, Q6, Q9, Q16, and Q18. Otherwise, optimization time may even increase.

We must use set S_3, that is, turn off both structural variation rules, for queries Q4, Q10, Q12, and Q15, because a dramatic decrease in optimization time is observed while performance decreases only moderately. Finally, we can use S_3 for queries Q5, Q7, Q8, Q11, Q13, Q19, and Q20. But, by doing so, we will not observe a remarkable decrease in optimization time.

Tailor-Made Minimal Transformation Rule Sets for XMark Queries

Apparently, there is no minimal set of transformation rules for the whole XMark workload. Therefore, we will now aggregate the results found so far and assemble a tailor-made set of transformation rules for each individual XMark query. Table 12.3 shows the minimal set for each query. Whenever we use symbol "✓", this rule *must* be turned on, otherwise the performance decrease is much higher than the decrease in optimization time. On the other hand, "(✓)" indicates that this rule *can* be switched off.

As we have already found out in Section 12.4.5, even though TAPs are exploited for almost all XMark queries, their usage does not lead to a noticeable performance increase compared to SAP-optimized plans. Therefore, we recommend to use TAP detection (TAP-*) only for queries Q1, Q4, Q7, and Q20, where their application positively affects query execution time.

12.5.2 Scalability

In this section, we test the scalability of the query optimizer on varying document sizes and simple to complex XQuery expressions.

Figure 12.9 shows the performance results of the QEPs produced by the query optimizer for the 20 XMark benchmark queries[9]. We created an element index on each document and used XTC's index recommendation tool (Schmidt and Härder, 2010) for creating additional path and CAS indexes. To get rid of the index-creation overhead, all measurements were performed after all relevant indexes were created.

For this experiment, we scaled the document size from 110 KB to 1.1 GB and recorded the execution times of the optimal QEPs according to the cost model. On the x-axis, you can see the different XMark document sizes, for example, the document with size 1.1 GB was created using XMark scaling factor $f = 10.0$. The y-axis shows the average execution times on a logarithmic scale. We do not show all results in a single diagram, due to readability reasons. Instead, we equally distributed the results on four illustrations. Each of them depicts the results of five queries. To give you an impression how optimal linear scale-up would look like, we draw dashed lines in each Figure 12.9(a)–12.9(d) for comparison.

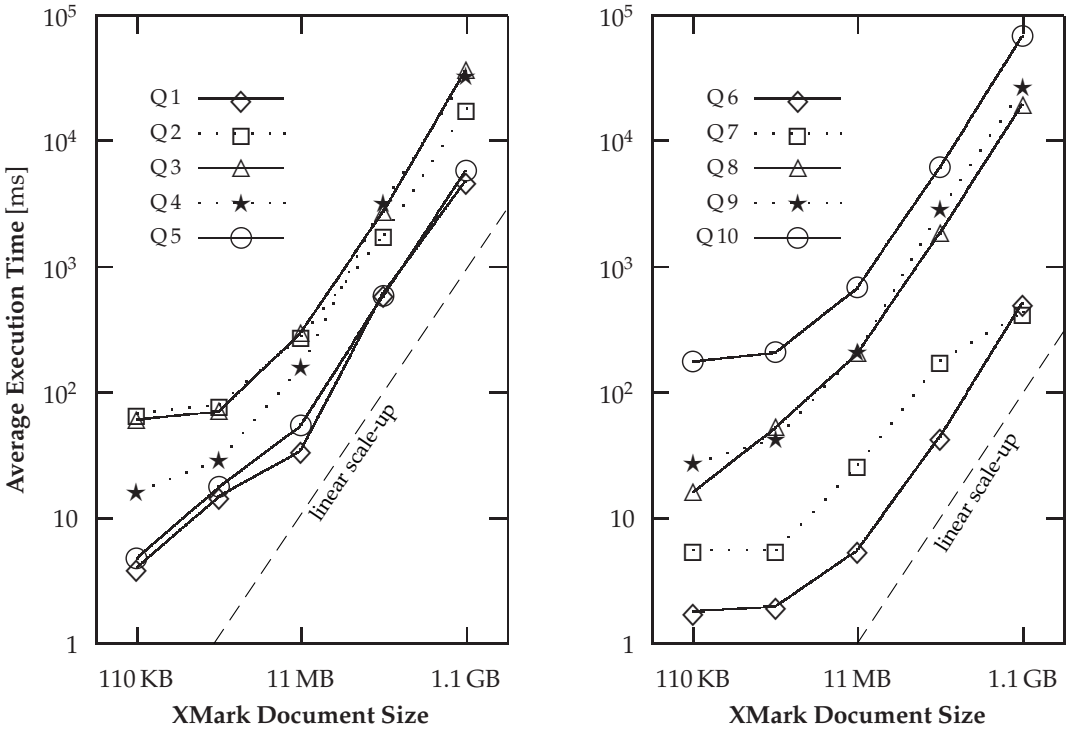
In Figure 12.9(a), we can see the results for the first five queries. On average, we get a scale-up of ~7, that is, by increasing the document size by an order of magnitude, query execution on the larger document takes only ~7 times longer than on the smaller one.

For queries Q 6–Q 10, we observed an average scale-up of only 5.97. Though this sounds promising, the extremely good scale-ups for 100 KB and 1.1 MB documents are distorting the actual results. Nevertheless, even for the largest two documents in the experiment, we get scaling factors of ~9, respectively.

The third query set, whose results are illustrated in Figure 12.9(c), falls victim of some irregularities. If we look at the results for queries Q 11 and Q 12, we notice linear scale-up only until a document size of 11 MB. For larger document sizes, scale-ups range in the twenties and seventies, respectively. Now, you might expect an error of the query optimizer, for example, a plan considered optimal that finally turned out to be worse. Luckily, the reason for the bad scale-up of these queries is not an error of the query optimizer. Instead, both queries inhabit value-based joins whose selectivities are rather high. Moreover, the queries produce intermediate results that scale quadratically with the document size[10]. Even though we cannot reach linear scalability for these queries, the results show subquadratical scale-up. For the remaining

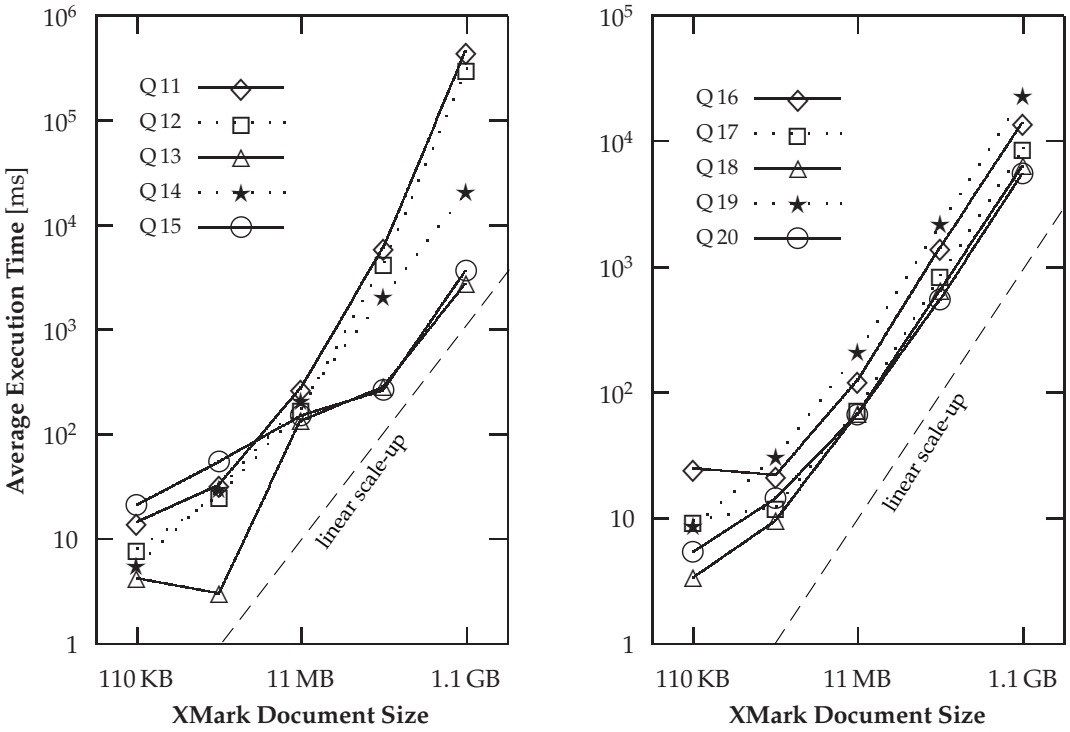
[9] The results reported in this section are based on the complete set of transformation rules. If we would use the minimal sets defined in Section 12.5.1, we expect to see even slightly better speed-up behavior.

[10] Compare also the results for MonetDB/XQuery: `http://monetdb.cwi.nl/XQuery/Benchmark/index.html`.



(a) Results for Q1–Q5

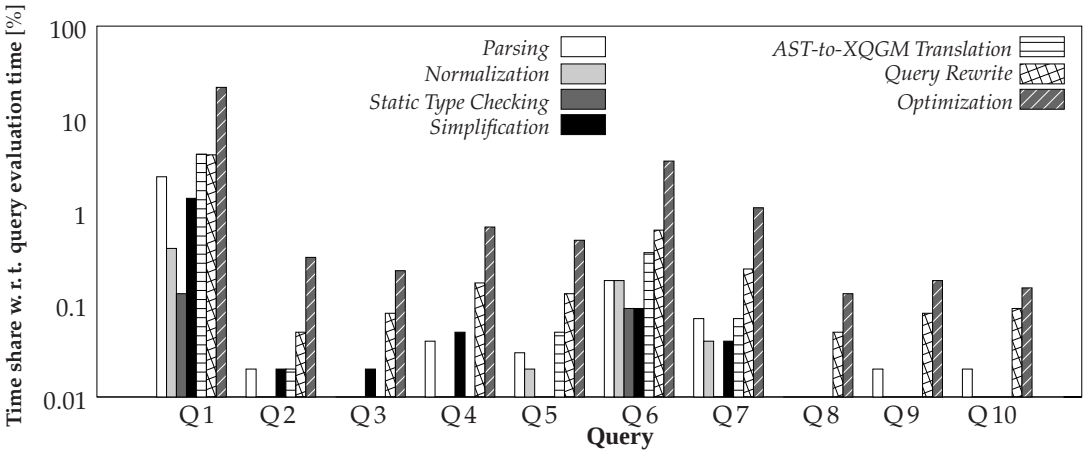
(b) Results for Q6–Q10



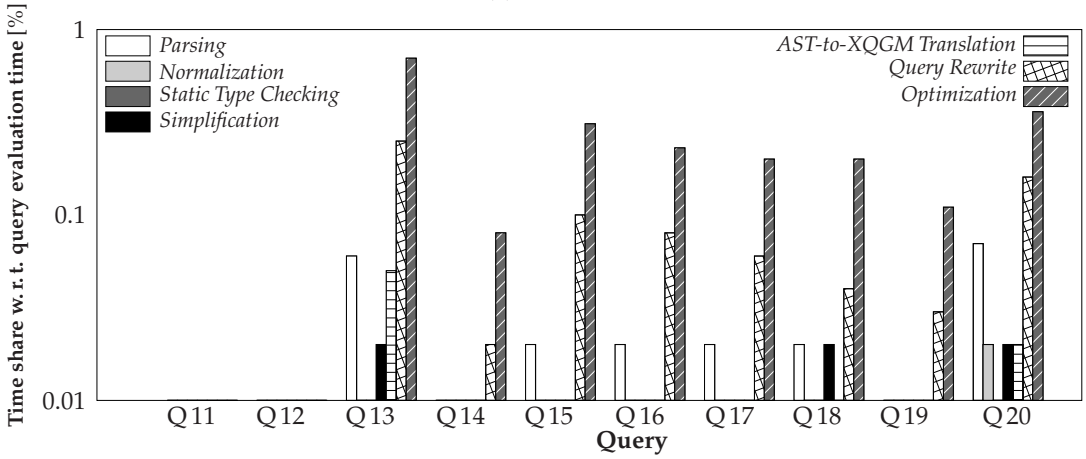
(c) Results for Q11–Q15

(d) Results for Q16–Q20

Figure 12.9: Scalability of QEPs on XMark benchmark



(a) Queries Q1–Q10



(b) Queries Q11–Q20

Figure 12.10: Query evaluation overhead

queries Q13–Q15, we once again get, on average, sublinear scale-up (factor 9.3).

Finally, Figure 12.9(d) depicts the experimental results for the last five queries. Here, on average, the QEPs' execution times sublinearly scale with factor ~7 with respect to the document size.

After analyzing the results, we must state that the query optimizer is able to produce scalable QEPs for the simple to complex XQuery expressions specified by the XMark benchmark. In fact, we reach an average scale-up of 9.2 for all queries on the considered document sizes. Note, these results can only be reached if the query optimizer can use PAPs and TAPs. If there is only a document index, linear scale-up is impossible for document sizes beyond 110 MB.

12.5.3 A Look at Query Optimization Overhead

Honestly, query optimization in general is a costly task, because different plans must be derived and compared regarding their costs. Now, we can ask about the price that we have to pay for scalable XQuery plans. To answer this question, we look at the costs of every stage in XTC's query evaluation process (see Figure 3.11 on page 57). For deriving the costs, we used the XMark benchmark workload and executed it on a document with scaling factor $f = 6.0$.

Obviously, query execution takes the lion's share of the overall query evaluation time, that is, the time spent from parsing the query until finally retrieving the query result. For the sake of readability, we omit the results for query execution in Figures 12.10(a) and 12.10(b). On a logarithmically-scaled y-axis, both figures illustrate the relative time share with respect to the overall query evaluation time spent for each query evaluation stage—except for query execution. Query optimization may exceed a margin of 10% (e. g., query Q 1). For queries Q 11 and Q 12, query execution is in such a manner costly that all remaining stages call for less than 0.01% of the evaluation time. For the vast majority of queries, query optimization spends not more than 1% of the overall query evaluation time (on average, it consumes only around 1.56% of the overall query processing time for the complete workload). Hence, query optimization results only in a minimal overhead. Even more important, it allows for reducing query execution times by two orders of magnitude (compare Section 12.4.5).

12.5.4 Bottom-Up versus Top-Down Plan Generation

In previous sections, we exclusively relied on the bottom-up algorithm (Section 9.2.2) as default search strategy for plan generation. When we started with the implementation of the query optimizer, we assumed that exhaustive plan generation might be too costly for complex XQuery expressions. Fortunately, as the scalability results in Section 12.5.2 and the minimal query optimization overhead observed in Section 12.5.3 indicate, this guess turned out wrong.

In this section, we additionally look at the three probabilistic search algorithms (Iterative Improvement, Simulated Annealing, and 2-Phase-Optimization), whose parameters are specified in Appendix A.2, that we can currently dispose of in our framework. We are highly interested in answering the following questions:

1. Do the QEPs differ in terms of execution time?
2. How long does it take to derive the QEPs?

3. How does probabilistic search compare to exhaustive search in terms of optimization time and QEP performance?

For answering these questions, we used the PathMark-A queries[11] and evaluated them on an XMark document with scaling factor $f = 5.0$. We created an element index and configured the plan generator with rule set S_1 as defined in Table 12.2. As probabilistic search strategies are not deterministic, we optimized each query thrice and executed each QEP five times for calculating the average optimization time and the average execution time, respectively.

In Figure 12.11(a) on page 203, we illustrate the results for the average execution times in milliseconds (y-axis). For query A 1, bottom-up search found a plan that was around four times faster than equivalent plans derived by the three probabilistic search strategies. For queries A 2 and A 3, the results of all strategies differ insignificantly. Again, for queries A 4 and A 6–A 8, bottom-up search is the clear winner. In contrast, for query A 5, bottom-up search derived the worst plan amongst all search strategies. Though, the QEP derived using bottom-up plan generation is not dramatically worse than its competitors (factor 1.57). For this query, top-down strategies decided to create a bushy QEP, whereas bottom-up plan generation opted for a left-deep QEP. The quality of QEPs derived using Iterative Improvement and Simulated Annealing are more or less on the same level. In contrast, QEPs generated using 2-Phase optimization turned out to be the worst plans in most situations.

The average optimization times for queries A 1–A 8 are depicted in Figure 12.11(b). As the optimization times of bottom-up optimization and the three probabilistic search algorithms show dramatic gaps, we use a logarithmic scale on the y-axis. Obviously, bottom-up optimization stands head and shoulders above the rest. The optimization times do not exceed 23.25 milliseconds (query A 8). While Iterative Improvement requires up to 8 times more time for finding the optimal plan, Simulated Annealing and 2-Phase-Optimization spend even more than two orders of magnitude of optimization time (e. g., query A 5) than bottom-up optimization.

Even though Simulated Annealing may find a slightly better QEP than bottom-up optimization (e. g., query A 5), it recklessly wastes optimization time. Moreover, it is not assured—due to the indeterministic nature of probabilistic search—that this QEP is always found. In contrast, bottom-up plan generation is reliable and frugally consumes optimization time.

[11] We consider these simpler queries appropriate for this test, because they easily allow for join reordering where mistakes by the optimizer can be noticed effortlessly.

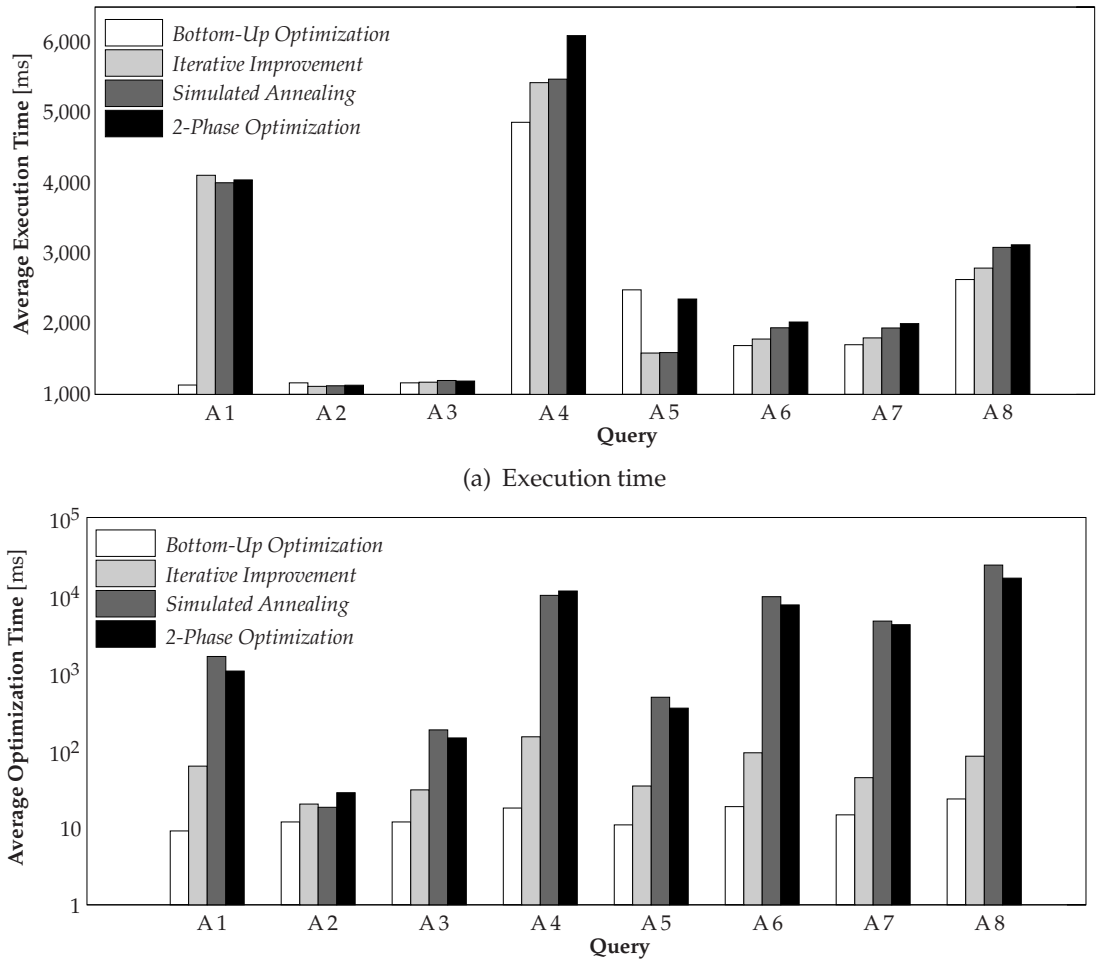
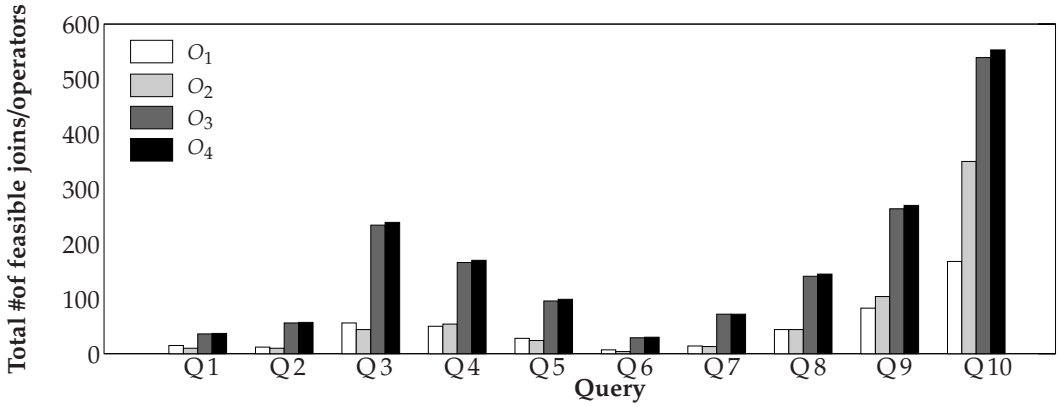


Figure 12.11: Comparison of search strategies

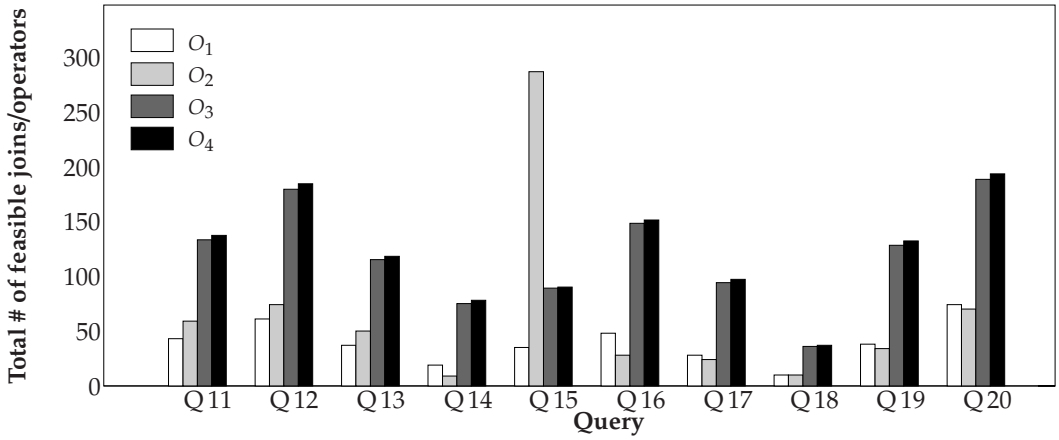
12.5.5 On the Complexity of Plan Enumeration

In Section 9.1, we discussed the upper bounds for search space sizes. From an engineering perspective, it is much more interesting to look at the SJ enumeration complexity from a practical point of view.

Ono and Lohman (1990) have shown that a plan enumerator that uses dynamic programming and is able to generate bushy trees, must at least inspect $(n^3-n)/6$ so-called *feasible joins*, that is, valid join trees in each bottom-up traversal step, for linear queries (in Section 9.1, we referred to them as *chain queries*) with n relations.



(a) Queries Q1–Q10



(b) Queries Q11–Q20

Figure 12.12: Complexity of structural join enumeration

XQuery path expressions can also be considered linear queries, where n is equal to the total number of node tests involved. As our bottom-up plan generator is able to produce bushy trees, we can use the theorem of Ono and Lohman (1990) to estimate the SJ enumeration complexity of our bottom-up plan generation algorithm.

We considered the XMark benchmark on a document with scaling factor $f = 1.0$. For estimating the SJ enumeration complexity, we use the finding above to estimate the enumeration complexity of each path expression specified by the XMark queries. Most queries involve more than one path expression. For them, we estimated the total number of feasible joins for each path expression and aggregated the numbers for comparison with the experimental results.

In Figure 12.12, we depict the empirical results for the actual and estimated complexity of SJ enumeration as O_1 and O_2, respectively. Moreover, we

generalize the concept of feasible joins to feasible operators, that is, we count the total number of valid plans generated in each bottom-up enumeration step. For SJ enumeration and access path selection, we illustrate the total number of feasible operators as O_3. Additionally, O_4 depicts the overhead caused by considering valid plans derived by join fusion.

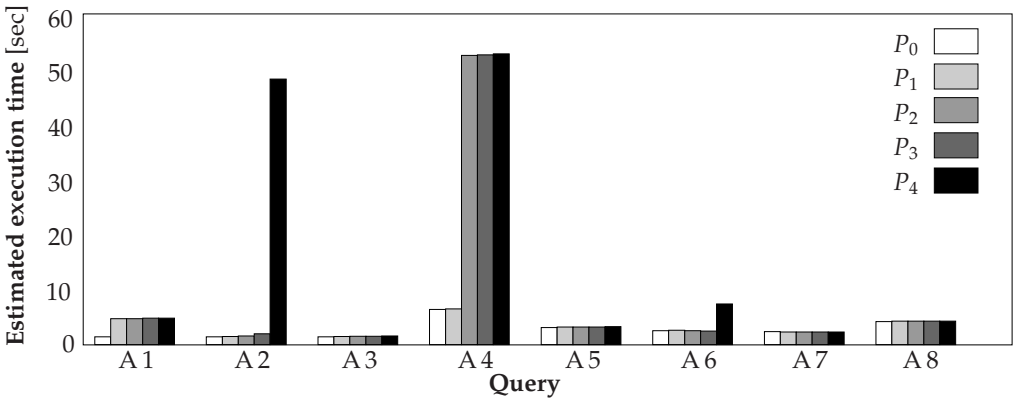
If we compare the results for O_1 and O_2, we see that we can use the theorem of Ono and Lohman (1990) for approximating the total number of feasible SJs for these queries. As the results indicate, the theorem does not provide an exact lower bound for SJ reordering. This is due to two reasons: (1) some join orders cannot be created because of semantic constraints (e. g., in queries Q 4 and Q 10, there are left-outer joins that cannot be reordered, hence, the actual number of feasible joins is lower than the estimated number) and (2) the creation of bushy plans is prevented due to cost restrictions (e. g., query Q 15).

In Section 12.5.1, we learned that for the XMark benchmark, rules IS-ACCESS-2 and SJ-FUSION are the hotspot implementation variation rule and structural variation rule, respectively. Let us now look at the results for O_3 and O_4: The total number of feasible operators increases by an average factor of 3.42 and 3.51, respectively, if we relate the results to O_1. By contrasting O_3 and O_4, we observed that the number of feasible operators increases only marginally, because join fusion can only be applied if there are more than two consecutive SJs at the "beginning" of an SJ tree. Moreover, in a situation where join fusion is considered once as too costly for an SJ cascade, it is not applied any further.

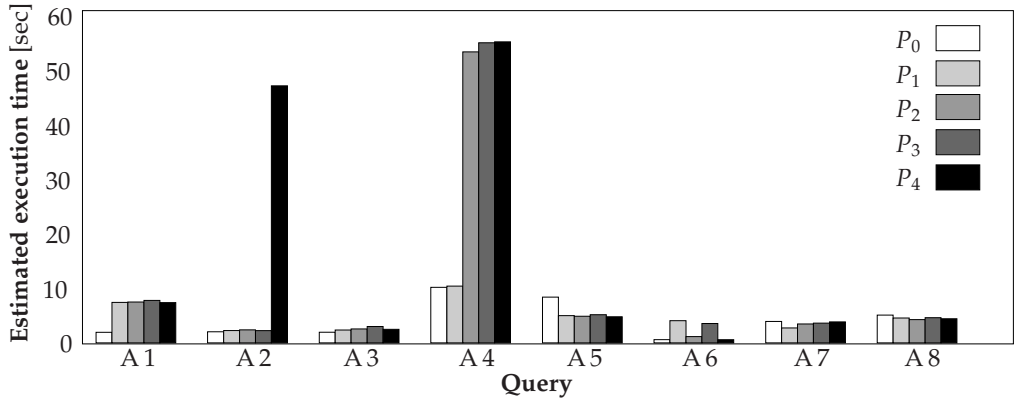
Now, what is the lesson that we learn from this experiment? Though the size of the search space for path expressions can become very large, we did not observe any performance bottlenecks during query optimization. As the results presented in Figure 12.12 show, the search space is effectively pruned by the cost model. In fact, in O_1, not more than 168 plans (query Q 10) had to be inspected to derive an efficient QEP. Hence, bottom-up enumeration is our prefered strategy for XQuery expressions, because we can derive scalable QEPs (Section 12.5.2) and, at the same time, we must only face a marginal overhead for query optimization (Section 12.5.3)

12.5.6 Top 5 Plans

For our final experiment, we modified the bottom-up plan generation in such a way that it retains in each optimization step not only the optimal plan, but also collects the plans whose costs rank on positions two to five. We optimized queries A 1–A 8 (see Appendix C.2 for their definitions) using the modified



(a) Estimated execution time



(b) Actual execution time

Figure 12.13: Estimated and actual execution times of top five plans

plan generation algorithm. In addition to the element index, we also created path indexes p_1, p_2, and p_4 as defined in Appendix C.4.1. In our experiments, the optimal plan is referred to as P_0, whereas the plan on position 5 is referred to as P_4. In Figure 12.13(a), you can see the estimated execution costs for the queries. In contrast, Figure 12.13(b) illustrates the actual execution costs of the queries on an XMark document with scaling factor $f = 10.0$. On the x-axis, we put the query identifiers, whereas the y-axis shows the estimated costs or actual costs for Figure 12.13(a) or Figure 12.13(b), respectively. Let us first look at Figure 12.13(a). For queries A 1, A 3, A 5, A 7, and A 8, we noticed only small differences between the estimated execution costs. Contrariwise, for query A 2, plan P_4 is dramatically more expensive (factor ~30) compared to the remaining four plans. For query A 4, we can see that the first two plans have almost the same estimated costs. For plans P_2–P_4, we also observed remarkably higher costs. Finally, for query A 6, only plan P_4 is expected to be

more costly compared to the best four plans. Actually, the expected costs for P_4 are more than twice as high as for the best plan P_0.

Now, we analyze the actual execution times of the top five plans for queries A 1–A 8. Figure 12.13(b) illustrates the results. If we compare the expected costs and the actual costs for queries A 1–A 4, we can see that the cost model ranked the plans correctly. For example, for query A 2, plan P_3 was expected—and turned out—to be very expensive compared to its competitors. If we look at queries A 5, A 7, and A 8, we can see that the optimizer preferred slightly more expensive plans than the actually optimal plan. For query A 6 and plan P_4, the actual execution time did not turn out being as bad as expected.

Nevertheless, for all queries where there exist dramatic differences between expected execution times and actual execution times of plans, the optimizer selected the best plan correctly.

12.6 Related Work

Ono and Lohman (1990) have shown the benefits of adaptively increasing or decreasing the search space by allowing or preventing the generation of certain shapes of query trees (e. g., bushy trees or star shapes). In Section 12.5.1, we saw that switching off some transformation rules has a positive effect on optimization time while the QEP performance is only marginally decreased. Moreover, in Section 12.5.5, we observed that their theorem can still be used—at least as a lower-bound approximation for the complexity of bottom-up search using dynamic programming.

Chaudhuri et al. (2009) present an interesting approach for testing query optimizers. As reliable cardinality estimation is crucial for effective cost-based optimization and still remains a source of error, the authors supply exact cardinalities to the optimizer for assessing how the system behaves if correct numbers are provided. Currently, our system does not rely on this testing framework. However, we plan to integrate it in the future, because we expect that we can even increase the robustness of our system.

Chaudhuri et al. (2010) propose an approach for profiling query transformation rules. Though their approach focuses on a relational database system, it can be easily transferred to XDBMSs, too. Their analysis identifies a so-called *essential set* of transformation rules, that is, a minimal set of transformation rules for a query that are absolutely necessary for finding an optimal QEP in the search space. The benefit for plan generation in terms of less optimization time is obvious: Using an essential set of transformation rules, the optimal plan is found while optimization overhead is reduced.

12.7 Summary

This chapter presented a comprehensive empirical evaluation of our query optimizer prototype. We have shown that the optimizer, even though it sometimes makes slightly suboptimal decisions, follows the "golden rule" of query optimization—omitting dramatically bad plans.

In Section 12.1, we introduced the hardware and software setup we used for the empirical evaluation. Section 12.2 demonstrated the effectiveness of *text()* push-ups whose corresponding rewrite rules were introduced in Chapter 4. Chapter 7 specified a rule set for cardinality inference whose accuracy and robustness was assessed in Section 12.3. Thereafter, Section 12.4 looked at the correctness of the cost model for various query types—ranging from simple XPath expressions to complex XQuery statements. Next, we looked at different aspects of plan generation in Section 12.5. For example, we found a minimal set of transformation rules for the XMark queries and showed that the optimizer is able to derive scalable QEPs for them. Finally, Section 12.6 discussed related work.

In the final Chapter 13, we highlight the scientific contributions of this work and point out the conclusions.

IV

Conclusions and Outlook

13 Summary and Future Work

"Finally, in conclusion,
let me say just this."

(Peter Sellers)

This chapter concludes this thesis with a short summary of the concepts and techniques presented (Section 13.1). Next, we correlate the ideas developed in this work to our previously published research papers in Section 13.2. In Section 13.3, we will draw conclusions from the empirical results presented in Chapter 12. Thereafter, we provide some design recommendations for effective cost-based XQuery optimization in Section 13.4. Finally, we point out future research directions in Section 13.5.

13.1 Summary

In this work, we introduced and discussed concepts and techniques for cost-based XQuery optimization in native XML database systems. Though we borrowed many ideas from cost-based query optimization in relational database systems, we had to adjust them to make most out of them in the context of XML database systems.

For demonstrating the effectiveness of our concepts, we implemented and integrated our cost-based XQuery optimizer in XTC—our prototype of a native XML database system—that also served as testbed for the empirical evaluation. Moreover, we are grateful that we were permitted to demonstrate our system at ICDE 2010 (Weiner et al., 2010).

The main contribution of this thesis is covered by Parts II (optimization framework) and III (implementation and empirical evaluation).

First, in Part II, we proposed *fn:text()* push-up as yet another query rewrite rule that allows for retaining the costly evaluation of *fn:text()* as long as possible (Chapter 4). Thereafter, Chapter 5 introduced a neat abstraction from XQGM instances (plans) that helped to describe the query optimization concepts developed in the subsequent chapters. In Chapter 6, we defined numerous transformation rules that enabled the query optimizer to perform *implementation variation* (e. g., access path selection) and *structural variation*

(e. g., SJ reordering). Using these rules, the plan generator was able to derive a plethora of equivalent plans that significantly differed in execution times. Cost-based query optimizers assign cost factors to plans that describe their estimated IO and CPU costs. Based on this information, optimizers select the plan with minimal cost for execution. In Chapter 7, we specified a set of *cardinality inference rules*, which adapted the concepts of abstract domain identifiers (Teubner et al., 2008) to XQGM, that are absolutely necessary to perform cost estimation in the subsequent step. Thereafter, Chapter 8 finally defined the cost formulæ of our cost model, where we were able to process cardinality information provided by cardinality inference. Finally, Chapter 9 put all pieces together and specified a bottom-up plan generation algorithm based on dynamic programming and customized probabilistic search algorithms, for example, Simulated Annealing, to our framework.

After all "ingredients" for cost-based XQuery optimization were provided in Part II, Part III looked at the implementation and the empirical evaluation of our system. In Chapter 10, we briefly discussed the implementation of our flexible query optimization framework that is open for further extensions. Next, Chapter 11 emphasized the features of our visual explain tool (XTC Universal Client) that allows to explore the complete query evaluation process from the beginning to its very end. Finally, Chapter 12 assessed our implementation using numerous experiments. For example, we showed that our plan generator is able to derive scalable QEPs for the XMark benchmark queries and defined a minimal set of transformation rules for them.

13.2 Contributions

The results of this thesis are partially based on scientific papers that we have published between 2008 and 2011. The following list shows a selection of the most important publications and stresses their correlations to the chapters of this work:

- Weiner et al. (2008b) proposed the basis for our query transformation rules[1]. For example, this paper introduced the numerous SJ reordering rules and join fusion that form most parts of our structural variation rule set (Chapter 6).

[1] Originally, the idea of join fusion and SJ reordering—together with numerous now outdated rewrite rules—was superfluously discussed as a case study in Weiner (2007). However, at that time, we considered them query rewrite rules at the logical level of a very primitive predecessor of XQGM that cannot be applied anymore to the XQGM specified in Mathis (2009).

- Weiner and Härder (2009) empirically compared a prominent SJ operator (StackTree) and an HTJ operator (TwigOpt) with respect to their performance in varying selectivity scenarios. Based on this analysis, we were able to derive a decision criteria that permits the plan generator to prefer SJs over HTJs and vice versa. This work was very important for estimating the costs of SJs and HTJs in Chapter 8. Moreover, we learned that the grand debate—SJs versus HTJs—will end in a draw, because their performance, at least in realistic query processing scenarios, does not differ significantly.

- Weiner et al. (2010) described our visual explain tool (Chapter 11) that proved to be very useful during the implementation of our optimizer. Even for the empirical evaluation, we were able to easily reconfigure the optimizer with different transformation rule sets and search strategies.

- Weiner and Härder (2010a) outlined the architecture of our query optimization framework (Chapter 10) in a book on XML processing edited by C. Li and T. W. Ling.

- Weiner and Härder (2010b) sketched the concert of transformation rules, cost estimation, and plan generation. In this paper, we were able to provide the first empirical results based on the assessment of our optimizer (Chapter 12).

- Finally, Weiner (2011) was entirely devoted to introducing XQuery cardinality estimation to our framework. Most parts of Chapter 7, which grew in parallel, are based on this paper.

13.3 Conclusions

When we started with developing concepts for cost-based XQuery optimization, we asked ourselves how far we can get with traditional relational optimization techniques. It turned out that we can successfully reuse the basic ideas of traditional cost-based relational query optimization such as transformation rules, cost models, and plan generation algorithms. As we have shown in this thesis, we can provide a complete framework for optimizing the declarative parts of XQuery expressions. This is especially true for XPath expressions for which we can derive numerous alternative plans using our transformation rules, balance multiple access paths against each other, and judge them based on our cost model. Nevertheless, XQuery is a Turing-complete programming language and contains many constructs—being less

relevant for database-centric processing—that cannot be optimized using the principles discussed in this work. For example, *for* and *let* bindings dictate a certain nesting and processing order that must be preserved. Such non-declarative language constructs annihilate effective query optimization in the traditional sense. As the success story of relational database systems is primarily based on a simple data model and a completely declarative query language, it is questionable whether XQuery can follow the footsteps of SQL. If XQuery is supposed to do so, traditional database query optimization must be accompanied by optimization techniques developed by the programming language community.

Nevertheless, XQuery is probably *the* next-generation programming language for the Web, as it has—compared to SQL—the important advantage that an impedance mismatch does not exist anymore. Hence, querying data sources and processing the queried data can be done using a common language, which can consume, manipulate, and return XML. Consequently, querying and programming can join hands and make application development more effective.

Especially for database-centric applications, that is, queries, the language features of XQuery are still open for improvements. In our opinion, the XQuery language must be extended in such a way that it provides constructs for defining twig query patterns in a declarative manner. We believe that this way (1) twig join operators could be used more effectively for query evaluation, as it would then be much easier to identify optimization opportunities for them[2] and (2) the complete query optimization process could be improved, because of higher chances for algebraic as well as non-algebraic optimization.

13.4 Design Recommendations

We believe that our optimization framework provides a neat toolkit for "playing around" with cost-based XQuery optimization in native XML database systems. Based on our experience with XTC's query optimizer, we assembled six recommendations for effective XQuery optimization:

Always create an element index As we have seen in the empirical analysis, efficient query evaluation in native XDBMSs is hardly possible without having an element index. In fact, the performance can be improved by up to two orders of magnitude if the optimizer is allowed to use element index scans.

[2]Currently, twig discovery is a complex task that is far from being trivial (compare Mathis, 2009).

Create TAPs whenever possible Our experiments in Chapter 12 indicated that TAPs are very important access paths in native XDBMSs. Path indexes can replace cascades of SJs and allow to reduce IO and CPU cost at the same time. For value-based predicates, relying on CAS indexes is really frugal.

Path indexes only for multi-step paths Our experiments indicated that single-step path indexes, for example, on *//book*, are only in rare cases superior to an element index on *book* nodes. To reduce the number of indexes that must be compared by the optimizer, we recommend only creating path indexes for multi-step paths.

Left-deep SJ trees are enough Even though we have only seen a small query optimization overhead in Chapter 12, we believe, that for most queries, access path selection is the primary optimization task. Hence, considering only left-deep SJ plans is a heuristics to reduce query optimization time.

The SJ versus HTJ debate is irrelevant We have argued before that typical XQuery expressions use several non-declarative language features that require blocking operators like grouping or unnesting. For native XDBMSs that are evaluating XQuery expressions, there is no clear winner of the SJ versus HTJ debate, because the aforementioned blocking operators as well as *for* and *let* bindings that cannot be removed by the optimizer slow down query execution. Perhaps in the future, there will also be a generalized join operator for XML nodes—similar to its recent relational counterpart (Graefe, 2011)—that will make the join selection problem superfluous.

Bottom-up plan generation works fine Fortunately, we did not observe any performance bottlenecks—even for more complex XQuery expressions with multiple SJ operators—while using bottom-up plan generation. Hence, it is not necessary to rely on probabilistic search algorithms, which are not very robust and might miss the optimal plan.

13.5 Future Research Directions

This thesis introduced cost-based XQuery optimization to native XML database systems. Fairly, we have provided a system prototype that can be used for developing concepts that make query processing in such systems even more effective.

Query rewrite was only a minor concern in this thesis. Nevertheless, we believe that we can "improve" XQGM instances using logical query optimization even before cost-based optimization starts.

Currently, our optimizer can dispose of a plethora of physical algebra operators. Nevertheless, we believe that the query optimizer can benefit from an additional set of alternative SJ and HTJ implementations. Moreover, external sort operators are still missing in our prototype.

We specified a comprehensive set of query transformation rules. Currently, our rules for SJ reordering cover only semi joins and full joins. By extending them to outer joins, we are convinced that we can create additional alternatives that might be more efficient than the plans that our plan generator can derive at the moment. As a starting point, considering the approach of Neumann and Moerkotte (2009) is promising.

The effectiveness of cardinality inference was shown in Chapter 12. Most rules provide rather rough estimations. We assume that these rules can still be refined to improve cardinality estimation quality.

As recent publications on query optimizer testing and profiling (e. g., compare Giakoumakis and Galindo-Legaria, 2008; Chaudhuri et al., 2010) indicate, even after 30 years of research on relational query optimizers, there is little understanding how these complex systems behave in scenarios for which they are not customized for. Most systems provide a plethora of transformation rules that handle special cases induced by customers and might undermine finding optimal plans even for trivial SPJ queries.

V

Appendix

A Optimizer Configuration File

A.1 Optimizer Parameter Values

Access path	$PageFetchCost_{IO}$ [ms]
Document Index	0.78
Element Index	2.48
Path Index	2.45
CAS Index	1.85

Table A.1: Optimizer settings

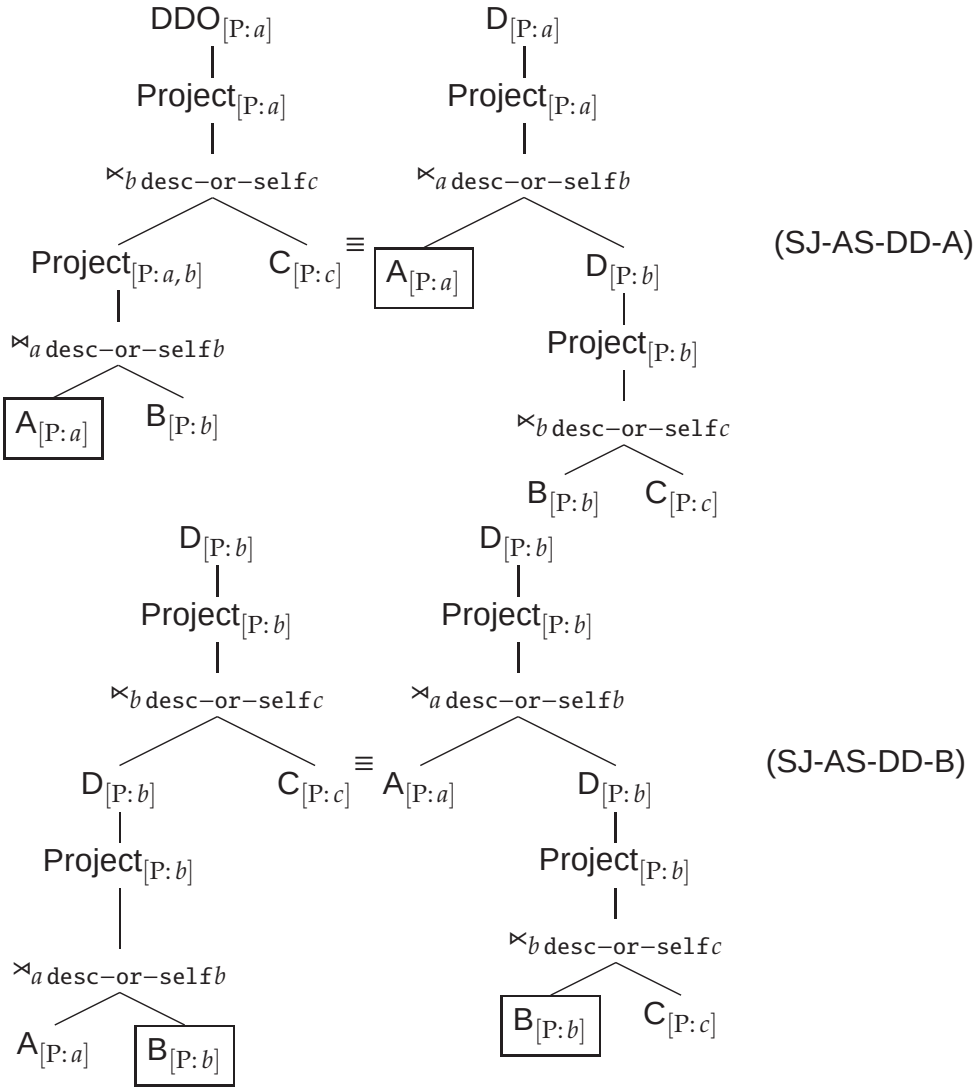
A.2 Probabilistic Search Parameters

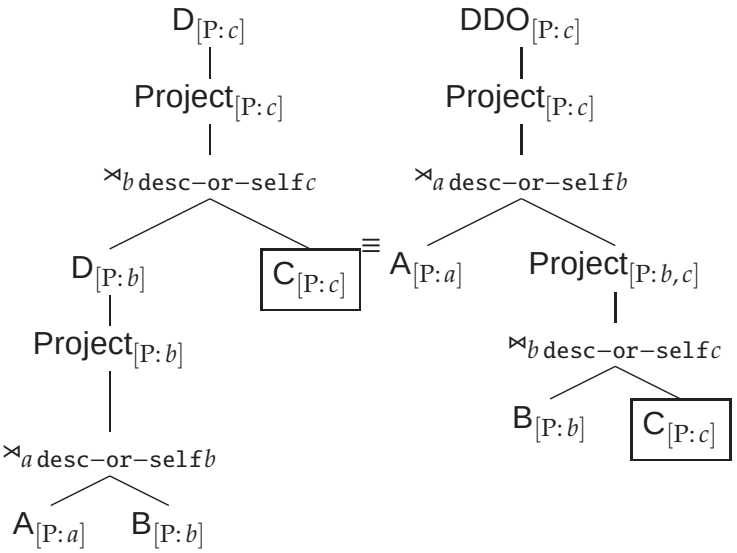
```
<optimizers>
  ...
  <optimizer id="Q4" strategy="ITERATIVE_IMPROVEMENT"
   transformer="RANDOM_TOP_DOWN">
  ...
    <parameters>
      <parameter key="iiNoGlobalOptimizations" value="5" />
      <parameter key="iiNoLocalOptimizations" value="5" />
    </parameters>
  </optimizer>

  <optimizer id="Q5" strategy="SIMULATED_ANNEALING"
   transformer="RANDOM_TOP_DOWN">
    ...
    <parameters>
      <parameter key="saNoLocalOptimizations" value="5" />
      <parameter key="saTemperatureReduction" value="0.01" />

      <parameter key="saTemperatureInitializationFactor"
```

```
        value="2.0" />
      <parameter key="saEquilibriumFactor" value="16.0" />
      <parameter key="saFrozenCount" value="3" />
      <parameter key="saFrozenTemperature" value="0.1" />
      <parameter key="saProbabilityThreshold" value="0.8" />
    </parameters>
  </optimizer>

  <optimizer id="Q6" strategy="TWO_PHASE_OPTIMIZATION"
   transformer="LINEAR_TOP_DOWN">
    ...
    <parameters>
      <parameter key="iiNoGlobalOptimizations" value="5" />
      <parameter key="iiNoLocalOptimizations" value="5" />
      <parameter key="saNoLocalOptimizations" value="5" />
      <parameter key="saTemperatureReduction" value="0.01" />
      <parameter key="saTemperatureInitializationFactor"
        value="2.0" />
      <parameter key="saEquilibriumFactor" value="16.0" />
      <parameter key="saFrozenCount" value="3" />
      <parameter key="saFrozenTemperature" value="0.1" />
      <parameter key="saProbabilityThreshold" value="0.8" />
    </parameters>
  </optimizer>
</optimizers>
```

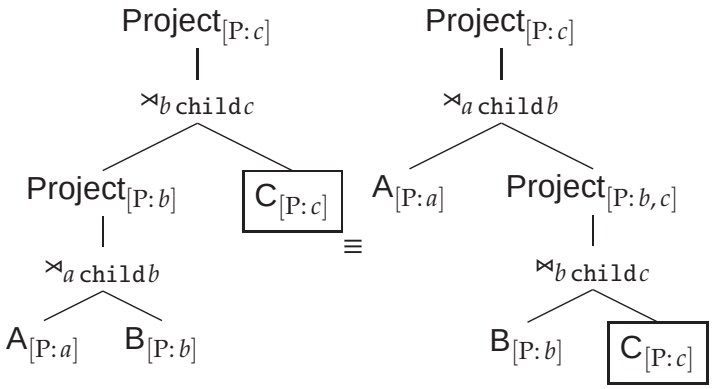
B A Glimpse at Structural Join Associativity Rules

In this Section, we illustrate some join associativity rules (SJ-AS-DD-*) for two structural joins that evaluate the *descendant-or-self* axis and one associativity rule (SJ-AS-CC-C) for two structural joins that evaluate the *child* axis. The complete set of 28 rules was already introduced and discussed in Weiner (2007); Weiner et al. (2008a,b).





(SJ-AS-DD-C)



(SJ-AS-CC-C)

C Query Sets and Index Definitions

C.1 XMark Benchmark Queries

Q 1

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person
where $b/@id = "person0"
return $b/name/text()
```

Q 2

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

Q 3

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2 <=
 $b/bidder[last()]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/text()}"
  last="{$b/bidder[last()]/increase/text()}"/>
```

Q 4

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

Q5

```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price)
```

Q6

```
let $auction := doc("auction.xml") return
for $b in $auction//site/regions return count($b//item)
```

Q7

```
let $auction := doc("auction.xml") return
for $p in $auction/site
return
  count($p//description) + count($p//annotation) +
   count($p//emailaddress)
```

Q8

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Q9

```
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let
    $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
```

```
    let $n := for $t2 in $ei where $t/itemref/@item =
     $t2/@id return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

Q 10

```
let $auction := doc("auction.xml") return
for $i in
  distinct-values($auction/site/people/person/profile/
   interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

Q 11

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
```

```
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

Q 12

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

Q 13

```
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>
```

Q 14

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

Q 15

```
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/annotation/
   description/parlist/listitem/parlist/listitem/text/
    emph/keyword/text()
return <text>{$a}</text>
```

Q 16

```
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/
       parlist/listitem/text/emph/
        keyword/
        text()))
return <person id="{$a/seller/@person}"/>
```

Q 17

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

Q 18 Please note, our implementation of XTC does currently not provide support for user-defined functions. Hence, we inlined function *local:convert*, which converts Dutch florins (DFL) to Euros.

```
let $auction := doc("auction.xml") return for $i in
$auction/site/open_auctions/open_auction return
zero-or-one($i/reserve) * 2.20371)
```

Q 19

```
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{$k}">{$b/location/text()}</item>
```

Q 20

```
let $auction := doc("auction.xml") return
<result>
  <preferred>
    {count($auction/site/people/person/profile
```

```
    [@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/person/profile
     [@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

C.2 XPathMark-A Queries

A 1

```
doc("auction.xml")/site/closed_auctions/closed_auction/
 annotation/description/text/keyword
```

A 2

```
doc("auction.xml")//closed_auction//keyword
```

A 3

```
doc("auction.xml")/site/closed_auctions/closed_auction//keyword
```

A 4

```
doc("auction.xml")/site/closed_auctions/closed_auction
 [annotation/description/text/keyword]/date
```

A 5

```
doc("auction.xml")/site/closed_auctions/closed_auction
 [descendant::keyword]/date
```

A 6

```
doc("auction.xml")/site/people/person[profile/gender and
  profile/age]/name
```

A 7

```
doc("auction.xml")/site/people/person[phone or homepage]/name
```

A 8

```
doc("auction.xml")/site/people/person[address and
 (phone or homepage) and (creditcard or profile)]/name
```

C.3 XPath Queries with Value-Based Predicates

B 1

```
doc('auction.xml')//asia/item[location='Germany']
```

B 2

```
doc('auction.xml')//asia/item[location > 'C'
 and location <= 'G']
```

B 3

```
doc('auction.xml')//text//*[keyword >= 'c' and keyword <= 'd']
```

B 4

```
doc('auction.xml')//profile[@income > 40000][age <= 19]
```

C.4 Index Definitions

C.4.1 Indexes for Cost Estimation

- p_1=create path index paths /site/closed_auctions/closed_auction
- p_2=create path index paths /site/people/person
- p_3=create cas index paths //@income of type double
- p_4=create path index paths //keyword

C.4.2 Indexes for Value-Based Predicates

- `create cas index paths //asia/item/location`
- `create cas index paths //text//*/keyword`
- `create cas index paths //age of type integer`

C.4.3 Indexes for Plan Generation on XMark

- `create path index paths //keyword`
- `create path index paths /site/people/person/profile`
- `create path index paths /site/people/person`
- `create path index paths /site/open_auctions/open_auction`
- `create path index paths /site/closed_auctions/closed_auction`
- `create path index paths //site//regions`
- `create path index paths /site/regions/australia/item`
- `create cas index paths //@income of type double`
- `create cas index paths //@id`
- `create cas index paths //@person`

D Proof—Search Space Size for Hybrid Join Plans

Let us assume a query plan consists of $n > 2$ Access plans and $n - 1$ StructuralJoin plans. Furthermore, let us refer to the total number of different joins orders by $f(n)$. Using Join Fusion (Section 6.3.3), we can replace two or more StructuralJoin plans by a single TwigJoin plan.

Basis If $n = 3$, we can create a plan that does not contain any StructuralJoin plans anymore. Instead, it contains a single TwigJoin plan that receives inputs from 3 Access operators.

For $n = 4$, we can additionally create hybrid plans, that is, plans that consist of a StructuralJoin plan and a TwigJoin plan. For them, we can create $f(2)$ join orders. In total, we can create $1 + f(2) = 1 + \sum_{i=2}^{2} f(n-i)$ plans containing TwigJoin plans.

Again, if we assume $n = 5$, we can create different join orders for two hybrid plans: (1) a plan that contains two StructuralJoin plans and a TwigJoin plan that receives inputs from three Access plans and (2) another plan plan that contains only a single StructuralJoin plan and a single TwigJoin plan that is now connected to four Access plans. For $n = 5$, we can create $1 + f(2) + f(3) = 1 + \sum_{i=2}^{3} f(n-i)$ different hybrid plans.

Inductive step We assume that for $n > 2$, the following equation holds:

$$g(n) = 1 + f(2) + f(3) + \ldots + f(n-2) = 1 + \sum_{i=2}^{n-2} f(n-i) \tag{D.1}$$

Proof Now, we have to show by induction over n that $g(n+1)$ holds, too. Using our hypothesis (D.1), we get:

$$
\begin{aligned}
g(n+1) &= \underbrace{1 + f(2) + f(3) + \ldots + f(n-2)}_{\text{D.1}} + f\Big((n+1)-2\Big) \\
&= 1 + \sum_{i=2}^{n-2} f(n-i) + f\Big((n+1)-2\Big) \\
&= 1 + \sum_{i=2}^{n-2} f(n-i) + f\Big((n+1)-2\Big) \\
&= 1 + \sum_{i=2}^{n-2} f(n-i) + f(n-1) \\
&= 1 + \sum_{i=1}^{n-2} f(n-i) \\
&= 1 + \sum_{i=1}^{n-2} f(n-1+1-i) \\
&= 1 + \sum_{i=2}^{n-1} f(n+1-i) \\
&= 1 + \sum_{i=2}^{(n+1)-2} f\Big((n+1)-i\Big)
\end{aligned}
$$

■

Bibliography

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J.: *The Lorel Query Language for Semistructured Data*, in Intl. Journal on Digital Libraries, vol. 1 (1) (1997), pages 68–88.

Aboulnaga, A., Alameldeen, A. R., and Naughton, J. F.: *Estimating the Selectivity of XML Path Expressions for Internet Scale Applications*, in Proc. 27th Intl. Conf. on Very Large Data Bases (VLDB), September 11–14, 2001, Roma, Italy, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-804-4, 2001, pages 591–600.

Aguiar Moraes Filho, J.: *Summarizing XML Documents: Contributions, Empirical Studies, and Challenges*, Ph.D. thesis, University of Kaiserslautern, Germany, 2010.

Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., et al.: *Structural Joins: A Primitive for Efficient XML Query Pattern Matching*, in Proc. 18th Intl. Conf. on Data Engineering (ICDE), February 26–March 1, 2002, San Jose, CA, USA, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7695-1531-2, 2002, pages 141–154.

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., et al.: *System R: Relational Approach to Database Management.*, in ACM Transactions on Database Systems, vol. 1 (2) (1976), pages 97—137, ISSN 0362-5915.

Balmin, A., Eliaz, T., Hornibrook, J., Lim, L., Lohman, G. M., et al.: *Cost-Based Optimization in DB2 XML*, in IBM Syst. J., vol. 45 (2) (2006), pages 299–320.

Beyer, K., Cochrane, R., Hvizdos, M., Josifovski, V., Kleewein, J., et al.: *DB2 Goes Hybrid: Integrating Native XML and XQuery With Relational Data and SQL*, in IBM Syst. J., vol. 45 (2) (2006), pages 271–298, ISSN 0018-8670.

Beyer, K., Gemulla, R., Haas, P. J., Reinwald, B., and Sismanis, Y.: *Distinct-Value Synopses for Multiset Operations*, in Commun. ACM, vol. 52 (2009), pages 87–95, ISSN 0001-0782.

Beyer, K. S., Cochrane, R., Josifovski, V., Kleewein, J., Lapis, G., et al.: *System RX: One Part Relational, One Part XML*, in Proc. ACM SIGMOD Intl. Conf. on Management of Data, June 14–16, 2005, Baltimore, MD, USA, ACM Press, New York, NY, USA, ISBN 1-59593-060-4, 2005, pages 347–358.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., et al.: *XQuery 1.0: An XML Query Language—W3C Recommendation 23 January 2007*, `http://www.w3.org/TR/2007/REC-xquery-20070123/`, 2007.

Bohannon, P., Freire, J., Haritsa, J. R., Ramanath, M., Roy, P., et al.: *LegoDB: Customizing Relational Storage for XML Documents*, in Proc. 28th Intl. Conf. on Very Large Data Bases (VLDB), August 20–23, 2002, Hong Kong, China, Morgan Kaufmann Publishers, Burlington, MA, USA, 2002, pages 1091–1094.

Boncz, P.: *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*, Ph.D. thesis, Dept. of Computer Science, University of Amsterdam, Netherlands, 2002.

Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., et al.: *MonetDB/XQuery: A fast XQuery Processor Powered by a Relational Engine*, in Proc. 2006 ACM SIGMOD Intl. Conf. on Management of Data, June 27–29, 2006, Chicago, IL, USA, ACM Press, New York, NY, USA, ISBN 1-59593-434-0, 2006, pages 479–490.

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., et al.: *Pathfinder: XQuery—The Relational Way*, in Proc. 31st Intl. Conf. on Very Large Data Bases (VLDB), August 30–September 2, 2005, Trondheim, Norway, ACM Press, New York, NY, USA, ISBN 1-59593-154-6, 1-59593-177-5, 2005, pages 1322–1325.

Brantner, M., Helmer, S., Kanne, C.-C., and Moerkotte, G.: *Full-fledged Algebraic XPath Processing in Natix*, in Proc. 21st Intl. Conf. on Data Engineering (ICDE), April 5–8, 2005, Tokyo, Japan, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7695-2285-8, 2005, pages 705–716.

Bray, T., Paoli, J., and Sperberg-McQueen, C. M.: *Extensible Markup Language (XML) 1.0—W3C Recommendation 10-February-1998*, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Male, E., and Yergeau, F.: *Extensible Markup Language (XML) 1.0 (Fifth Edition)—W3C Recommendation 26 November 2008*, `http://www.w3.org/TR/2008/REC-xml-20081126/`, 2008.

Bruno, N., Chaudhuri, S., and Ramamurthy, R.: *Power Hints for Query Optimization*, in Proc. 25th Intl. Conf. on Data Engineering (ICDE), March 29–April 2, 2009, Shanghai, China, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3545-6, 2009, pages 469–480.

Bruno, N., Koudas, N., and Srivastava, D.: *Holistic Twig Joins: Optimal XML Pattern Matching*, in Proc. 2002 ACM SIGMOD Intl. Conf. on Management of Data, June 3–6,2002, Madison, WI, USA, ACM Press, New York, NY, USA, ISBN 1-58113-497-5, 2002, pages 310–321.

Carey, M. J., DeWitt, D. J., Graefe, G., Haight, D. M., Richardson, J. E., et al.: *The EXODUS Extensible DBMS Project: An Overview*, in S. B. Zdonik and D. Maier (eds.), Readings in Object-Oriented Database Systems, pages 474–499, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-55860-000-0, 1990.

Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Robie, J., et al.: *XQuery Update Facility 1.0—W3C Candidate Recommendation 09 June 2009*, `http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/`, 2009.

Chaudhuri, S.: *An Overview of Query Optimization in Relational Systems.*, in Proc. 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1–3, 1998, Seattle, WA, USA, ACM Press, New York, NY, USA, ISBN 0-89791-996-3, 1998, pages 34–43.

Chaudhuri, S.: *Technical Perspective: Relational Query Optimization—Data Management Meets Statistical Estimation*, in Commun. ACM, vol. 52 (2009), pages 86–86, ISSN 0001-0782.

Chaudhuri, S., Giakoumakis, L., Narasayya, V. R., and Ramamurthy, R.: *Rule Profiling for Query Optimizers and Their Implications*, in Proc. 26th Intl. Conf. on Data Engineering (ICDE), March 1–6, 2010, Long Beach, CA, USA, IEEE Computer Society, Washington, DC, USA, ISBN 978-1-4244-5444-0, 2010, pages 1072–1080.

Chaudhuri, S., Narasayya, V. R., and Ramamurthy, R.: *Exact Cardinality Query Optimization for Optimizer Testing*, in Proc. VLDB Endowment, vol. 2 (1) (2009), pages 994–1005, ISSN 2150-8097.

Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V. J., and Zaniolo, C.: *Efficient Structural Joins on Indexed XML Documents.*, in Proc. 28th Intl. Conf. on Very Large Data Bases (VLDB), August 20–23, 2002, Hong Kong, China,

Morgan Kaufmann Publishers, Burlington, MA, USA, 2002, pages 263–274.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: *Introduction To Algorithms, Second Edition*, The MIT Press, Cambridge, MA, USA, 2001, ISBN 0-262-03293-7.

Das, D. and Batory, D. S.: *Praire: A Rule Specification Framework for Query Optimizers*, in Proc. 11th Intl. Conf. on Data Engineering (ICDE), March 6–10, 1995, Taipei, Taiwan, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-8186-6910-1, 1995, pages 201–210.

DeHaan, D. and Tompa, F. W.: *Optimal Top-Down Join Enumeration*, in Proc. ACM SIGMOD Intl. Conf. on Management of Data, June 12–14, 2007, Beijing, China, ACM Press, New York, NY, USA, ISBN 978-1-59593-686-8, 2007, pages 785–796.

Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., et al.: *XQuery 1.0 and XPath 2.0 Formal Semantics—W3C Recommendation 23 January 2007*, `http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/`, 2007.

Ellson, J., Gansner, E., Koutsofios, E., and Woodhull, S. N. G.: *Graphviz and Dynagraph—Static and Dynamic Graph Drawing Tools*, in M. Junger and P. Mutzel (eds.), Graph Drawing Software, pages 127–148, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-540-00881-1, 2003.

Fallside, D. C. and Walmsley, P.: *XML Schema Part 0: Primer Second Edition—W3C Recommendation 28 October 2004*, `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, 2004.

Fernández, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N.: *XQuery 1.0 and XPath 2.0 Data Model (XDM)—W3C Recommendation 23 January 2007*, `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/`, 2007.

Fernández, M. F., Siméon, J., Choi, B., Marian, A., and Sur, G.: *Implementing XQuery 1.0: The Galax Experience*, in Proc. 29th Intl. Conf. on Very Large Data Bases (VLDB), September 9–12, 2003, Berlin, Germany, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-12-722442-4, 2003, pages 1077–1080.

Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., et al.: *Anatomy of a Native XML Base Management System*, in VLDB Journal, vol. 11 (4) (2002), pages 292–314.

Fisher, D. K. and Maneth, S.: *Structural Selectivity Estimation for XML Documents*, in Proc. 23rd Intl. Conf. on Data Engineering (ICDE), April 15–20, 2007, Istanbul, Turkey, IEEE Computer Society Press, Washington, DC, USA, ISBN 1-4244-0802-4, 2007, pages 626–635.

Fomichev, A., Grinev, M., and Kuznetsov, S. D.: *Sedna: A Native XML DBMS*, in Proc. 32nd Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM), January 21–27, 2006, Merín, Czech Republic, vol. 3831 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-31198-X, 2006, pages 272–281.

Fontoura, M., Josifovski, V., Shekita, E. J., and Yang, B.: *Optimizing Cursor Movement in Holistic Twig Joins*, in Proc. 2005 ACM Intl. Conf. on Information and Knowledge Management (CIKM), October 31–November 5, 2005, Bremen, Germany, ACM Press, New York, NY, USA, ISBN 1-59593-140-6, 2005, pages 784–791.

Franceschet, M.: *XPathMark: An XPath Benchmark for the XMark Generated Data*, in Proc. 3rd Intl. XML Database Symposium, XSym 2005, August 28–29, 2005, Trondheim, Norway, vol. 3671 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-28583-0, 2005, pages 129–143.

Freire, J., Haritsa, J. R., Ramanath, M., Roy, P., and Siméon, J.: *StatiX: Making XML Count*, in Proc. 2002 ACM SIGMOD Intl. Conf. on Management of Data, June 3–6, 2002, Madison, WI, USA, ACM Press, New York, NY, USA, ISBN 1-58113-497-5, 2002, pages 181–191.

Freytag, J. C.: *The Basic Principles of Query Optimization in Relational Database Management Systems*, in Proc. 11th IFIP World Computer Congress, August 28–September 1, 1989, San Francisco, CA, USA, 1989, pages 801–807.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, 1995, ISBN 0-201-63361-2.

Giakoumakis, L. and Galindo-Legaria, C. A.: *Testing SQL Server's Query Optimizer: Challenges, Techniques, and Experiences*, in IEEE Data Engineering Bulletin, vol. 31 (1) (2008), pages 36–43.

Goldman, R. and Widom, J.: *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*, in Proc. 23rd Intl. Conf. on Very

Large Data Bases (VLDB), August 25–29, 1997, Athens, Greece, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-470-7, 1997, pages 436–445.

Graefe, G.: *Rule-Based Query Optimization in Extensible Database Systems*, Ph.D. thesis, University of Wisconsin-Madison, Madison, WI, USA, 1987.

Graefe, G.: *Encapsulation of Parallelism in the Volcano Query Processing System*, in Proc. 1990 ACM SIGMOD Intl. Conf. on Management of Data, May 23–25, 1990, Atlantic City, NJ, USA, ACM Press, New York, NY, USA, ISSN 0163-5808, 1990, pages 102–111.

Graefe, G.: *Query Evaluation Techniques for Large Databases*, in ACM Computing Surveys, vol. 25 (2) (1993), pages 73–170.

Graefe, G.: *The Cascades Framework for Query Optimization*, in IEEE Data Engineering Bulletin, vol. 18 (3) (1995), pages 19–29.

Graefe, G.: *The Microsoft Relational Engine*, in Proc. 12th Intl. Conf. on Data Engineering (ICDE), February 26–March 1, 1996, New Orleans, LA, USA, IEEE Computer Society, Washington, DC, USA, ISBN 0-8186-7240-4, 1996, pages 160–161.

Graefe, G.: *A Generalized Join Algorithm*, in Proc. 14. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW 2011), March 2–4 2011, Kaiserslautern, Germany, vol. 180 of Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn, Germany, ISBN 978-3-88579-274-1, 2011, pages 267–286.

Graefe, G. and DeWitt, D. J.: *The EXODUS Optimizer Generator*, in Proc. 1987 ACM SIGMOD Intl. Conf. on Management of Data, May 27–29, 1987, San Francisco, CA, USA, ACM Press, New York, NY, USA, ISBN 0-89791-236-5, 1987, pages 160–172.

Graefe, G. and McKenna, W. J.: *The Volcano Optimizer Generator: Extensibility and Efficient Search*, in Proc. 9th Intl. Conf. on Data Engineering (ICDE), April 19–23, 1993, Vienna, Austria, IEEE Computer Society, Washington, DC, USA, ISBN 0-8186-3570-3, 1993, pages 209–218.

Grust, T., van Keulen, M., and Teubner, J.: *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps*, in Proc. 29th Intl. Conf. on Very Large Data Bases (VLDB), September 9–12, 2003, Berlin, Germany, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-12-722442-4, 2003, pages 524–525.

Haas, L. M., Freytag, J. C., Lohman, G. M., and Pirahesh, H.: *Extensible Query Processing in Starburst*, in Proc. 1989 ACM SIGMOD Intl. Conf. on Management of Data, May 31–June 2, 1989, Portland, OR, USA, ACM Press, New York, NY, USA, 1989, pages 377–388.

Hameurlain, A. and Morvan, F.: *Evolution of Query Optimization Methods*, in Trans. on Large-Scale Data- and Knowledge-Centered Systems I, vol. 5740 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-03721-4, 2009, pages 211–242.

Hamilton, J.: *Interview Database Dialogue with Pat Selinger*, in Commun. ACM, vol. 51 (12) (2008), pages 32–35, ISSN 0001-0782.

Härder, T.: *DBMS Architecture—The Layer Model and its Evolution*, in Datenbank-Spektrum, vol. 13 (2005), pages 45–57.

Härder, T., Haustein, M. P., Mathis, C., and Wagner, M.: *Node Labeling Schemes for Dynamic XML Documents Reconsidered*, in Data & Knowledge Engineering, vol. 60 (1) (2007), pages 126–149.

Härder, T., Mathis, C., Bächle, S., Schmidt, K., and Weiner, A. M.: *Essential Performance Drivers in Native XML DBMSs*, in Proc. 36th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM), January 23–29, 2010, Spindleruv Mlýn, Czech Republic, vol. 5901 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-11265-2, 2010, pages 29–46.

Härder, T. and Rahm, E.: *Datenbanksysteme—Konzepte und Techniken der Implementierung*, Springer-Verlag, Berlin, Heidelberg, 2001, ISBN 3-540-42133-5.

Härder, T. and Reuter, A.: *Concepts for Implementing a Centralized Database Management System*, in H. J. Schneider (ed.), Proc. Intl. Computing Symposium on Application Systems Development, Nürnberg, März 1983, Teubner Verlag, Wiesbaden, 1983, pages 28–60.

Haustein, M. P.: *Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen*, Ph.D. thesis, Dept. of Computer Science, University of Kaiserslautern, Germany, Verlag Dr. Hut, München, Germany, 2006, ISBN 3-89963-280-X.

Haustein, M. P. and Härder, T.: *An Efficient Infrastructure for Native Transactional XML Processing*, in Data & Knowledge Engineering, vol. 61 (3) (2007), pages 500–523.

Hoare, C. A. R.: *Quicksort*, in The Computer Journal, vol. 5 (1) (1962), pages 10–15.

Ibaraki, T. and Kameda, T.: *On the Optimal Nesting Order for Computing n-Relational Joins*, in ACM Transactions on Database Systems, vol. 9 (3) (1984), pages 482–502, ISSN 0362-5915.

Ioannidis, Y. E.: *Query Optimization*, in A. B. Tucker (ed.), The Computer Science and Engineering Handbook, CRC Press, Boca Raton, FL, USA, ISBN 1-58488-360-X, 1997, pages 1038–1057.

Ioannidis, Y. E. and Kang, Y. C.: *Randomized Algorithms for Optimizing Large Join Queries*, in Proc. 1990 ACM SIGMOD Intl. Conf. on Management of Data, May 23–25, 1990, Atlantic City, NJ, USA, ACM Press, New York, NY, USA, ISBN 0-89791-365-5, 1990, pages 312–321.

Ioannidis, Y. E. and Wong, E.: *Query Optimization by Simulated Annealing*, in Proc. 1987 ACM SIGMOD Intl. Conf. on Management of Data, May 27–29, 1987, San Francisco, CA, USA, ACM Press, New York, NY, USA, ISBN 0-89791-236-5, 1987, pages 9–22.

Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., et al.: *TIMBER: A Native XML Database*, in VLDB Journal, vol. 11 (4) (2002), pages 274–291.

Jarke, M. and Koch, J.: *Query Optimization in Database Systems*, in ACM Computing Surveys, vol. 16 (2) (1984), pages 111–152.

Jiang, H., Lu, H., Wang, W., and Ooi, B. C.: *XR-Tree: Indexing XML Data for Efficient Structural Joins*, in Proc. 19th Intl. Conf. on Data Engineering (ICDE), March 5–8, 2003, Bangalore, India, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7803-7665-X, 2003, pages 253–263.

Kabra, N. and DeWitt, D. J.: *OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization*, in VLDB Journal, vol. 8 (1) (1999), pages 55–78.

Kacimi, M. and Neumann, T.: *System R (R*) Optimizer*, in Encyclopedia of Database Systems, pages 2900–2905, Springer-Verlag, New York, NY, ISBN 978-0-387-35544-3, 2009.

Kaushik, R., Krishnamurthy, R., Naughton, J. F., and Ramakrishnan, R.: *On the Integration of Structure Indexes and Inverted Lists*, in Proc. 2004 ACM

SIGMOD Intl. Conf. on Management of Data, June 13–18, 2004, Paris, France, ACM Press, New York, NY, USA, ISBN 1-58113-859-8, 2004, pages 779–790.

Kay, M.: *Ten Reasons Why Saxon XQuery is Fast*, in IEEE Data Engineering Bulletin, vol. 31 (4) (2008), pages 65–74.

Knuth, D. E.: *Structured Programming with go to Statements*, in ACM Computing Surveys, vol. 6 (4) (1974), pages 261–301.

Lanzelotte, R. S. G. and Valduriez, P.: *Extending the Search Strategy in a Query Optimizer*, in Proc. 17th Intl. Conf. on Very Large Data Bases (VLDB), September 3–6, 1991, Barcelona, Catalonia, Spain, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-150-3, 1991, pages 363–373.

Lehner, W. and Schöning, H.: *XQuery—Grundlagen und fortgeschrittene Methoden*, dpunkt.verlag, Heidelberg, 2004, ISBN 978-3-89864-266-8.

Loeser, H.: *XML Storage—It's The Flexibility, Stupid!*, Colloquium of the Department of Computer Science, University of Kaiserslautern, November 11, 2008.

Lohman, G. M.: *Grammar-Like Functional Rules for Representing Query Optimization Alternatives*, in Proc. 1988 ACM SIGMOD Intl. Conf. on Management of Data, June 1–3, 1988, Chicago, IL, USA, ACM Press, New York, NY, USA, ISBN 0-89791-268-3, 1988, pages 18–27.

Lohman, G. M., Mohan, C., Haas, L. M., Daniels, D., Lindsay, B. G., et al.: *Query Processing in R**, in W. Kim, D. S. Reiner, and D. S. Batory (eds.), Query Processing in Database Systems, pages 31–47, Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-13831-5, 1985.

Mackert, L. F. and Lohman, G. M.: *R* Optimizer Validation and Performance Evaluation for Distributed Queries*, in Proc. 12th Intl. Conf. on Very Large Data Bases (VLDB), August 25–28, 1986, Kyoto, Japan, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-934613-18-4, 1986, pages 149–159.

Mannino, M. V., Chu, P., and Sager, T.: *Statistical Profile Estimation in Database Systems*, in ACM Computing Surveys, vol. 20 (3) (1988), pages 191–221.

Mathis, C.: *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*, Ph.D. thesis, Dept. of Computer Science,

University of Kaiserslautern, Germany, Verlag Dr. Hut, München, Germany, 2009, ISBN 978-3-86853-209-8.

Mathis, C., Härder, T., and Schmidt, K.: *Storing and Indexing XML Documents Upside Down*, in Computer Science—Research and Development, vol. 24 (1-2) (2009), pages 51–68.

Mathis, C., Weiner, A. M., Härder, T., and Hoppen, C. R. F.: *XTCcmp: XQuery Compilation on XTC*, in Proc. VLDB Endowment, vol. 1 (2) (2008), pages 1400–1403.

May, N.: *An Algebraic Approach to XQuery Optimization*, Ph.D. thesis, University of Mannheim, Germany, 2007.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J.: *Lore: A Database Management System for Semistructured Data*, in ACM SIGMOD Record, vol. 26 (3) (1997), pages 54–66.

McHugh, J. and Widom, J.: *Query Optimization for XML*, in Proc. 25th Intl. Conf. on Very Large Data Bases (VLDB), September 7–10, 1999, Edinburgh, Scotland, UK, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-615-7, 1999, pages 315–326.

McJones, P.: *System R*, Online available at: `http://www.mcjones.org/System_R/`, 2011.

McKenna, W. J., Burger, L., Hoang, C., and Truong, M.: *EROC: A Toolkit for Building NEATO Query Optimizers*, in Proc. 22nd Intl. Conf. on Very Large Data Bases (VLDB), September 3–6, 1996, Mumbai, India, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-382-4, 1996, pages 111–121.

Meier, W.: *eXist: An Open Source Native XML Database*, in Web, Web-Services, and Database Systems, October 7–10, 2002, Erfurt, Germany, vol. 2593 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-00745-8, 2002, pages 169–183.

Meng, X., Luo, D., Lee, M.-L., and An, J.: *OrientStore: A Schema Based Native XML Storage System*, in Proc. 29th Intl. Conf. on Very Large Data Bases (VLDB), September 9–12, 2003, Berlin, Germany, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-12-722442-4, 2003, pages 1057–1060.

Michiels, P., Mihaila, G. A., and Siméon, J.: *Put a Tree Pattern in Your Algebra*, in Proc. 23rd Intl. Conf. on Data Engineering (ICDE), April 15–20, 2007, Istanbul, Turkey, IEEE Computer Society Press, Washington, DC, USA, 2007, pages 246–255.

Mitschang, B.: *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte*, Reihe Datenbanksysteme, Vieweg, Braunschweig, Wiesbaden, 1995, ISBN 3-528-05488-3.

Moerkotte, G.: *Building Query Compilers (Under Construction)*, Online available at: `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`, 2009, Version of September 3, 2009.

Moerkotte, G. and Neumann, T.: *Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products*, in Proc. 32nd Intl. Conf. on Very Large Data Bases (VLDB), September 12–15, 2006, Seoul, Korea, VLDB Endowment, ISBN 1-59593-385-9, 2006, pages 930–941.

Muralikrishna, M. and DeWitt, D. J.: *Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries*, in Proc. 1988 ACM SIGMOD Intl. Conf. on Management of Data, June 1–3, 1988, Chicago, IL, USA, ACM Press, New York, NY, USA, ISBN 0-89791-268-3, 1988, pages 28–36.

Naughton, J. F., DeWitt, D. J., Maier, D., Aboulnaga, A., Chen, J., et al.: *The Niagara Internet Query System*, in IEEE Data Engineering Bulletin, vol. 24 (2) (2001), pages 27–33.

Neumann, T.: *Query Simplification: Graceful Degradation for Join-Order Optimization*, in Proc. ACM SIGMOD Intl. Conf. on Management of Data, June 29–July 2, 2009, Providence, RI, USA, ACM Press, New York, NY, USA, ISBN 978-1-60558-551-2, 2009, pages 403–414.

Neumann, T. and Moerkotte, G.: *Generating Optimal DAG-Structured Query Evaluation Plans*, in Computer Science—Research and Development, vol. 24 (3) (2009), pages 103–117.

O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., et al.: *ORDPATHs: Insert-Friendly XML Node Labels*, in Proc. ACM SIGMOD Intl. Conf. on Management of Data, June 13–18, 2004, Paris, France, ACM Press, New York, NY, USA, ISBN 1-58113-859-8, 2004, pages 903–908.

Ono, K. and Lohman, G. M.: *Measuring the Complexity of Join Enumeration in Query Optimization*, in Proc. 16th Intl. Conf. on Very Large Data Bases (VLDB), August 13–16, 1990, Brisbane, Queensland, Australia, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-149-X, 1990, pages 314–325.

Piatetsky-Shapiro, G.: *The Optimal Selection of Secondary Indices is NP-Complete*, in SIGMOD Record, vol. 13 (2) (1983), pages 72–75, ISSN 0163-5808.

Pirahesh, H., Hellerstein, J. M., and Hasan, W.: *Extensible/Rule Based Query Rewrite Optimization in Starburst*, in Proc. 1992 ACM SIGMOD Intl. Conf. on Management of Data, June 2–5, 1992, San Diego, CA, USA, ACM Press, New York, NY, USA, ISBN 0-89791-521-6, 1992, pages 39–48.

Poosala, V. and Ioannidis, Y. E.: *Selectivity Estimation Without the Attribute Value Independence Assumption*, in Proc. 23rd Intl. Conf. on Very Large Data Bases (VLDB), August 25–29, 1997, Athens, Greece, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 1-55860-470-7, 1997, pages 486–495.

Poosala, V., Ioannidis, Y. E., Haas, P. J., and Shekita, E. J.: *Improved Histograms for Selectivity Estimation of Range Predicates*, in Proc. 1996 ACM SIGMOD Intl. Conf. on Management of Data, June 4–6, 1996, Montreal, QC, Canada, ACM Press, New York, NY, USA, ISSN 0163-5808, 1996, pages 294–305.

Rao, P. and Moon, B.: *PRIX: Indexing and Querying XML Using Prüfer Sequences*, Tech. Rep. 03-06, Dept. of Computer Science, University of Arizona, Tuscon, AZ, USA, 2003, Online available at: `ftp://ftp.cs.arizona.edu/reports/2003/TR03-06.pdf`.

Ré, C., Siméon, J., and Fernández, M. F.: *A Complete and Efficient Algebraic Compiler for XQuery*, in Proc. 22nd Intl. Conf. on Data Engineering (ICDE), April 3–8, 2006, Atlanta, GA, USA, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7695-2570-9, 2006, page 14.

Rizzolo, F. and Mendelzon, A.: *Indexing XML Data with ToXin*, in Proc. 4th Intl. Workshop on the Web and Databases (WebDB), May 24–25, 2001, Santa Barbara, CA, USA, 2001, pages 49–54.

Rosenthal, A. and Reiner, D. S.: *An Architecture for Query Optimization*, in Proc. 1982 ACM SIGMOD Intl. Conf. on Management of Data, June 2–4,

1982, Orlando, FL, USA, ACM Press, New York, NY, USA, ISBN 0-89791-073-7, 1982, pages 246–255.

Russell, S. J. and Norvig, P.: *Artificial Intelligence—A Modern Approach*, Prentice Hall Series in Artificial Intelligence, Prentice Hall Intl., London, UK, Second (International) ed., 2003, ISBN 0-13-080302-2.

Sartiani, C.: *A General Framework for Estimating XML Query Cardinality*, in 9th Intl. Workshop on Database Programming Languages (DBPL), September 6–8, 2003, Potsdam, Germany, vol. 2921 of Lecture Notes in Computer Science (LNCS), Springer-Verlag, Berlin, Heidelberg, ISBN 3-540-20896-8, 2004, pages 257–277.

Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., et al.: *XMark: A Benchmark for XML Data Management*, in Proc. 28th Intl. Conf. on Very Large Data Bases (VLDB), August 20–23, 2002, Hong Kong, China, Morgan Kaufmann Publishers, Burlington, MA, USA, 2002, pages 974–985.

Schmidt, K. and Härder, T.: *On the Use of Query-Driven XML Auto-Indexing*, in Proc. 26th Intl. Conf. on Data Engineering (ICDE) Workshops, March 1–6, 2010, Long Beach, CA, USA, IEEE Computer Society Press, Washington, DC, USA, ISBN 978-1-4244-6522-4, 2010, pages 81–86.

Schöning, H.: *Tamino—A DBMS Designed for XML*, in Proc. 17th Intl. Conf. on Data Engineering (ICDE), April 2–6, 2001, Heidelberg, Germany, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1001-9, 2001, pages 149–154.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G.: *Access Path Selection in a Relational Database Management System*, in Proc. 1979 ACM SIGMOD Intl. Conf. on Management of Data, May 30–June 1, 1979, Boston, MA, USA, ACM Press, New York, NY, USA, ISBN 0-89791-001-X, 1979, pages 23—34.

Shapiro, L. D., Maier, D., Benninghoff, P., Billings, K., Fan, Y., et al.: *Exploiting Upper and Lower Bounds In Top-Down Query Optimization*, in Proc. Intl. Database Engineering and Applications Symposium (IDEAS), July 16–18, 2001, Grenoble, France, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1140-6, 2001, pages 20–33.

Teubner, J., Grust, T., Maneth, S., and Sakr, S.: *Dependable Cardinality Forecasts for XQuery*, in Proc. VLDB Endowment, vol. 1 (1) (2008), pages 463–477.

Wang, H., Park, S., Fan, W., and Yu, P. S.: *ViST: a Dynamic Index Method for Querying XML Data by Tree Structures*, in Proc. 2003 ACM SIGMOD Intl. Conf. on Management of Data, June 9–12, 2003, San Diego, CA, USA, ACM Press, New York, NY, USA, ISBN 1-58113-634-X, 2003, pages 110–121.

Wang, W., Jiang, H., Lu, H., and Yu, J. X.: *Bloom Histogram: Path Selectivity Estimation for XML Data with Updates*, in Proc. 30th Intl. Conf. on Very Large Data Bases (VLDB), August 31–September 3, 2004, Toronto, ON, Canada, Morgan Kaufmann Publishers, Burlington, MA, USA, ISBN 0-12-088469-0, 2004, pages 240–251.

Weiner, A. M.: *Plangenerierung zur Anfrageauswertung in nativen XML-Datenbankverwaltungssystemen*, Diploma thesis, University of Kaiserslautern, Germany, 2007.

Weiner, A. M.: *Advanced Cardinality Estimation in the XML Query Graph Model*, in Proc. 14. GI-Fachtagung Datenbanksysteme für Business, Techologie und Web (BTW), February 28–March 4, Kaiserslautern, Germany, vol. P-180 of Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn, Germany, ISBN 978-3-88579-274-1, 2011, pages 207–226.

Weiner, A. M. and Härder, T.: *Using Structural Joins and Holistic Twig Joins for Native XML Query Optimization*, in Proc. 13th East European Conf. on Advances in Databases and Information Systems (ADBIS), September 7–10, 2009, Riga, Latvia, vol. 5739 of Lecture Notes in Computer Science (LNCS), ISBN 978-3-642-03972-0, 2009, pages 149–163.

Weiner, A. M. and Härder, T.: *A Framework for Cost-Based Query Optimization in Native XML Database Management Systems*, in C. Li and T. W. Ling (eds.), Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization, and Data Query Technologies, pages 160–182, IGI Global, ISBN 1-61520-727-9, 2010a.

Weiner, A. M. and Härder, T.: *An Integrative Approach to Query Optimization in Native XML Database Management Systems*, in Proc. 14th Intl. Database Engineering and Applications Symposium (IDEAS), August 16–18, 2010, Montreal, QC, Canada, ACM Intl. Conf. Proc. Series, ACM Press, New York, NY, USA, ISBN 978-1-60558-900-8, 2010b, pages 64–74.

Weiner, A. M., Härder, T., and da Silva, R. O.: *Visualizing Cost-Based XQuery Optimization*, in Proc. 26th Intl. Conf. on Data Engineering (ICDE), March

1–6, 2010, Long Beach, CA, USA, IEEE Computer Society Press, Washington, DC, USA, ISBN 978-1-4244-5444-0, 2010, pages 1165–1168.

Weiner, A. M., Mathis, C., and Härder, T.: *Associativity Rules for Native XML Databases*, Internal Report, Online available at: `http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/publications/2008/WMH08.Int.pdf`, 2008a.

Weiner, A. M., Mathis, C., and Härder, T.: *Rules for Query Rewrite in Native XML Databases*, in 3rd Intl. Workshop on Database Technologies for Handling XML Information on the Web (DataX) in conjunction with 11th Intl. Conf. on Extending Database Technology (EDBT), March 25, 2008, Nantes, France, ACM Press, New York, NY, USA, ISBN 978-1-59593-966-1, 2008b, pages 21–26.

Weiner, A. M., Mathis, C., Härder, T., and Hoppen, C. R. F.: *Now it's Obvious to The Eye—Visually Explaining XQuery Evaluation in a Native XML Database Management System*, in Proc. 13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW 2009), March 2–6, 2009, Münster, Germany, vol. P-144 of Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn, Germany, ISBN 978-3-88579-238-3, 2009, pages 616–619.

Wu, Y., Patel, J., and Jagadish, H.: *Structural Join Order Selection for XML Query Optimization*, in Proc. 19th Intl. Conf. on Data Engineering (ICDE), March 5–8, 2003, Bangalore, India, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7803-7665-X, 2003, pages 443–454.

Zhang, N., Özsu, M. T., Aboulnaga, A., and Ilyas, I. F.: *XSEED: Accurate and Fast Cardinality Estimation for XPath Queries*, in Proc. 22nd Intl. Conf. on Data Engineering (ICDE), April 3–8, 2006, Atlanta, GA, USA, IEEE Computer Society Press, Washington, DC, USA, ISBN 0-7695-2570-9, 2006, page 61.

Index

Curriculum Vitæ

Personal Information	
Lastname	Weiner
Firstname	Andreas Matthias
Day of Birth	June 11, 1981
Place of Birth	Siegburg, Germany
Marital Status	Single
Nationality	German
Business Address	Gottlieb-Daimler-Straße 47 67663 Kaiserslautern, Germany Phone: +49 (0) 631 – 205 3072 Fax: +49 (0) 631 – 205 3299 E-mail: weiner@cs.uni-kl.de

Academic Studies	
August 2007–today	Scientific staff member and Ph. D. candidate at Kaiserslautern University of Technology (Databases and Information Systems Group)
July 2007	Graduation Academic Degree: Diplom-Informatiker
October 2002–July 2007	Studies of Applied Computer Science at Kaiserslautern University of Technology Major: Databases and Enterprise Information Systems, Minor: Business Administration, Artificial Intelligence, and Knowledge Management

Work Experience	
January 2007–July 2007	Programmer, XTC project
October 2006–December 2006	Research Internship, DaimlerChrysler TSS, Ulm
January 2006–September 2006	Programmer, XTC project

Alternative Civilian Service	
September 2001–June 2002	Kreisseniorenzentrum St.-Maximilian Kolbe, Kenzingen (Senior Citizens' Residence)

Education	
July 2001	General Qualification for University Entry (Allgemeine Hochschulreife)
September 1998–July 2001	Secondary School: Wirtschaftsgymnasium Emmendingen
September 1991–July 1998	Secondary School: Markgrafen-Realschule Emmendingen
September 1987–July 1991	Primary School: Grundschule Windenreute