Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: 15/03

Efficient Management of Semistructured XML Data

Carlo Sartiani

Referee
Alberto Mendelzon

Referee
Dan Suciu

Supervisor
Giorgio Ghelli

Chair
Andrea Maggiolo-Schettini

Abstract

The last few years have seen the rapid emerging of the eXtensible Markup Language (XML). XML was designed as a simplification of SGML, and it has become the most widely used universal data representation format. In particular, the ability of XML to easily represent data with irregular structure has imposed XML as the standard incarnation for *semistrucured* data, i.e., data with irregular, unstable, or even unknown structure.

In the context of XML data management systems, the estimation of query cardinality is becoming more and more important: the information provided by a query result estimator can be used as input to the query optimizer, as an early feedback to user queries, as well as input for determining an optimal storage schema.

This Thesis describes the result size estimation model of Xtasy, a prototype XML database management system. Unlike other existing models, which focus on very limited subsets of XQuery, the proposed model covers the FLWR core of XQuery, and estimate not only the *raw* cardinality of query results, but also their distribution.

To my beloved family (cats included)

I HAVE a rendezvous with Death
At some disputed barricade,
When Spring comes back with rustling shade
And apple-blossoms fill the air
I have a rendezvous with Death
When Spring brings back blue days and fair.

It may be he shall take my hand
And lead me into his dark land
And close my eyes and quench my breath
It may be I shall pass him still.
I have a rendezvous with Death
On some scarred slope of battered hill,
When Spring comes round again this year
And the first meadow-flowers appear.

God knows 'twere better to be deep
Pillowed in silk and scented down,
Where love throbs out in blissful sleep,
Pulse nigh to pulse, and breath to breath,
Where hushed awakenings are dear...
But I've a rendezvous with Death
At midnight in some flaming town,
When Spring trips north again this year,
And I to my pledged word am true,
I shall not fail that rendezvous.

I Have a Rendezvous with Death, Alan Seeger

Contents

1 Introduction **7**

I Overview **9**

2 XML and Related Technologies **11**
2.1 XML Overview 11
2.2 XQuery 16

3 Issues in XML Queries Result Size Estimation **19**
3.1 Introduction 19
3.2 Path and Twig Cardinality Estimation 21
3.3 Predicate Selectivity Estimation 22
3.4 Groups 22
3.5 Nested Queries 23

4 State of the Art in XML Result Size Estimation **25**
4.1 Correlated Subpath Trees 25
4.2 TIMBER's Models 26
4.3 Niagara's Models 28
4.4 StatiX Statistical Model 29

5 Thesis Contribution **31**

II The System **33**

6 System Overview **35**
6.1 Architecture 35
6.1.1 Query Parser 37
6.1.2 Query Translator 38
6.1.3 Persistent Store Manager 39
6.1.4 Database Manager 39
6.1.5 XML Manager 40

6.1.6 Catalog Manager 40
6.1.7 Index Manager 40

7 Query Algebra 41
7.1 Data Model and Term Language 41
7.2 Algebra Operators 43
7.2.1 *Env* structures 43
7.2.2 *Border* Operators 44
7.2.3 Basic Operators 48
7.3 Optimization Properties 53
7.3.1 Classical equivalences 53
7.3.2 *path* decompositions 54
7.3.3 Nested queries equivalences 57
7.4 Expressive Power 60
7.4.1 Relational Completeness 61
7.4.2 Representation and Translation of XQuery Queries 61

8 Storage Schema 63
8.1 Node Clustering 63
8.2 The Structural Index 66
8.3 Structural Labeling 66

9 Physical Operators 69
9.1 General Concepts 69
9.2 Unary Physical Operators 70
9.3 Binary Physical Operators 75
9.4 Evaluation of Nested Queries 76
9.4.1 Assigning structural labels to nested query results 80

III The Result Size Estimator 83

10 Estimation Framework 85
10.1 Issues in Result Size Estimation 85
10.2 The Framework 88
10.2.1 Basics 88
10.2.2 Tagged Region Graph 89
10.2.3 Match Occurrences, ECLSs, and ETLSs 92
10.2.4 Cardinality Notions 93
10.2.5 Correlation 94
10.2.6 Group cardinality estimation 95
10.2.7 Predicate selectivity estimation 97
10.3 Framework Algorithms 100

11 Statistics **105**
11.1 Tagged Region Graph 105
11.2 Correlation Table 107
11.3 Path Statistics 108
11.4 Predicate Selectivity Factors 108
11.5 Statistics Collection Algorithms 109

12 Estimation Model **113**
12.1 Basic Concepts 113
12.2 ECLS and ETLS Manipulation Functions 116
12.3 Estimation for Unary Physical Operators 116
12.4 Estimation for Binary Physical Operators 123

13 Experimental Results **129**
13.1 Configuration 129
13.1.1 Space Requirement Experiments 129
13.1.2 Path Statistics 130
13.2 Accuracy Tests 130
13.2.1 Benchmark queries 140
13.2.2 Error metrics 141
13.2.3 Experimental results 141

14 Conclusions **155**

Bibliography **157**

A Formalizations **165**
A.1 Formalization 165
A.1.1 *Env* and tuples operations 165
A.1.2 Support operators 165
A.1.3 Basic Operators 166
A.1.4 Path 166
A.1.5 Return 167

B Proofs **169**
B.1 Nested Query Rewriting Rules 169

C Benchmarks **175**
C.1 XMark Queries 175
C.1.1 Path queries 175
C.1.2 Twig queries 175
C.1.3 Twig queries with groups 176
C.1.4 Predicate queries 176
C.1.5 Nested queries 176

C.1.6 Negative queries 177
C.2 DBLP Queries 177
C.2.1 Path queries 177
C.2.2 Twig queries 177
C.2.3 Twig queries with groups 178
C.2.4 Predicate queries 178
C.2.5 Nested queries 178
C.2.6 Negative queries 179

Chapter 1

Introduction

The last few years have seen the rapid emerging of the eXtensible Markup Language (XML [BPSM98]). XML was originally designed to bring structured documents to the Web, hence reducing the chaos introduced by poor HTML code as well as by proprietary HTML extensions. Despite its original objective, XML succeeded as a universal data representation (as a matter of fact, the Web is still full of poorly written HTML documents); the ability of XML to represent nearly any kind of data, together with being a well-documented international standard, contributed to much of its success.

During the same years, part of the database community focused its research efforts on *semistructured data*, i.e., data with an irregular or quite unstable structure. Semistructured data are usually represented by means of labeled graphs or trees, the labels capturing the structural properties of data. Since XML is a syntax for representing node-labeled trees (even graphs, if a proper semantic interpretation is performed), XML has become the *de-facto* standard representation format for semistructured data, hence making the two research areas fuse together.

Much work has been done in the field of XML and semistructured data management, ranging from data models and query languages (e.g., UnQL [Bun97] [BDS95], Strudel [FFK+98], YATL [CDSS98], [CDQT02], WebOQL [AM98], Lorel [QRSU95], XML-QL [DFF+99], XQL [RDF+99], Quilt [CRF00], and XQuery [BCF+03]), to query algebras (e.g., YAT [CDSS98], SAL [BT99], TAX [JLST01], the Xtasy algebra [SA02], and the former W3C XML Query Algebra [FSW01]), to storage models (e.g., STORED [DFS99], StatiX [BFRS02], and also [STZ+99] and []), and to optimization techniques and index data structures [CCMS98] [MS99] [FM00].

Many systems for storing, querying, and manipulating XML and semistructured data also appeared. These systems can be categorized in *main-memory* systems, which load data entirely in main memory (e.g., IPSI-XQ [ips] [gal]), *back-end independent* systems, which are designed to work with any back-end compliant to a given interface (e.g., TOX [BBM+01] and Kweelt [Sah00]), and *persistent* systems, which persistently store data in secondary storage: this class can be further divided in so-called *native* systems (e.g., [tam] and [tim]) and in object-relational systems

with XPath/XQuery interfaces (e.g., XPeranto [CKS+00], Silkroute [FKS+02], but also [CKN03] and [DTCÖ03]).

Thesis contribution This Thesis deals with the problem of managing semistructured XML data, and, in particular, with the problem of estimating the cardinality of XML queries. The information provided by the query result size estimator can be used as input to the query optimizer (cost equations usually rely on the estimation of intermediate result size), as an early feedback to the user about the selectivity of her query, as input for determining the optimal storage schema [FHR+02], and as input for the compiler/run-time system of database programming languages.

The Thesis presents an estimation model for the FLWR core of XQuery: for what is known to the author, no other model covers such fragment of XQuery, other models being limited to twig queries with predicates.

Thesis outline The Thesis is organized as follows. Chapter 2 briefly illustrates XML and its related technologies. Chapter 3, then, introduces the main issues in result size estimation, and Chapter 4 covers the state of the art in the field.

Chapters 6, 7, 8, and 9 describe the system on top of which the result size estimator has been built. Chapter 10, then, describes the estimation framework that forms the basis for the model; Chapter 11 discusses the statistics being used, and Chapter 12 illustrates the estimation model. In Chapter 14, finally, we draw our conclusions.

Part I

Overview

Chapter 2

XML and Related Technologies

This Chapter describes XML and some related technologies, such as schema languages, transformation languages, and query languages. Due to the vastness of the subject, we refer the reader to the literature for a more detailed description.

2.1 XML Overview

XML [BPSM98] was born as a *meta-language* for document markup. Designed to bring *structured* documents to the Web, it allows the user to define custom markup schemes, hence freeing her from the constraints of special purpose markup languages, such as HTML [RHJ99].

XML has been designed as a simplification and an evolution of SGML [SGM86]: many unusual SGML constructs were removed, in order to simplify the development of XML tools, and to broaden its application field; moreover, XML has been created for delivering richly structured documents to the Web, which was not feasible which SGML.

Besides its original incarnation, XML assumed new roles. Indeed, XML is a representation syntax for node-labeled forests (in particular, node-labeled trees), which implies that XML can be used for representing nearly any kind of data, from structured documents to purely unstructured data. As a consequence, XML became the *de-facto* standard representation format for semistructured data, i.e., data with irregular or unstable structure, as well as the standard format for data exchange among heterogeneous applications.

An XML document is basically a sequence of *elements*. As in SGML, each element has a *tag*, and may contain a list of *attributes*. Consider, for example, the following XML fragment:

```
<patient patientID="12345">
   <name> Mark Greene </name>
   ...
```

```
</patient>
```

This fragment describes a patient element whose content, delimited by a pair $(start - tag, end - tag)$, consists of a nested list of elements; the patient element contains also an attribute patientID whose value is the string "12345".

An XML element may contain a nested list of XML elements, as the patient element, a string value, as the name element, or a mix of string values and nested elements. An XML element may also be empty.

An XML document must obey *well-formedness* rules, which themselves define the notion of "XML document": in particular, unlike SGML, elements must be properly nested. Consider, for example, the following fragment:

```
<patient patientID="12345">
   <name> </patient> Mark Greene </name>
```

This is a legal SGML document fragment, but it does not obey XML well-formedness rules, since the patient tag is closed before the closing of the name tag. By definition, if a document is not well-formed, it is not an XML document.

A total order relation is defined among elements of an XML document. This relation reflects the ordering of XML elements in the textual representation of the document. Moreover, a local order is defined among the children elements of the same father; this order must be compatible with the global ordering. Unlike SGML, no ordering is defined among attributes of the same element.

XML documents can also be seen as node-labeled trees, where nodes represent elements, attributes, PIs, etc, and are labeled with tags, attribute names, etc. The total order relation among elements is compatible with the ordering induced by a depth first traversal of the tree.

This is the very basic core of XML. On top of this core, many other technologies were defined, ranging from schema languages such as XML DTDs and XML Schema, to transformation languages (e.g., XSLT), and to query languages (e.g., XML-QL and XQuery). Below we will briefly examine these technologies.

Document Type Definitions XML inherits from SGML the notion of *DTD*. DTDs (*Document Type Definitions*) are tree grammars, which define the syntactical structure of XML documents as well as some constraints on the values that might be present in the document. Consider, for example, the XML fragment shown in the previous example; it can be described by the following DTD:

```
<!ELEMENT patientDatabase (patient)+>
<!ELEMENT patient (name)>
<!ATTLIST patient
          patientID    ID    #REQUIRED>
<!ELEMENT name #PCDATA>
```

This DTD defines a patientDatabase element, which should contain a non-empty list of patient element (`(patient)+`). The patient element in turn must contain a string-valued name element; moreover, each patient element must have a *type-ID* attribute patientID (`<!ATTLIST ... #required>`).

Type-ID attributes are exploited, in conjunction with the type-IDREF ones, to establish links among different elements in a document. Consider, for example, the following XML element, where the patient attribute has been declared as a type-IDREF one:

```
<prescription patient = "12345">
```

This element references a patient element, hence establishing a connection between a prescription and the corresponding patient.

Type-ID attributes have to be *unique* in a document, i.e., there must not exist two different elements with the same ID. Because the scope of an attribute definition is always the whole document, this constraint ensures the non-ambiguousness of references, but it may cause problems when fragments coming from different documents are merged together.

It should be noted that type-ID and type-IDREF attributes are meaningful only in the presence of a DTD: indeed, unlike SGML, XML DTDs are optional, and XML documents can live without them.

DTD definitions are globally scoped; this implies that the tag of an element identifies its type. Moreover, as in common language grammars, DTD definitions can be organized in a *recursive* fashion, as shown below.

```
<!ELEMENT chapter  (title, section*)>
<!ELEMENT section  (title, section*)>
<!ELEMENT title    (#PCDATA)>
```

A document which fully satisfies a given DTD is said to be *valid* with respect to that DTD.

XML Schema XML Schema [TBMM02] [BM02] is a schema language designed to overcome some limitations of DTDs, such as the absence of datatypes, the 1-1 correspondence between tags and types, and the impossibility of local definitions.

A schema basically consists of several element and types definitions; an element definition specifies the element name as well as the type of its content, which can be described locally, or by referring to an external type definition.

The following example illustrates the basic feature of XML Schema.

Example 2.1.1 Consider the DTD about the patient database; this DTD can be rewritten as follows.

```
<xsd:element name = "patientDatabase" type = "PatientDatabase"/>
<xsd:complexType name = "PatientDatabase"/>
   <xsd:sequence>
      <xsd:element ref = "patient" minOccurs = "1"
                                   maxOccurs = "unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:element name = "patient">
   <xsd:complexType>
      <xsd:element name = "name" type = "xsd:string"/>
      <xsd:attribute name = "patientID" type = "xsd:string"
                     use = "required"/>
   </xsd:complexType>
</xsd:element>
```

This schema defines an outer element, patientDatabase, of type PatientDatabase-Type; PatientDatabaseType is defined as a list of one or more patient elements, whose type is locally defined (despite its name, `<xsd:sequence>` is used for defining records). ■

XML Transformation Language Together with XML Schema, XSLT [Cla99] is one of the most widely used technologies directly related to XML. XSLT is a rule-based transformation language for XML documents: an XSLT program (*XSL stylesheet* [Dea01]) consists of a collection of rules (*templates*), which are applied to the input document to form the output document. Transformation rules can be applied recursively, and they can contain conditional and iterative statements.

Consider, for example, the following XSLT program:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"  xmlns="http://www.w3.org/1999/xhtml">
<xsl:template match="patientDatabase">
  <?xml version="1.0" encoding="iso-8859-1"?>
  <!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/
     xhtml1-transitional.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
       <title>Patient List</title>
    </head>
    <body bgcolor="#ffffff">
      <table border="3">
        <xsl:template match="patient">
```

```
          <tr>
            <td>
              <xsl:apply-templates select="name"/>
            </td>
            <td>
              <xsl:apply-templates select="@patientID"/>
            </td>
          </tr>
        </xsl:template>
      </table>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

This stylesheet creates an XHTML document containing a table with information about patients. Template rules are applied iteratively to match any patient in the database, and to extract the relevant information.

XML-QL XML-QL [DFF+98], designed by the same team as StruQL [FFK+97], is an evolution and adaptation of StruQL to XML. It is based on a variant of the OEM data model [PWM96].

This language is able to express querying operations by means of *patterns*, which describe paths and conditions on XML documents (where clause); the variables bound by the where clause are, then, used in transformation operations for reconstructing arbitrary XML graphs (*construct* clause).

The following example illustrates a sample XML-QL query.

Example 2.1.2 Referring to the patient database, suppose you want to retrieve the list of all patients whose name is "Mark Greene". This can be done with the following XML-QL query.

```
where <database>
         <patient>
            <name> Mark Greene </name>
         </patient> AS $p
      </database> IN "patientdatabase.xml"
construct <result> $p </result>
```

■

The input of an XML-QL query is a set of XML graphs, and its output is always an XML graph; hence, it is possible to compose queries, e.g., a user query over a *view definition* query.

2.2 XQuery

XQuery [BCF+03] is a functional, *Turing-complete* query language for XML data. XQuery has been designed as an extension of Quilt [CRF00] with concepts taken from other languages; its definition, together with the definition of its brother XPath 2.0, is still *in-progress*, even though there exists a relatively stable core of the language.

Unlike other query languages for XML and semistructured data, XQuery is strongly and statically typed: each expression has given a type at compile-time. The type system of XQuery, which drained much of the effort of the W3C XML Query Working Group, is mainly oriented toward *result analysis*, i.e., the analysis of the inferred type of query results.

The core of XQuery is represented by FLWR expressions (for, let, where, and return clauses). for and let clauses are used for exploring data trees, and for binding variables: the for clause iteratively binds a variable to a sequence of nodes, hence generating as many bindings as the sequence cardinality; the let clauses, instead, binds the variable to the whole sequence, hence generating only one binding: the let clause, together with nested queries, can also be used to mimic the behavior of groupby operators, still missing in the language.

The following example briefly illustrates the binding clauses of XQuery.

Example 2.2.1 Consider the following query on the patient database.

```
for $p in input()//patient,
    $n in $p/name
```

This clause bounds $\$p$ to each patient element in the database, and in turn $\$n$ to the name sub-element of each patient element. Variable bindings are concatenated to obey the dependent product semantics. ■

The where condition is used for filtering variable bindings; this clause allows the user to specify existentially as well as universally quantified predicates, which can be freely connected through boolean connectors.

The return clause, finally, is used for constructing new XML fragments.

The following shows a complete FLWR query.

Example 2.2.2 Consider the following query.

```
<newPatients> {
              for $p in input()//patient,
                  $n in $p/name
              where $p/id > 12000
              return $n }
</newPatients>
```

This query returns a single newPatients element, containing the name of recently enrolled patients. ■

Beyond this basic core, XQuery allows the user to define her own functions, to arbitrarily sort the result of a FLWR expression(sortby/orderby), to freely nest expressions, etc. Given its features, XQuery can be considered as a database programming language rather than a plain query language.

Chapter 3

Issues in XML Queries Result Size Estimation

As briefly stated in Chapter 1, the main argument of this thesis is the study of the issues related to the estimation of result size for XML queries. This Chapter presents an overview of such issues.

3.1 Introduction

Result size estimation for XML queries requires the system to predict the cardinality of nearly any query in the language. Referring to the FLWR fragment of XQuery, the most problematic aspects concern the estimation of path and twig cardinality, the estimation of predicate selectivity, the estimation of group cardinality (let binder of XQuery), as well as the estimation of the cardinality of nested queries. While path and twig estimation is a peculiar issue of XML and semistructured query languages, predicate, group, and nested queries cardinality estimation are well-known problems in database theory and practice. Nevertheless, these problems receive new strength from the irregular nature of XML.

Irregular tree or forest structure XML data can be seen as node-labeled trees or forests; these trees, being commonly used for representing semistructured data, usually have a deeply nested structure, and are far from being well-balanced. Moreover, the same tag may occur in different parts of the same document with a different semantics, e.g., the tag name under person and the tag name under city.

The irregular and overloaded structure of XML documents influences cardinality estimation, since the location of a node inside a tree may determine its semantics, and, then, its relevance in operations like path and predicate evaluation. For example, consider the XML document shown in Figure 3.1.

The name element under person is defined as a complex element, containing the sub-elements fullname and gensname; the name element under city, instead, is

```
<root>
   <persons>
      <person>
         <name>
            <fullname> Caius Julius Caesar </fullname>
            <gensname> Julia </gensname>
         </name>
         <city> Roma </city>
      </person>
   </persons>
   <cities>
      <city>
         <name>
            <ancientname> Roma </ancientname>
            <currentname> Roma </currentname>
         </name>
         <nick> Caput Mundi </nick>
         <nick> Eternal City </nick>
      </city>
      <city>
         <name> New York </name>
         <nick> The Big Apple </nick>
      </city>
   </cities>
</root>
```

Figure 3.1: A sample XML document

defined as a simple element having a textual content. These elements, even though having the same tag, have a very different semantics, and the evaluation of any query operation starting from name elements should take this into account.

Non-uniform distribution of tags and values Past estimation models, in particular those for relational databases, were based on the assumption that data are uniformly distributed, e.g., the distribution of values into tuple fields is uniform. This hypothesis greatly simplifies the design of estimation models, and reduces the size of the statistics about data.

As shown by the relational case, this hypothesis is usually **false**: data tend to distribute in a non-uniform way, partly because the world being modeled is not uniform, and partly because of functional dependencies among attributes. Hence, size equations based on this assumption are used as *last-chance* equations, e.g., when no information is available about a given predicate.

In the context of XML, the irregular structure of XML data, together with their hierarchical tree-shaped nature, leads to the *non-uniform* distribution of tags and values in XML trees. XML non-uniformity is further strengthened by the presence of *structural* dependencies among elements. Consider, for example, the document shown in Figure 3.1. Elements with tag name can appear only as sub-elements of person or city elements; furthermore, these occurrences obeys different semantics, hence the distribution of name elements is far from being uniform.

3.2 Path and Twig Cardinality Estimation

Path and twig expressions are used in XQuery and in many other XML query languages for retrieving nodes from a XML tree, and for binding them to variables for later use.

The main difficulties in cardinality estimation for path and twig expressions come from the need to reduce the prediction errors induced by joins (paths and twigs are usually translated in sequences of joins), and, for twigs only, from the need to correlate results coming from different branches.

Consider, for example, the following query fragment:

```
for $p in input()//person,
    $n in $p/name,
    $c in $p/city
```

This query retrieves, for each person in the database, her name and her city. The twig being used in the query is composed by two branches: `person-name`, and `person-city`. Correlating results from these branches means correlating each person name with her correspondent city; without proper information, we can only relate each name element with each city element, hence producing a huge overestimation error.

3.3 Predicate Selectivity Estimation

The estimation of predicate selectivity is a well-known problem in database theory and practice. The most effective and accurate solutions rely on histograms for capturing the distribution of values in the data, and on the use of the uniform distribution when nothing is known about the data involved in the predicate.

In the context of XML, predicate selectivity estimation poses new challenges. First, XML data are usually distributed in a (very) non-uniform way, hence the use of the uniform distribution can lead to many potential errors. Second, the selectivity of a predicate such as $data(\$n)\ \theta\ value$ depends not only on θ and $value$, but also on a) the nodes bound to $\$n$, which may be heterogeneous, b) the semantics of those nodes (e.g., name under person is quite different from name under city), and c) the "region" of the document where those nodes appear.

Many existing prediction models, while very sophisticated and accurate, return raw numbers as result of the estimation. Raw numbers, denoting the cardinality of matching nodes in the data tree, do not carry sufficient information for the estimation of subsequent predicates being accurate, hence making the enclosing models not so accurate.

3.4 Groups

As noted about nested queries, XQuery misses explicit constructs for performing groupby-like operations.[1] Nevertheless, the let binder can be used for creating heterogeneous sets of nodes, hence for building, together with nested queries, groups and partitions. The let binder, unlike the for binder, accumulates each node returned by its argument into a set, which is then bound to the binding variable. For example,

```
for $c in input()//city,
let $n_list := $c/name,
```

returns, for each city, the list of its names (ancient as well as modern ones).

Estimating the cardinality of the let binder requires the system to a) estimate the number of distinct groups created, b) correlate each group to the variables on which it depends ($\$n_list$ depends on $\$c$), and c) estimate the distribution of nodes and values into each group. This information is necessary since the groups created by the let binder can be used as starting point for further navigational operations, as argument for aggregate functions or for predicates.

[1] We are aware of proposals, both public and private to the W3C XQuery Working Group, for extending XQuery with explicit group-by constructs. Due to lack of time, we cannot extend the framework to take these proposals into account.

The estimation of group cardinality is one of the missing points in current XML prediction model. For what is known to the author, no existing model for XML query languages faces this problem.

3.5 Nested Queries

XQuery, as many other XML query languages, is a *free nesting* language, where nested queries are primarily used for reshaping or regrouping elements. Since the result of nested queries may be the input for navigational or filtering operations in the outer query, predicting the size of nested query results requires the system not only to estimate the raw cardinality, but also to build temporary statistics for them.

Consider, for example, the following query:

```
for $p in input()//person,
    $n in for $nn in $p/name,
          return < name > data($nn/fullname) < /name >
return < card > {$p/city, $n } < /card >
```

This query returns the name and the city of each person in the database; name elements are retrieved by the inner query, which also changes thier structure. As a consequence, the estimation model should generate *on-the-fly* statistics for these newly created name elements. These statistics must resemble persistent data statistics, and they disappear once the external query is executed.[2]

[2]A smart size estimator may permanently store statistics about frequently used nested queries.

Chapter 4

State of the Art in XML Result Size Estimation

This Chapter discusses some estimation models for XML queries, and, in particular, those supporting twig queries. Early models, such as the model used in the LORE system [GMW99], are not presented.

4.1 Correlated Subpath Trees

In [CJK$^+$01] authors deal with the problem of estimating the number of matches of a twig query over tree structured data. Data trees and twig queries are represented in the same way as node-labeled trees, where leaf nodes are labeled with strings in $\Sigma*$ (Σ is an alphabet), while internal nodes are instead labeled with strings in $\Pi \subset \Sigma*$; as a consequence, queries with closure operators or wildcards (e.g., // and $*$) are not supported by the proposed models.

The main idea behind the paper is the extension of summarization and prediction techniques for substring selectivity [FKMS01] to the context of twig queries. Authors first define a summary structure, the *Correlated Subpath Tree*, hosting the most frequent subpaths of the data tree; any subpath in the CST is endowed with its frequency as well as with the hash signature of the set of nodes, which the path is rooted in. Hash signatures are used for correlating subpaths, hence for increasing the accuracy of the prediction. The CST can be pruned by defining a threshold for supbath frequency, and by discarding low frequency paths.

By leveraging on this statistic structure, authors propose four estimation models: pure MO (Maximal Overlap), MOSH (Maximal Overlap with Set Hashing), PMOSH (Piecewise MOSH), and MSH (Maximal Set Hashing). The proposed algorithms have a common structure, and are organized in three phases: path parsing, where a twig query Q is matched over the CST T' to produce a set of paths S; twiglet decomposition, where paths in S are combined to form a set of twiglets (i.e., very small twigs) S', and MO conditioning, where twiglets are combined to form the

original query Q. During the first step, path frequencies are retrieved from the CST, while during twiglet decomposition and MO conditioning twiglet and twig frequencies are computed with probabilistic formulae obeying the inclusion-exclusion principle.

The four proposed algorithms differ in the way the three phases are performed. For what concerns path parsing, pure MO, MOSH, and MSH algorithms match root-to-leaf paths in Q with the longest possible subpaths in T', while PMOSH first decomposes Q into segments, and then matches them against T.

For twiglet decomposition, pure MO algorithm just form twiglets consisting of single paths, while MOSH and PMOSH algorithms form twiglets by combining paths in S via the set hash signature; MSH distinguishes from MOSH and PMOSH because twiglets are formed from paths in Q and from their suffixes in T'.

MO conditioning is the only step where the four proposed algorithms behave exactly the same way.

Differences in the path parsing and the twiglet decomposition steps mainly affect the shape of the resulting twiglets: pure MO twiglets are just paths, as in Niagara's models [AAN01] ; MOSH twiglets are deep but often skinny, while PMOSH twiglets are bushy but often shallow; MSH twiglets, finally, are the result of the trade-off between deepness and bushiness.

Authors experiment the proposed algorithms on two datasets by varying the space reserved for statistics, the database size, and the query workload: three query workloads were considered, consisting respectively of trivial *positive* (e.g., non-empty) queries, non-trivial positive queries, and non-trivial *negative* (e.g., empty) queries. In any test, MOSH and MSH algorithms outperform the others in accuracy.

The main contribution of this paper is the identification of the need to explicitly store correlation information in XML statistics for obtaining good size predictions. On the other hand, the proposed models, while very accurate, deal with a very limited class of twig queries, and the way they can be extended to more general twig queries and to wider query sets appears unclear.

4.2 TIMBER's Models

In [WPJ02] authors propose two models for estimating the number of matches of a twig query Q over a XML tree T (these models are used in the TIMBER system [tim]). Authors represent twigs as rooted, node-labeled trees, where each edge denotes an ancestor/descendant relationship, and each node is labeled by a boolean combination of predicates from a set of basic predicates $\mathcal{P}$.

The proposed models rely on the notion of *position histogram*. For the purpose of building position histograms, each node in a document T is associated with a pair of integer numbers $(startpos(x), endpos(x))$, where $startpos(x)$ is the position of x in a pre-order visit of T, $startpos(x) < endpos(x)$, and, for each descendant y of x, $endpos(y) < endpos(x)$. This numbering scheme, which slightly differs from

existing numbering schemes for XML documents, is the *architrave* of the models, and ensures that the intervals associated to x and y have non-empty intersection if and only if x and y are in ancestor/descendant relationship.

Given this numbering scheme, and given and interval width h, an histogram is built for each predicate in $\mathcal{P}$. The histogram has *startpos* intervals on the x-axis as well as *endpos* intervals on the y-axis: each grid cell, then, contains the number of nodes in the corresponding part of the data tree satisfying the predicate.

Since no grid cell has non-zero node count under the diagonal (recall that positions satisfy the property: $startpos(x) < endpos(x)$), and since no intersection between intervals of nodes of different paths is allowed, each position histogram is sparse, and contains only $O(g)$ non-zero node count cells, where $g = n/h$ and n is the number of nodes of T, hence dramatically decreasing the space requirements of the models.

A crucial issue of the models is the choice of the set of basic predicates $\mathcal{P}$. Authors suggest to include in $\mathcal{P}$ any structural predicate, e.g., $elementtag = "faculty"$, as well as the most frequently referred value predicates: in particular, authors propose to choose exact match predicates for numeric leaf nodes, and prefix match predicates for string-based leaf nodes.

Once built the collection of position histograms, the first estimation model can be applied. Given a pattern (P_1, P_2), meaning "all nodes satisfying P_2 and being descendant of nodes satisfying P_1", the node count for the result can be estimated by finding, for each non-zero count cell in $Hist_{P_1}$, the cells in $Hist_{P_2}$ containing its descendants (ancestor-based evaluation), or by starting from $Hist_{P_2}$ and finding the cells in $Hist_{P_1}$ containing ancestors (descendant-based evaluation). The two evaluation techniques have linear complexity ($O(g)$).

The second estimation model is an evolution of the first one, and it exploits a kind of structural information for increasing the accuracy of the prediction; unlike the first model, however, this one can be used only when the schema information is available, and, in particular, when the ancestor predicate in a pattern (P_1, P_2) satisfies the *no-overlap* property:

$\forall x, y. P(x), P(y)$, x and y have no ancestor/descendant relationship

Given this property, which is satisfied, for instance, by `book` elements in a BibTeX-like XML tree, an upper bound of the node count of (P_1, P_2) is given by the number of nodes satisfying P_2 and participating in the join. This number can be estimated by using a new summary structure called *coverage histogram*.

A coverage histogram for a predicate P has grid cells on both the x-axis and the y-axis, and it is defined as follows:

$Cvg_p[i][j][m][n]$ = the number of nodes in cell (i, j) that are descendants of nodes in cell (m, n) satisfying P

Even though built on cells rather than on position intervals, coverage histograms are still sparse, and require $O(g)$ space.

The model based on coverage histograms is much more accurate than the model based on position histograms only; unfortunately, its applicability is limited, and, in particular, it cannot be exploited in recursive documents, where the two proposed models behave badly.

Moreover, the estimations are limited to ancestor/descendant paths, and it is not clear how they can be extended to complex twigs involving also parent/child relationships. Furthermore, the scalability of the proposed models is limited by the need to build a relatively high number of histograms, in particular for complex documents, e.g., the XMark dataset [SWK+01].

Finally, the model only deals with twig matching, hence ignoring critical issues such as iterators, binders, nested queries, etc.

4.3 Niagara's Models

In [AAN01] authors present the path expression selectivity estimation models employed in Niagara, a system for querying XML data dispersed over the Internet. The models can be used to compute the selectivity of path expressions of the form `a/b/.../f`, i.e., XPath patterns without closure operators (//) and inline conditions; moreover, the models cannot be applied to twigs. The models differ in the statistics they exploit, in the way these statistics are summarized, as well as in the estimation algorithms.

The first model is based on a structure called *path tree*. A path tree is a tree containing each distinct rooted path in the database; path tree nodes are labeled by the tags of the reached nodes in the database as well as by the number of such nodes.

To estimate the selectivity of a path p, p is matched against the path tree, and the frequency of the leaf nodes in the matching path is returned.

Since a path tree may have the same size as the database (e.g., when paths in the database are distinct from each other), summarization techniques should be applied to constrain the size of the path tree to the available main memory. For this purpose, authors describe four summarization techniques based on the deletion of low frequency nodes, and on their replacement by means of $*$-nodes. A $*$-node, thus, denotes a set of deleted nodes, and inherits their structural properties as well as their frequencies.

The summarization techniques differ in the way $*$-nodes are employed, hence they require different efforts for decreasing the size of the tree. In the *sibling-** technique, low frequency siblings are replaced by $*$-nodes, and decreasing the size of the tree by n nodes may require to delete up to $2n$ nodes; the *level-** technique, instead, introduces a $*$-node for representing all nodes deleted at the same level, hence returning a DAG (Direct Acyclic Graph) and requiring $n + l$ deletion operations (l is the tree height); the *global-** technique, then, uses only one $*$-node denoting all the deleted nodes, wherever they appear in the database, and it produces a cyclic graph

by deleting $n+1$ nodes; the *no-** technique, finally, does not replace deleted nodes with $*$-nodes, hence returning a forest without any replacement overhead.

The unsummarized path tree estimation algorithm applies to summarized path trees by allowing a path step to match a $*$-node; $*$-node summarization techniques, hence, are based on the assumption that any matching path is really present in the database (false positives), while the no-* technique conservatively assumes that any non-matching path is missing from the database (false negatives).

The second model is based on a more sophisticated statistic structure called *Markov table*. This table, implemented as an ordinary hash table, contains any distinct path of length up to m ($m \geq 2$), and its selectivity. Thus, the frequency of a path of length n can be directly retrieved from the table ($n \leq m$), or it can be computed by using a formula that correlates the frequency of a tag to the frequencies of its $m-1$ predecessors. As a result, the estimation formula exploits the *short memory* principle of a Markov process of order $m-1$.

As for path trees, the size of a Markov table may exceed the total amount of available main memory, hence summarization techniques are required. Authors propose three summarization techniques, which delete low frequency paths, and replace them with $*$-paths: in particular, deleted paths of length up to 2 are replaced, while longer paths are just removed. As in the case of path trees, these techniques differ in the way $*$-paths are used: in particular, the *suffix-** technique uses $*$-paths of the form $*$, $*/*$, and $A/*$, while the *global-** technique employs only $*$ and $*/*$ paths; the *no-** technique, finally, does not use $*$-paths at all.

Once the summarized Markov table is generated, path expression selectivity can be estimated by using the standard correlation formula and $*$-paths when needed (a $*$-path only estimation is rejected).

Authors tested the proposed models on synthetic and real XML document (e.g., the DBLP dataset), and compared them with the *pruned suffix tree* technique described in [CJK+01]. The Markov table technique showed the best overall accuracy results, and it is particularly suited for data having a lot of common structures; moreover, Markov tables can be tuned to the accuracy and storage needs by changing the value of m.

The proposed approaches are quite simple and effective: the Markov table technique, in particular, delivers an high level of accuracy (much more than the pruned suffix tree methods). Unfortunately, they are limited to simple path expressions, and there is no clear way to extend them to twigs or predicates.

4.4 StatiX Statistical Model

In [FHR+02] authors describe a methodology for collecting statistics about XML documents; the proposed approach is applied in LegoDB [BFH+02] for providing statistics about XML-to-relational storage policies, and, to a less extent, in the Galax system [gal] for predicting XML query result size.

The proposed approach is based on three main points: the massive use of schema information (in the form of XML Schema types); the storage of gathered statistics into *equi-depth* histograms; and the reuse and extension of existing XML validators for collecting raw statistics. As a consequence, the methodology applies to documents described by XML Schemas only, hence it is not appropriate for structurally unstable documents: despite this limitation, the methodology can be profitably applied to most common XML data, e.g., business information.

The StatiX approach aims to build statistics capturing both the irregular structure of XML documents and the *non-uniform* distribution of tags and values within documents. To this purpose, it relies on the schema associated to each document. Indeed, given a XML Schema description $\mathcal{S}$ describing a XML document $\mathcal{T}$, StatiX builds $O(m + n)$ histograms, where m and n are respectively the number of edges and nodes in the graph representation of $\mathcal{S}$. StatiX histograms fall into two categories: *structural* histograms, which describe the distribution and the correlation of non-terminal type instances, and *value* histograms, which, as in the relational case, represent value distribution of simple elements (i.e., elements whose content is a base value). Histograms are built by applying standard histogram creation techniques [PIHS96, PI97] to previously collected raw information; this information is gathered during document validation, by applying the following algorithm:

1. XML Schema types are labeled with unique IDs;
2. the document is validated, and, during the validation process, each type instance is assigned a unique ID, recorded into a sort of type ID extent, together with the ID and the type of its parent node;
3. once validation is complete, type ID extents and parent ID information are organized into structural histograms.

The statistics gathering algorithm seems to have linear complexity in the data size, even though authors do not report any result about complexity; moreover, the claim about the independence of statistics size from the data size seems only partially true, since data size still influences the number and the dimension of histogram buckets required for achieving good accuracy.

Given this algorithm, the StatiX system can tune statistics granularity by applying *conservative* schema transformations to the original XML Schema description, i.e., transformations preserving the class of described documents, and not introducing ambiguity.

Once described the statistics gathering algorithm, authors show its application in the context of the LegoDB system for producing “virtual” relational statistics about XML-to-relational storage schemas. Moreover, authors present experimental results on the prediction of XQuery query result size, but unfortunately they do not describe their estimation model.

Chapter 5

Thesis Contribution

The main contribution of this Thesis is a result size estimation model for XML queries: the model is described through size equations and estimation algorithms; moreover, experimental results about the space required by data statistics, and about the accuracy of the model are presented (see Chapter 13).

The novelty of the proposed model is twofold. First, the model is an instance of a generic *metamodel*, described in Chapter 10 (see also [Sar03]), which can be used by any prediction model for obtaining specific estimation services, such as predicate selectivity factor propagation, twig branch correlation, etc. Second, unlike existing estimation models, which are limited to path and twig queries, the proposed model deals with the full FLWR subset of XQuery, hence discarding recursive functions and universally quantified predicates only: in particular, the model deals with nested queries too, which requires the estimator to generate *on-the-fly* new statistics during query estimation.

Beside the result size prediction model, the Thesis also contains other minor contributions. In particular, Chapter 7 describes a logical query algebra for XML queries, which features decomposition rules for path and twig expressions, as well as three rewriting rules for nested queries (proofs are shown in Appendix B: for what is known to the author, rewriting rules for nested queries on XML data have not been presented in other query algebras.

Part II

The System

Chapter 6

System Overview

Xtasy is a database management system for XML semistructured data. The main purposes behind the design and implementation of the system are: the study of issues related to the management of *persistent* XML data; the study of the suitability of traditional database architectures and techniques to this context; and the study of result size estimation models for XML queries. As a consequence, objectives like efficiency in time and space, optimal performance, original indexing techniques, etc, were not strongly pursued during system development and implementation.

The target query language supported by Xtasy is the *long-awaited* W3C standard query language XQuery. Given the instability of its specification, development efforts were focused on a relatively stable core fragment of the language, namely FLWR queries without recursive functions and typing; despite these limitations, the chosen fragment is wide enough to allow one to study most of the problems related to XML query processing.

Xtasy is entirely implemented in Pure Java (for both Java 1.3.1 and Java 1.4.1). The choice of Java as development platform is motivated by the high portability of Java applications that, despite some minor GUI glitches, can be successfully executed on most operating systems; moreover, we found the Java language much easier to use and less prone to run-time errors than other object-oriented languages. Nevertheless, during the development of Xtasy we struggled with some severe limitations of the Java platform, in particular those related with the I/O libraries, which make Java not suitable for large-scale, data-intensive applications.

6.1 Architecture

As already stated, one key goal of the design and implementation of Xtasy was the experimentation of traditional DBMS architecture in the context of XML data management. Thus, the architecture of Xtasy is quite traditional, even though some minor tweaks were necessary to adapt the system to the nature of semistructured and XML data.

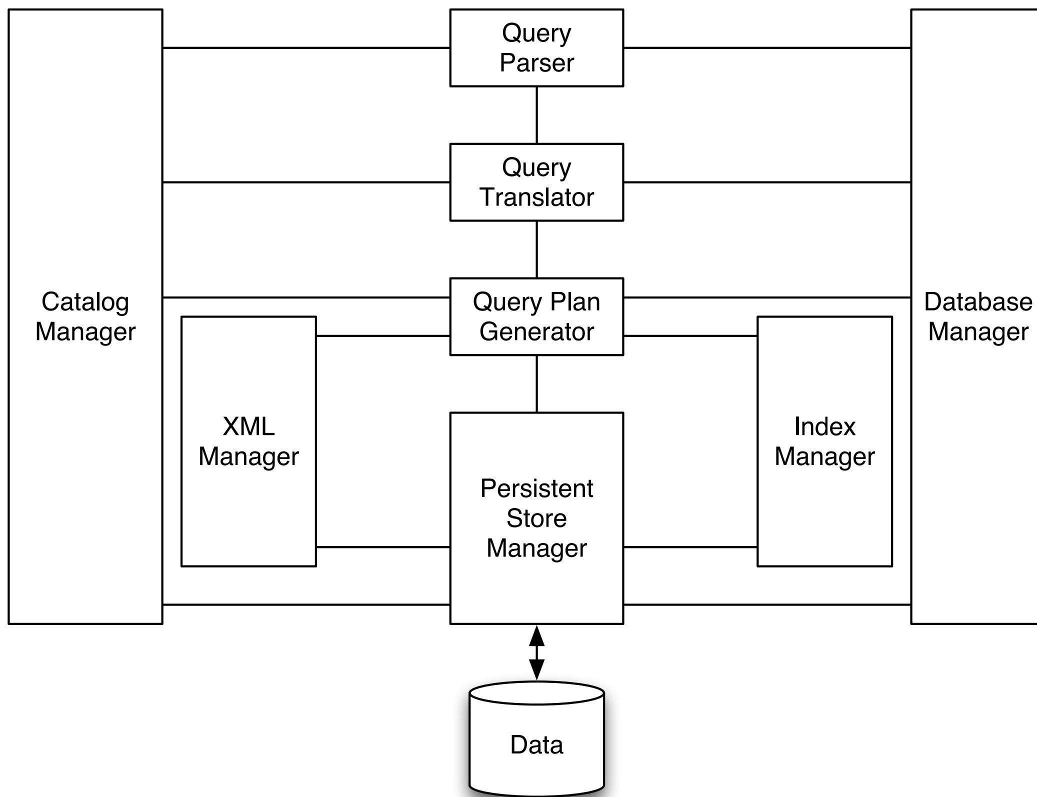


Figure 6.1: Xtasy architecture

The architecture of Xtasy is shown in Figure 6.1. At the top level, the Query Parser receives user queries and translates them into an intermediate representation, which forms the input for the Query Translator; the Query Translator applies common transformations, such as normalization and subexpression factorization, to its input, and then produces a logical query plan representing the query.

Given the logical query plan produced by the Query Translator, the Query Plan Generator generates a physical query plan.[1] This plan is a tree formed by physical operators, described in Chapter 9. During the execution of the query plan, physical operators access data through the Persistent Store Manager, which offers persistence services to all database modules.

Beside these modules, Xtasy contains other components offering services useful for the whole system. The Database Manager manages all operations concerning the creation, the opening, as well as the closure or the deletion of a database. The XML Manager provides facilities for translating the persistent representation of XML nodes into Java objects. The Catalog Manager hosts database statistics, and it plays a key role during result size estimation. The Index Manager manages the indexes used by the system.

Below the reader can find a more detailed description of modules and components of the system.

6.1.1 Query Parser

The Query Parser takes as input a user query, and returns an equivalent intermediate representation. Among the several kinds of intermediate representations for database queries ([PHH92] for instance), we chose a representation inspired by Moerkotte's query blocks [MH01]. An Xtasy query block is a box containing the list of all query clauses (for, let, where, return, and sortby/orderby), as well as sublists for most commonly accessed clauses, such as for and let; furthermore, the query block also contains a list of define clauses, which are generated by the Query Translator only, and are exploited for managing nested queries.[2] A query block can contain nested blocks, which can be further nested. The structure of a query block is shown in Figure 6.2.

The Query Parser was generated by using the JavaCC parser generator. The limitations in the class of grammars supported by the tool imposed some minor syntactical tweaks, with no real impact on the XQuery fragment supported by Xtasy.

[1] The current version of Xtasy contains a *pseudo-random* query plan generator, i.e., a plan generator that, given an input operation o and the set of available mappings $\mathcal{M}$ for o, picks randomly one element of $\mathcal{M}$.

[2] The semantics of the define clause is the same of the let clause; the only difference is that define is not part of the language.

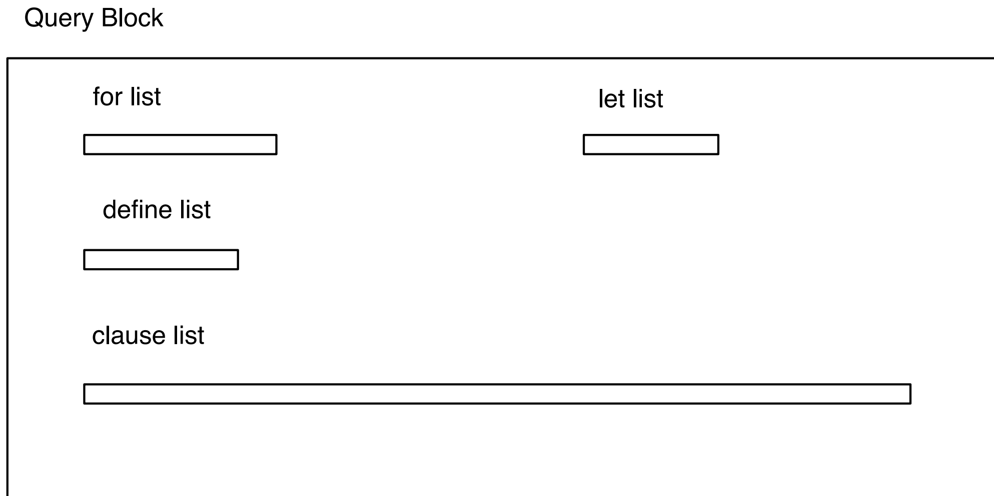


Figure 6.2: Diagram of a query block

6.1.2 Query Translator

The Query Translator takes a query block, applies some useful manipulations (e.g., common subexpression factorization), and then produces an equivalent algebraic expression. The Query Translator, first, identifies nested blocks contained in the main block, and isolates them by defining correspondent define clauses; a define clause binds a temporary variable to the inner query, which is then replaced in the containing clause by the temporary variable. Consider, for example, the following query fragment:

```
for $b in input()//book,
    $a in $b/author,
    $sold_book in for $bb in doc("amazon.xml")//book,
                      $aa in $bb/author
                  where $a = $aa
                  return $bb
```

The introduction of a define clause transforms the query as follows:

```
for $b in input()//book,
    $a in $b/author,
    $sold_book in $_var1
define $_var1 = for $bb in doc("amazon.xml")//book,
                    $aa in $bb/author
                where $a = $aa
                return $bb
```

As a result, nested queries occurring in any clause are moved to define clause, and then replaced by occurrences of the correspondent variable. The introduction of define clauses isolates nested queries, and allows the system to evaluate constant nested queries only once.

Then, the Query Translator replaces path expressions occurring in any clause but for and let with temporary variables, and introduces correspondent let clauses in the query block. As for the previous transformation, path expression relocation is a step toward the easy translation of queries.

The Query Translator, then, factorizes common subexpressions to avoid duplicate evaluation of the same expression. Before performing the translation step, the Query Translator also performs semantic checks on the query block: for example, it checks whether variables are properly defined and used in the query.

6.1.3 Persistent Store Manager

The Persistent Store Manager offers facilities for storing and accessing persistent data. These services are used during document indexing and storing, as well as during query evaluation. The Persistent Store Manager stores data in *heapfiles*; each heapfile can contain both fixed-length and variable-length records, where a record is just a string. Hence, heapfiles can be used for storing nearly any kind of data.

Access methods on heapfiles allows the system to scan heapfiles as well as to lookup and retrieve single records by means of their record identifiers. Operations on heapfiles are buffered, and the size of the buffer pool can be tuned to the needs of the specific application.

6.1.4 Database Manager

The Database Manager manages the creation, loading, indexing, closure, and deletion of Xtasy databases. An Xtasy database is a collection of data sources, where each data source is associated with an XML document; data source documents can be stored in the local filesystem, or dispersed over the Internet. Wherever they are located, documents are retrieved into the system, and, then, stored into the persistent store of Xtasy.

Xtasy poses no special constraint on XML documents, the only requirements being the *well-formedness* of documents.

As a consequence, an Xtasy database is a collection of data sources, where each data source is a well-formed document (possibly, an XML forest).

The database and data source creation process is organized in the following steps. During the first step, the user creates the database, and specifies where data should be stored in the accessible filesystem. Then, the user creates any single data source of the database by assigning them a name, and by specifying an URL for accessing documents; the specified documents are then fetched, stored into the persistent store,

as shown in Chapter 8, and indexed. Finally, the user specifies a schema for any single document, which is then exploited for building statistics about the document: if no schema is available, the system infers a minimal schema for the document.

It must be noted that database and datasource creation must be invoked before trying to execute queries on the corresponding documents.

6.1.5 XML Manager

The XML Manager provides facilities for translating the persistent representation of XML elements, attributes, and values into corresponding Java objects. For instance, the Java object associated with an element e describes its father, its children, its position in the document order, etc. In order to avoid scalability problems, Java objects are created when necessary, most of the database operations being directly executed on the persistent representation.

6.1.6 Catalog Manager

The Catalog Manager hosts system, database, and data source statistics. These statistics range from low-level data (e.g., the IDs of open files as well as their size) to database information (e.g., the number of documents into a data source and their size), and to document-related statistics, used during the query optimization process. The latter statistics are described in full detail in Chapter 11.

6.1.7 Index Manager

The Index Manager hosts the two kinds of indexes currently used by Xtasy: a persistent *centralized* B-tree index, and a persistent *segmented* B-tree index. The first one is used to store the parent/child relationship for elements of a same document (the Structural Index, as shown in Chapter 8): this relationship is stored into a single persistent B-tree. The latter one, instead, is a persistent B-tree organized in multiple fragments, which can be dynamically created and deleted; this kind of index is exploited during nested query execution for dynamically re-indexing newly created elements (more details can be found in Chapter 9).

Chapter 7

Query Algebra

This Chapter presents the logical query algebra used in Xtasy. The formal definitions of logical operators can be found in Appendix A.

7.1 Data Model and Term Language

The Xtasy query algebra employs a data model similar to the W3C XML Query Data Model [FMM$^+$03]. A data model instance is a *well-formed* XML document represented as an unordered forest of *node-labeled* trees, the global ordering being preserved by a special-purpose function *pos*; internal nodes are labeled with constants (tags and attribute names), and leaves with atomic values. Each internal node has a unique *object identifier* (oid) that can be accessed by the special-purpose function *oid*; an algebraic support operator ν is used to generate new oids and to refresh existing ones, hence allowing the algebra to support *copy* semantics as well as *reference* semantics operations.

Example 7.1.1 Consider the XML fragment shown below:

```
<book class = "OpSys">
   <author> Stuart Madnick </author>
   <author> John Donovan </author>
   <title> Operating Systems </title>
   <year> 1974 </year>
</book>
<book class = "Database">
   <author> Serge Abiteboul </author>
   <author> Peter Buneman </author>
   <author> Dan Suciu </author>
   <title> Data on the web: from relations to ... </title>
   <year> 2000 </year>
   <publisher> ... </publisher>
</book>
```

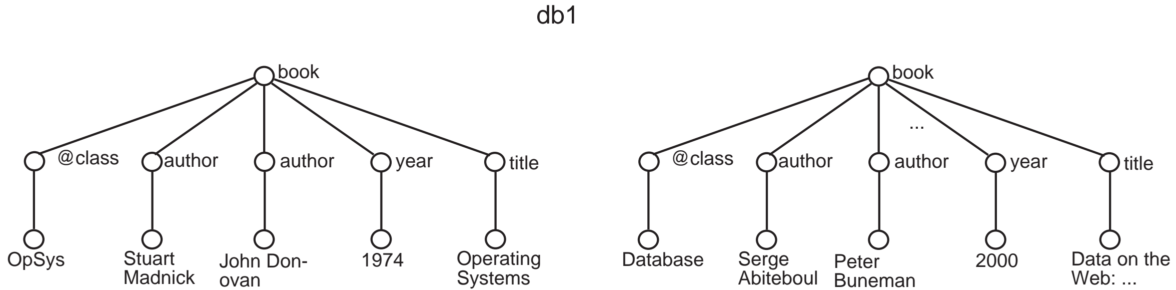


Figure 7.1: A data model instance

This fragment can be represented by the forest depicted in Figure 7.1 (oids are omitted).

■

Data model instances can be represented as terms conforming to the following grammar (quite close to the term grammar of [HP00]):

$$t ::= t_1, \ldots, t_n \quad | \quad (oid)label[t] \quad | \quad (oid)@label[v_B] \quad | \quad v_B$$
where *label* is defined by XML specifications and
$v_B \in Integer \cup String \cup Boolean \cup \ldots$

Example 7.1.2 The fragment shown in the previous example can be described by the following term:

```
book[
   @class["OpSys"],
   author["Stuart Madnick"],
   author["John Donovan"],
   title["Operating Systems"],
   year["1974"]
],
book[
   @class["Database"],
   author["Serge Abiteboul"],
   author["Peter Buneman"],
   author["Dan Suciu"],
   title["Data on the web: from relations to ..."],
   year["2000"],
   publisher["..."]
]
```

■

As shown by the term grammar, the data model represents only elements and attributes, hence discarding processing instruction, comments, etc; moreover, no special treatment is given to ID-type and IDREF-type attributes, as well as to linking mechanisms such as XLink. Though this is a significant limitation of the model, it helps to keep the model simple enough to allow easy proofs of rewriting rules.

Three auxiliary functions are defined on XML nodes: *label*, *pos*, and *oid*. $label(t)$ returns the label of the node t, while $pos(t)$ returns an integer denoting the position of t into the global ordering of its surrounding document ($pos(t_1, \ldots, t_n)$ just returns $pos(t_1), \ldots, pos(t_n)$). The *oid* function returns the oid of a given node, thus allowing the algebra to support structural as well as identity comparison among elements.

7.2 Algebra Operators

Xtasy algebra is an extension of common object-oriented and semistructured query algebras to XML. The starting points of the algebra are the YAT query algebra, described in [CCS98] and in more detail in [Sim99], as well as the algebra described in [CM93]; from those the Xtasy algebra borrows the idea of relational-like intermediate structures, hence extending to XML common relational and OO optimization strategies, as well as the presence of *border* operators, which insulate other algebraic operators from the technicalities of XML. The algebra provides two border operators, namely *path* and *return*, which respectively build up intermediate structures from XML documents and publish these structures into XML; these operators are quite different from those of YAT, since they allow direct evaluation of recursive XPath patterns, and cannot handle complex grouping and sorting operations as YAT *bind* and *tree*; these operations, instead, are performed by other algebraic operators, namely *GroupBy* and *Sort*.

In order to ensure the closure of the algebra, intermediate structures are themselves represented as node-labeled trees conforming to the algebra data model; this representation also allows one to apply useful optimizations to border operators. In addition to *path* and *return*, the Xtasy algebra provides *relational-like* operators such as *Selection*, *Projection*, *TupJoin*, *Join*, *DJoin*, *Map*, *Sort*, *TupSort*, and *GroupBy*.

There exist both *set-based* and *list-based* versions of the algebraic operators. For the sake of brevity, in the following Sections only the set-based versions will be presented.

7.2.1 *Env* structures

As already stated, algebraic operators manipulate relational-like structures. These structures, called *Env*, are very similar to YAT *Tab* structures [CDSS98] [Sim99], and contain the variable bindings collected during query evaluation. As in [CM93]

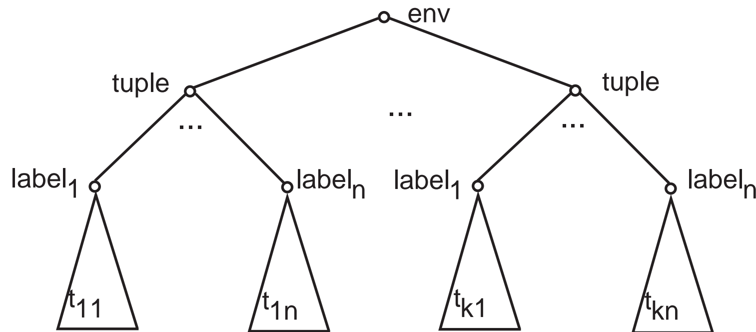


Figure 7.2: An intermediate structure

and YAT, *Env* structures allow one to define algebraic operators that manipulate sets of tuples, instead of trees; hence, common optimization and execution strategies (which are based on tuples rather than trees) can be easily adapted to XML without the need to redefine them.

An *Env* structure is a collection of *flat* tuples, each tuple describing a set of variable bindings. With the only exception of sorting operators, each algebraic operator manipulates unordered *Env* structures, e.g., tuple order is irrelevant.

As shown in Figure 7.2, *Env* structures are modeled as rooted trees; each tuple element describes a binding tuple, where $label_i$ are variable names and t_{ji} the corresponding values. The *env* structure depicted above can also be represented by the following term:

$env \equiv env[tuple[label_1[t_{11}], \ldots, label_n[t_{1n}]], \ldots, tuple[label_1[t_{k1}], \ldots, label_n[t_{kn}]]]$

In the following sections, set-based unordered *Env* structured will be denoted as[1]:

$e \equiv \{[label_1 : t_{11}, \ldots, label_n : t_{1n}], \ldots, [label_1 : t_{m1}, \ldots, label_n : t_{mn}]\}$, while ordered *Env* structures as:

$e \equiv \langle[label_1 : t_{11}, \ldots, label_n : t_{1n}], \ldots, [label_1 : t_{m1}, \ldots, label_n : t_{mn}]\rangle$

7.2.2 *Border* Operators

Xtasy query algebra provides two *border* operators: *path* and *return*; they are used for insulating other operators, such as *Join* and *DJoin*, from the nested structure of XML, and they play a key role in the whole algebra.

path

The main task of the *path* operator is to extract information from the database, and to build variable bindings. The way information is extracted is described by an *input filter*; a filter is a tree, describing the paths to follow into the database (and the way to traverse these paths), the variables to bind and the binding style,

[1]This notation is borrowed from YAT.

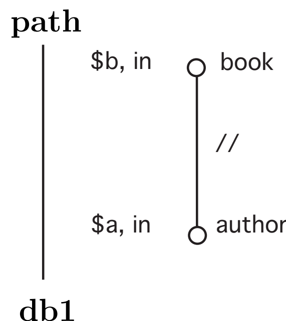


Figure 7.3: A simple *path* operation (the second graph represents the subscript of the *path* operation)

as well as the way to combine results coming from different paths. Input filters are described by the following grammar:

$$\begin{array}{rll} F & ::= F_1, \ldots, F_n & \text{conjunctive input filters} \\ & \mid\ F_1 \vee \ldots \vee F_n & \text{disjunctive input filters} \\ & \mid\ (op, var, binder) label[F] & \text{simple input filter} \\ & \mid\ \emptyset & \text{empty filter} \\ \text{where} & op \in \{/, //, _\} & \\ & var \in String \cup \{_\} & \\ & binder \in \{_, in, =\} & \end{array}$$

A simple filter $(op, var, binder)label[F]$ tells the *path* operator a) to traverse the current context by using the navigational operator *op*, b) to select those elements or attributes having label *label*, c) to perform the binding expressed by *var* and *binder*, and d) to continue the evaluation by using the nested filter F.[2]

The *path* operator takes as input a single data model instance and an input filter, and it returns an *Env* structure containing the variable bindings described in the filter. The following example shows a simple input filter and its application to a sample document.

Example 7.2.1 Consider the following fragment of XQuery query:

```
for $b in input()/book,
    $a in $b//author,
```

This clause traverses the path book//author into the sample document, binding each book and author element to $\$b$ and $\$a$, respectively. This clause can be translated into the following *path* operation (also shown in Figure 7.3):

$path_{(_,\$b,in)book[(//,\$a,in)author[\emptyset]]}(db1)$

■

[2] $op = _$ means that no navigation is required; $var = _$, instead, means that no variable is being bound; finally, $label = _$ is used to match any label.

Input filters provide a simple path language, containing common path operators such as / and //. No direct support, instead, is given to the resolution of ID/IDREF attributes, e.g., `a/b/@c=>/d` is represented by using joins. Moreover, input filters provide two binding styles (*in* and =), which directly correspond to Quilt and XQuery binders.

The following example shows the grouping binder =.

Example 7.2.2 Consider the following XQuery clause:

```
for $b in input()/book,
let $a_list := $b//author,
```

This clause traverses the path book//author; each book element is bound to $\$b$, and, for each book element, the whole set of author sub-elements is bound to $\$a_list$. This clause can be expressed by using the following *path* operation:

$path_{(_,\$b,in)book[(//,\$a_list,=)author[\emptyset]]}(db1)$

■

As shown by the filter grammar, multiple input filters can be combined to form more complex filters. Xtasy algebra allows filters to be combined in a *conjunctive* way, or in a *disjunctive* way. In the first case, the *Env* structures built by simple filters are joined together, hence imposing a product semantics; in the second case, partial results are combined by using an *outer union* operation. Therefore, disjunctive filters can be used to map XPath union paths into input filters (e.g., book/(author| publisher)), as well as more sophisticated queries; the use of outer union ensures that the resulting *Env* has a uniform structure, i.e., all binding tuples have the same fields. Disjunctive filters may also be used to express *unsafe* queries, as well as to map queries of potentially unsafe languages (e.g., [CG01]).

The following examples show the use of disjunctive filters.

Example 7.2.3 Consider the following XQuery clause:

```
for $b in input()/book,
    $p in $b/(author|publisher),
```

This clause binds the $\$p$ variable to publishers and authors of each book. It can be expressed by using the following *path* operation:

$path_{(_,\$b,in)book[(/,\$p,in)author[\emptyset]\vee(/,\$p,in)publisher[\emptyset]]}(db1)$

■

Due to the presence of disjunction, a precedence order among combinators has to be established: we chose to give precedence to disjunction, i.e., $f_1 \vee f_2, f_3 \vee f_4$ is evaluated as $(f_1 \vee f_2), (f_3 \vee f_4)$.

return

While the *path* operator extracts information from existing XML documents, the *return* operator uses the variable bindings of an *Env* to produce new XML documents. *return* takes as input an *Env* structure and an *output filter*, i.e., a skeleton of the XML document being produced, and returns a data model instance (i.e., a well-formed XML document) conforming to the filter. This instance is built up by filling the XML skeleton with variable values taken from the *Env* structure: this substitution is performed once per each tuple contained in the *Env*, hence producing one skeleton instance per tuple.

Output filters satisfy the following grammar:

$$\begin{array}{lll} OF & ::= & OF_1, \ldots, OF_n \\ & \mid & label[OF] \\ & \mid & @label[val] \\ & \mid & val \\ val & ::= & v_B \;\mid\; var \;\mid\; \nu var \end{array}$$

An output filter may be an *element constructor* ($label[OF]$), which produces an element tagged *label* and whose content is given by OF, an *attribute constructor* ($@label[val]$), which builds an attribute containing the value *val*, or a combination of output filters ($OF_1, \ldots, OF_n$). The second production needs further comments. The algebra offers two way to publish information contained into an *Env* structure: by copy (νvar) and by reference (var). Referenced elements are published as they are in query results; in particular, their object ids are not changed, thus allowing support for the definition and management of views over the database. Copied elements, instead, are published with *fresh* oids, hence losing the ties with their originating databases.

The following example shows the use of the *return* operator.

Example 7.2.4 Consider the following XQuery query:

```
for $b in input()/book,
    $t in in $b/title,
    $author in $b/author
return < entry > {$t, $author} < /entry >
```

This query returns the title and the authors of each book. This query can be represented by the following algebraic expression (also shown in Figure 7.4):

$$return_{entry[\nu\$t,\nu\$author]}(\\ \quad path_{(_,\$b,in)book[(/,\$author,in)author[\emptyset],(/,\$t,in)title[\emptyset]]}(db1))$$

■

The following example shows the use of the *return* operator to define a view over the database.

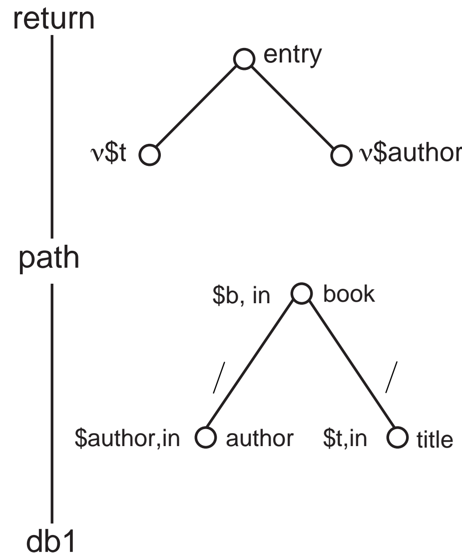


Figure 7.4: A query containing *return*

Example 7.2.5 Assume that you want to define a database view restricting the access to only those books published before 2001. By using a reference output filter this task can be accomplished by the following algebraic expression:

$$
\begin{aligned}
&return_{view[\$b]}(\\
&\quad \sigma_{\$y<2001}(\\
&\qquad path_{(_,\$b,in)book[(/,\$y,in)year[\emptyset]]}(db1)))
\end{aligned}
$$

■

7.2.3 Basic Operators

Xtasy algebra basic operators manipulate *Env* structures only, and perform standard operations. They resemble very closely their relational or object-oriented counterparts, thus allowing the query optimizer to employ usual algebraic optimization strategies. This class contains *Map*, *TupJoin*, *Join*, *DJoin*, *Selection*, *Projection*, *GroupBy*, *Sort*, as well as *Union*, *Intersection*, *Difference*, *OuterUnion*, and their list-based counterparts. In the following the most important operators will be presented.

Selection *Selection* σ takes as input an *Env* and a boolean predicate P, and returns a new *env* structure where binding tuples not satisfying P are missing. The predicate language of the Xtasy algebra is quite rich, offering existential as well as universal quantification over variables. These quantifications are required for easily translating universally quantified XQuery queries, and can be optimized by using

standard rewriting techniques [CKMP97]. The following example shows the use of the *Selection* operator.

Example 7.2.6 Consider the following XQuery query:

```
for $b in input()/book,
    $t in $b/title
where EVERY $a in $b/author SATISFIES
      $a/data() != "Vassilis Christophides"
return < entry > $t < /entry >
```

This query returns the title of each book not written by Vassilis Christophides. This query can be represented by the following algebraic expression:

$$return_{entry[\nu \$t]}(\sigma_{\forall \$\alpha \in \$a : \$\alpha \neq "VassilisChristophides"}(path_{(_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,=)author[\emptyset]]}(db1)))$$

The selection predicate compares the content of each author element with "Vassilis Christophides", and returns true if and only if each author element content is not equal to "Vassilis Christophides".

■

TupJoin $TupJoin \bowtie_P$ is the Xtasy counterpart of standard join operators. So, it takes as input two Env structures e_1 and e_2 as well as a boolean predicate P; it evaluates the predicate P over each pair of tuples $(t_1, t_2) \in e_1 \times e_2$, returning only the pairs satisfying P. The primary use of the $TupJoin$ operator is to combine *path* operations over independent data sources, and it is also introduced during query unnesting. The following example shows the typical use of $TupJoin$.

Example 7.2.7 Consider the following query fragment:

```
for $b in input()/book,
    $a in $b/author,
    $b1 in doc("amazoncatalog.xml")/book,
    $t in $b1/title
```

This query accesses two data sources, an internal one ($db1$) and an external one (`catalog.xml`). This fragment can be represented by the algebraic expression shown below (see also Figure 7.5).

$$(path_{(_,\$b,in)book[(/,\$a,in)author[\emptyset]]}(db1)) \bowtie_{true} (path_{(_,\$b1,in)book[(/,\$t,in)title[\emptyset]]}(extdb))$$

■

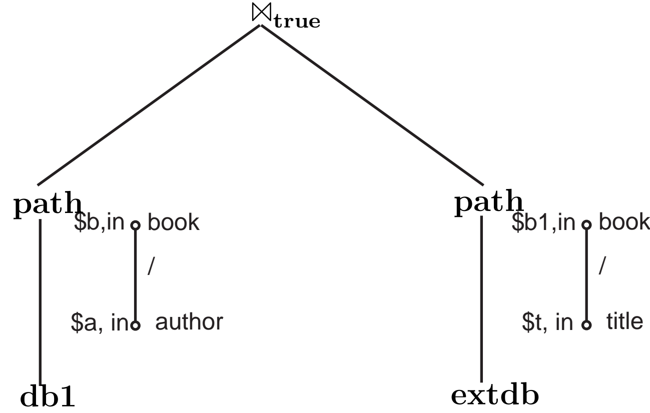


Figure 7.5: A query containing *TupJoin*

The previous example requires further comments. Unlike XQuery joins, Xtasy algebra joins are unordered, e.g., $e_1 \bowtie_P e_2 \equiv e_2 \bowtie_P e_1$; so, the previous algebraic expression can be rewritten as:

$$(path_{(_,\$b1,in)book[(/,\$t,in)title[\emptyset]]}(extdb)) \bowtie_{true} (path_{(_,\$b,in)book[(/,\$a,in)author[\emptyset]]}(db1))$$

This is a significant divergence from XQuery Formal Semantics, since XQuery Formal Semantics joins are (unless otherwise stated by the user) *non-commutative*, even on independent operands, i.e., $e_1 \bowtie_P e_2 \neq e_2 \bowtie_P e_1$. This divergence imposes the use of a *Sort* operation before the *return* operation in the translation of XQuery queries into algebraic expressions, as it will be shown in the next paragraphs; we chose this approach since we believe that the join efficiency improvement might compensate for the cost of the additional *Sort* operation.

Join Unlike *TupJoin*, which has a fixed tuple combination function, the *Join* $\bowtie_P^f$ operator takes as input a predicate P, two *Env* structures e_1 and e_2, and a combination function f. f is used to combine tuples in the resulting *Env* structure. As a consequence, *Join* is a higher order operator. The *Join* operator has been inserted to enhance the flexibility of the algebra, and to provide room for future extensions.

DJoin Unlike the *TupJoin* operator, which joins together two independent *Env* structures, the *DJoin* $< \cdot >$ performs a join between two *Env* e_1 and e_2, where the evaluation of e_2 may depend on e_1. This operator comes from object-oriented query algebras, and it is used to translate `for` and `let` clauses of XQuery and, in particular, to combine an inner nested block with the outer one.

The only way to evaluate a *DJoin* is to perform a nested loop among operands, hence one major goal of the optimization process is to transform, whenever possible, *DJoins* into more tractable *TupJoin* operations.

The following example shows the use of $DJoin$ during query translation.

Example 7.2.8 Consider the following query:

```
for $b in input()/book,
    $t in $b/title
where EVERY $a in $b/author SATISFIES
      $a/data() != "Vassilis Christophides"
      AND $b/publisher IN Q
return < entry > $t < /entry >
```

This query returns the title of each book not written by Vassilis Christophides, whose publisher is contained into the result of a nested query Q. This query can be translated into the following algebraic expression:

$$
\begin{aligned}
&return_{entry[\nu\$t]}(\sigma_{\forall\alpha\in\$a:\alpha\neq"VassilisChristophides"\wedge\$p\subseteq\$_var}(\\
&\quad (path_{(_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,=)author[\emptyset],(/,\$p,=)publisher[\emptyset]]}(db1)) < \\
&\qquad path_{(_,\$_var,=)_[\emptyset]}(Q) >)))
\end{aligned}
$$

■

Sort The *Sort* operator is used for dealing with the three sorting issues described in the Introduction: translating the sortby/orderby clause of XQuery (and similar clauses of other languages), preserving document order, and retaining join order. *Sort* takes as input an *Env* structure e and an ordering predicate P, and returns e sorted according to P. Ordering predicates are binary predicates defined on binding tuples, and used to impose the desired order; they have signature: $tuple \times tuple \rightarrow boolean$. The following example shows the use of *Sort* for translating sortby/orderby clauses.

Example 7.2.9 Consider the following query:

```
UNORDERED(
for $b in input()/book
return $b
sortby (title))
```

This query just returns the list of all books sorted by title. To translate this query, we need to define an appropriate predicate, as the following: $Pred(u, v) \equiv u.\$t < v.\$t$, where u and v ranges over *Env* tuples, and $\$t$ is bound to book titles. Thus, this query can be represented by the following algebraic expression:

$$
\begin{aligned}
&return_{\nu\$b}(\\
&\quad Sort_{u.\$t<v.\$t}(\\
&\qquad path_{(_,\$b,in)book[(/,\$t,in)title[\emptyset]]}(db1)))
\end{aligned}
$$

■

For preserving join order and order among elements a specialized version of *Sort* is used (called *TupSort*). *TupSort* takes as input an ordered list of variables $(\$x_1, \ldots, \$x_n)$, and an *Env* e; it returns e sorted according to the following ordering predicate:

$$
\begin{aligned}
<_{Tup} (\$x_1, \ldots, \$x_n)(u, v) = \ & lt(\$x_1) \vee \\
& (eq(\$x_1) \wedge lt(\$x_2)) \vee \ldots \vee \\
& (eq(\$x_1) \wedge \ldots \wedge eq(\$x_{n-1}) \wedge \\
& lt(\$x_n))
\end{aligned}
$$

where $lt(\$x_i) \equiv pos(u.\$x_i) < pos(v.\$x_i)$ and $eq(\$x_i) \equiv pos(u.\$x_i) = pos(v.\$x_i)$

This predicate allows the algebra to mimic the behavior of XQuery joins, whose semantics requires the system to retain the order in which variables are bound, unless the programmer qualifies the query with the keyword UNORDERED. The following example shows how *TupSort* can be used to preserve order among variables and XML elements.

Example 7.2.10 Consider the following query:

```
for $b in input()/book,
    $t in $b/title,
    $a in $b/author
return < entry > {$t, $a} < /entry >
```

XQuery semantics [DFF+03] prescribes that joins should be executed in an ordered fashion. Hence, a correct translation of this query should contain the *TupSort* operation $TupSort_{(\$b,\$t,\$a)}(\ldots)$, which sorts tuples in the *Env* structure according to the order specified in the query.

■

n-ary sortby/orderby clauses can be translated by using a n-ary ordering predicate, or by a combination of unary *Sort* operations. The following example shows the translation of such sortby/orderby clauses.

Example 7.2.11 Consider the query of Example 3.9 and assume that we want to return books ordered by title and by author.

```
UNORDERED(
for $b in input()/book
return $b
sortby (title, author))
```

The Xtasy algebra offers two ways to translate this query. The first augments the ordering predicate of the *Sort* operation: $(u.\$t < v.\$t) \vee (u.\$t = v.\$t \wedge u.\$a = v.\$a)$

where $a is bound to each book author; the second one breaks the orderby clause into two smaller clauses, as shown below:

$$
\begin{array}{l}
return_{\nu\$b}(\\
\quad Sort_{u.\$t<v.\$t}(\\
\quad\quad Sort_{u.\$a<v.\$a}(\\
\quad\quad\quad path_{(_,\$b,in)book[(/,\$t,in)title[\emptyset],(/,\$a,in)author[\emptyset]]}(db1))))
\end{array}
$$

■

GroupBy The *GroupBy* operator Γ of Xtasy takes as input an *Env* structure e, and partitions it according to the following definition: $\Gamma_{g;\ A;\ f_1;\ f;\ \theta}(e) = \{y.A \bullet [g : G] \mid y \in e, G = f(\{x \mid x \in e, f_1(x)\theta f_1(y)\})\}$ where $A \subseteq Att(e)$ and $g \notin Att(e)$.

As shown by the definition (very close to that of [CM93]), Xtasy *GroupBy* projects e tuples over A, and augments them with the corresponding groups G, obtained by applying the function f to the set of related tuples.

7.3 Optimization Properties

Three classes of algebraic equivalences can be applied to the Xtasy query algebra. The first class contains *classical* equivalences inherited from relational and OO algebras (e.g., *push-down* of *Selection* operations and commutativity of joins); the second class consists of path decomposition rules, which allow the query optimizer to break complex input filters into simpler ones; the third class, finally, contains equivalences used for unnesting nested queries. In the next sections, the following notation will be used:

- $Att(e)$ is the set of labels of an *Env* structure e;
- $FV(exp)$ is the set of free variables occurring in an algebraic expression exp;
- $symbols(of)$ is the set of node labels used in the output filter of.

7.3.1 Classical equivalences

Given the close resemblance of Xtasy algebraic operators to relational and OO operators, the Xtasy algebra supports a wide range of classical equivalences. In particular, *Selection*, *Projection*, *Map*, *TupJoin*, and even *return* are linear, so common *reordinability* laws can be easily applied to these operators.

Here follows a brief (and quite incomplete) list of supported algebraic equivalences.

$$\sigma_{P_1 \wedge P_2}(e) \equiv \sigma_{P_1}(\sigma_{P_2}(e))) \tag{7.3.1}$$

$$\sigma_{P_1}(\sigma_{P_2}(e)) \equiv \sigma_{P_2}(\sigma_{P_1}(e)) \tag{7.3.2}$$

$$\sigma_{P_1}((e_1) \bowtie_{P_2} (e_2)) \equiv (e_1) \bowtie_{P_1 \wedge P_2} (e_2) \tag{7.3.3}$$

$$(e_1) \bowtie_{P_1 \wedge P_2} (e_2) \equiv (\sigma_{P_1}(e_1)) \bowtie_{P_2} (e_2) \qquad \text{if } P_1 \text{ applies to } e_1 \text{ only} \tag{7.3.4}$$

$$(e_1) \bowtie_{Pred} (e_2) \equiv (e_2) \bowtie_{Pred} (e_1) \tag{7.3.5}$$

7.3.2 *path* decompositions

As already stated, *path* is the most important operator in the algebra, since it performs the basic tasks of evaluating path expressions and binding variables (both in an iterative fashion and in a grouping fashion). As a result, its efficiency affects the efficiency of the whole query processing. An efficient evaluation of *path* relies on the ability of the query compiler to simplify path expressions and to exploit existing access support structures, indexes in particular, which can dramatically speed up the evaluation. To this purpose the ability to decompose a complex filter into smaller ones is crucial, since it allows to match existing access structures as well as to replace expensive filters (i.e., filters involving //) with less expensive ones.

The Xtasy algebra provides three decomposition laws for *path* operations: the first works on the nested structure of a filter, while the remaining ones work on the horizontal structure of a filter.

Proposition 7.3.1 *Vertical decomposition of path operations*[3]

$$path_{(op,var,binder)label[F]}(t)$$

$$\equiv$$

$$path_{(_,_,in)env[(/,_,in)tuple[(/,_,in)var[(/,var,binder)_[F]]]]}(path_{(op,var,binder)label[\emptyset]}(t))$$

The following example shows how this decomposition law can be exploited during query optimization.

Example 7.3.2 Consider the following XQuery clause:

```
for $b in input()/library/book,
    $p in $b/(author|publisher),
    $t in $b/title,
    $y in $b/year,
```

This clause retrieves the sub-elements of each book element, binding them to a corresponding variable. This clause can be mapped into a *path* operation using the filter shown in Figure 7.6 (where linked edges denote disjunctive filters).

Assume now that a path index on library/book is available. To exploit the presence of this index, the previous *path* operation should be decomposed into a *path* operation with filter $(_, _, in)library[(/, \$b, in)book[\emptyset]]$ and a new *path* operation,

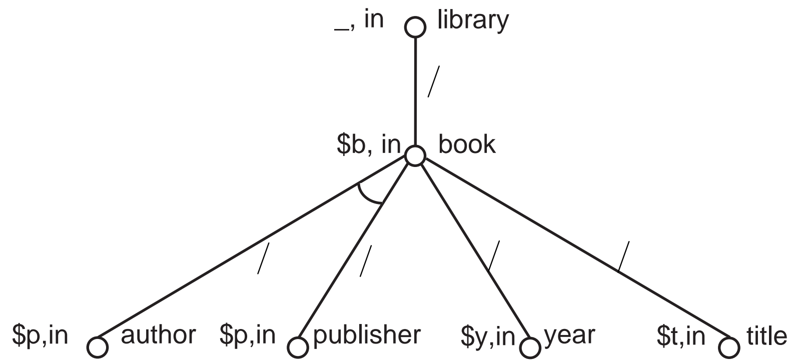


Figure 7.6: A complex input filter

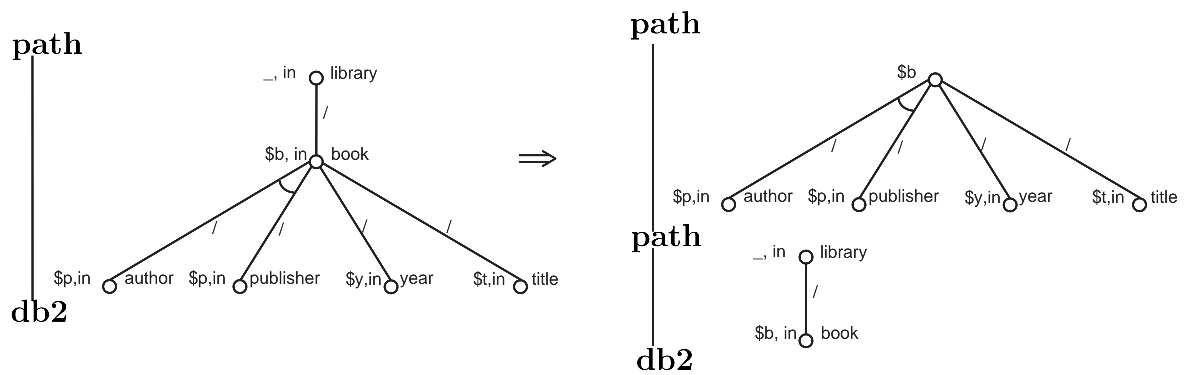


Figure 7.7: Vertical decomposition of *path*

which further explores the subtrees bound to $\$b$. This decomposition is shown in Figure 7.7.

∎

Proposition 7.3.3 *Horizontal decomposition of conjunctive input filters*

$$path_{f_1,\ldots,f_m}(t)$$

$$\equiv$$

$$(path_{f_1,\ldots,f_{i-1}}(t)) \bowtie_{true} (path_{f_i,\ldots,f_m}(t))$$

Proposition 7.3.4 *Horizontal decomposition of disjunctive input filters*

$$path_{f_1\vee\ldots\vee f_m}(t)$$

$$\equiv$$

$$(path_{f_1\vee\ldots\vee f_{i-1}}(t))OuterUnion(path_{f_i\vee\ldots\vee f_m}(t))$$

The Xtasy query algebra supports also decompositions of *path* via d-joins; those decompositions allow the compiler to translate XQuery for and let clauses by using arbitrarily complex filters, or simple filters only (as in XQuery Formal Semantics).

Proposition 7.3.5 *Dependent decomposition of input filters*[4]

$$path_{(op,var,binder)label[F]}(t)$$

$$\equiv$$

$$path_{(op,var,binder)label[\emptyset]}(t) < path_F(var) >$$

[3]The filter $(_,_,in)env[(/,_,in)tuple[(/,_,in)var[(/,var,binder)_[$ is used for navigating in the *Env* structure to the trees bound to *var*; this filter is required for ensuring the closure of the query algebra, and it should be considered as a formal artifact.

[4]This only holds for unordered collections.

7.3.3 Nested queries equivalences

This Section presents some equivalence rules that can be used to transform d-joins induced by nested queries into $TupJoin$ operations. These rules are not intended to be exhaustive, nor to be the most efficient transformations; they just rewrite $DJoins$ induced by nested queries into more tractable joins, and do not exploit *special-purpose* algebraic operators such as binary grouping. Before presenting our rewriting rules a brief introduction to the problem of query unnesting is necessary. In the reference paper about nested queries in object databases [CM94], the authors extend Kim's taxonomy of relational nested queries by defining three classification criteria: the *kind of nesting*, i.e., queries of type A, N, J, JA; the *nesting location*, i.e., the presence of nested queries into the *select*, $from$, or $where$ clause of OQL queries; the *kind of dependency*, i.e., the location of references to external variables. Referring to such classification, our rewriting rules apply to queries of type J (nested dependent queries returning sets of elements/attributes/values), and deal with the three kinds of dependencies (**projection** dependency, **range** dependency, and **predicate** dependency). In the next paragraphs these three kinds of dependencies will be discussed in more detail.

A typical query has the following structure:

$$
\begin{array}{l}
return_{of}(\\
\quad Sort_{P_1}(\\
\quad\quad \sigma_{P_2}(\\
\quad\quad\quad path_f(db))))
\end{array}
$$

The output filter of, the selection predicate P_2 as well as the input filter f can define dependencies with an outer query by referring (to) external variables. Depending on where these references are located into the nested query, **projection** dependencies (of), **range** dependencies (db), or **predicate** dependencies (P_2) may occur.

Predicate dependency Predicate dependencies occur when an external variable is referenced into the $where$ clause of the inner query, i.e., into the predicate P_2. Consider, for example, the following query returning authors and the list of their papers.

```
for $a in input()/library//author
return { $a, < publist > for $p in input()/library/*,
                                $aa in $p/author
                            where $aa = $a
                            return $p
            < /publist > }
```

This query contains a nested block (`for $p ... return $p`) that scans the papers and returns only those papers written by a given author (`where $aa = $a`).

This query can be represented by the following algebraic expression.

$$
\begin{array}{l}
return_{\nu \$a, publist[\nu \$_var]}(\\
\quad path_{(_,_,in)library[(//,\$a,in)author[\emptyset]]}(db) < \\
\quad\quad path_{(_,\$_var,=)_[\emptyset]}(return_{\$p}(\\
\quad\quad\quad \sigma_{\$aa=\$a}(\\
\quad\quad\quad\quad path_{(_,_,in)library[(/,\$p,in)_[(/,\$aa,in)author[\emptyset]]]}(db)))) >)
\end{array}
$$

In this kind of dependency the predicate P_2 has the form $Pred(\$X, \$Z, \$Y)$, where \$X are external variables, and \$Y and \$Z local variables. In order to remove this dependency (and the related $DJoin$ operation), we need to decompose $Pred(\$X, \$Z, \$Y)$ into $Pred_{Glob}(\$X, \$Z) \wedge Pred_{Loc}(\$Y, \$Z)$, i.e., to separate local variables from global ones, and to transform the $return$ filter.

Proposition 7.3.6 *Predicate Dependency*

$$
e_1 < (path_{(_,\$var,=)}(return_{of}(\sigma_{Pred(\$X,\$Z,\$Y)}(path_{f_1}(db))))) >
$$

$$
\equiv
$$

$$
\begin{array}{l}
\pi_{\overrightarrow{Att(e_1)};\ \$var}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(e_1 \ltimes_{Pred_{Glob}(\$X,\$Z)} \\
\quad (path_{f'}(return_{of'}(\sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db)))))))
\end{array}
$$

where

- $of' = _nested[_result[of], _env[z_1[\$z_1], \ldots, z_k[\$z_k]]]$
- $f' = (_, \$_n, in)_nested[(/, \$_res, =)_result[\emptyset], (/, _, in)_env[(/, _, in)z_1[(/, \$z_1, =)_[\emptyset], \ldots, (/, _, in)z_k[(/, \$z_k, =)_[\emptyset]]]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$, $\$z_1, \ldots, \$z_k \notin Att(e_1)$, $\$Y \notin Att(e_1)$, $z_1, \ldots, z_k \notin symbols(of)$

By applying this transformation the predicate dependency is brought out of the inner query, hence the previous algebraic expression can be rewritten as follows.

$$
\begin{array}{l}
return_{\nu \$a, publist[\nu \$_var]}(\\
\quad \pi_{\$a;\ \overrightarrow{\$_var}}(unnestvar_{\$_var}(rename_{\$_res \rightarrow \$_var}(\\
\quad\quad path_{(_,_,in)library[(//,\$a,in)author[\emptyset]]}(db) \ltimes_{\$a=\$aa} \\
\quad\quad path_{(_,\$_n,in)_nested[(/,\$_res,=)_result[\emptyset],(/,_,in)_env[(/,_,in)aa[(/,\$aa,=)_[\emptyset]]]]}(\\
\quad\quad\quad return_{_nested_[result[\$p],_env[aa[\$aa]]]}(\\
\quad\quad\quad\quad path_{(_,_,in)library[(/,\$p,in)_[(/,\$aa,in)author[\emptyset]]]}(db)))))))
\end{array}
$$

Range dependency In this form of dependency, the input filter of the inner query is applied to variables coming from the outer query. Consider, for example, the following query associating each paper with the list of its Italian authors.

```
for $p in input()/library/*
return < italianrd > {$p, for $a in $p/author
                          where data($a/country) = "Italy"
                          return $a }
       < /italianrd >
```

This query contains an inner block (`for $a ... return $a`) retrieving, for each given paper $\$p$, the list of Italian authors (if any). This query can be translated as follows.

$$
\begin{array}{l}
return_{italianrd[\$p,\$_var]}(\\
\quad path_{(_,_,in)library[(/,\$p,in)_[\emptyset]]}(db) < \\
\quad path_{(_,\$_var,=)_[\emptyset]}(\\
\quad\quad return_{\$a}(\sigma_{\$c="Italy"}(\\
\quad\quad\quad path_{(/,\$a,in)author[(/,\$c,=)country[\emptyset]]}(\$p)))) >)
\end{array}
$$

Object-oriented rewriting rules for range dependencies are based on the use of *type extents*; in particular, if the domains of the external variables referenced by the inner block are covered by type extents, their references are replaced by scans over these extents, and results are then combined through object equality predicate. Such transformations cannot be applied to data without extents, therefore we rely on a different rewriting techniques, whose main idea is to **copy** the left part of the $DJoin$ into the nested query, hence transforming it into a constant block, and then to combine results by using an equality predicate.

Proposition 7.3.7 *Range dependency*

$$e_1 < (path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_P(path_{f_1,\ldots,f_k}(\$x_1,\ldots,\$x_k))))) >$$

$$\equiv$$

$$
\begin{array}{l}
\pi_{\overrightarrow{Att(e_1)};\ \$var}(unnestvar_{\$var}(rename_{\$_res\rightarrow\$var}(e_1 \bowtie_{\$X=\$X'} \\
\quad (path_{f'}(return_{of'}(\sigma_P(path_{f''}(e_1)))))))
\end{array}
$$

where

- $f'' \equiv (_, \$_tuple, in)tuple[(/,_,in)x_1[(/,\$x_1,=)_[f_1],\ldots,(/,_,in)x_k[(/,\$x_k,=)_[f_k]]]]$
- $of' \equiv _nested[_result[of], _env[x_1[\$x_1],\ldots,x_k[\$x_k]]]$
- $f' \equiv (_,\$_n,in)_nested[(/,\$_res,in)result[\emptyset], (/,_,in)_env[(/,_,in)x_1[(/,\$x_1,=)_[\emptyset]],\ldots,(/,_,in)x_k[(/,\$x_k,=)_[\emptyset]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1,\ldots,f_k) \cap Att(e_1) = \emptyset$, $FV(P) \cap Att(e_1) = \emptyset$, $FV(db) = \$X \subseteq Att(e_1)$, $x_1,\ldots x_k \notin symbols(of)$

By applying Proposition 7.3.7, the previous query can be rewritten as follows.

$$
\begin{array}{l}
return_{italianrd[\nu\$p,\nu\$_var]}(\\
\quad \pi_{\$p;\ \overrightarrow{\$var}}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(\\
\qquad path_{(_,_,in)library[(/,\$p,in)_[\emptyset]]}(db)) \bowtie_{\$p=\$p} \\
\qquad path_{(_,\$_n,in)_nested[(/,\$_res,in)_result[\emptyset],(/,_,in)_env[(/,_,in)p[(/,\$p,=)_[\emptyset]]]]}(\\
\qquad\quad return_{_nested[_result[\$a],_env[p[\$p]]]}(\sigma_{\$c="Italy"}(\\
\qquad\qquad path_{(/,\$_tuple,in)tuple[(/,_,in)p[(/,\$p,=)_[(/,\$a,in)author[(/,\$c,=)country[\emptyset]]]]]}(\\
\qquad\qquad\quad \pi_{\$p}(e_1))))))))
\end{array}
$$

Projection dependency In this form of dependency the output filter of the inner block refers to external variables. As these variables may be deeply nested into complex XML skeleton and mixed with local variables (user abruptness has no limits), the output filter cannot be decomposed into a local part and a global one. A rule to unnest such dependencies is based on the *copy&join* technique used for range dependencies, as well as on the introduction of cross products. Therefore, the unnested expression may be much more expensive than the nested one, hence making this transformation not convenient. For the sake of completeness, the next Proposition shows this (almost useless) unnesting rule.

Proposition 7.3.8 *Projection dependency*

$$
e_1 < (path_{(_,\$_var,=)_[\emptyset]}(return_{of(\$X)}(\sigma_P(path_{f_1}(db))))) >
$$

$$
\equiv
$$

$$
\begin{array}{l}
\pi_{Att(e_1);\ \overrightarrow{\$_var}}(unnestvar_{\$_var}(rename_{\$_res \rightarrow \$_var}(e_1 \bowtie_{\$X=\$X} \\
\quad path_{f'}(return_{of'}(\pi(\$X)(e_1) \bowtie_{true} (\sigma_P(path_{f_1}(db)))))))
\end{array}
$$

where

- $of' \equiv _nested[_result[of], _env[x_1[\$x_1], \ldots, x_k[\$x_k]]]$
- $f' \equiv (_, \$_n, in)_nested[(/, \$_res, in)result[\emptyset], (/, _, in)_env[(/, _, in)x_1[(/, \$x_1, =)_[\emptyset]], \ldots, (/, _, in)x_k[(/, \$x_k, =)_[\emptyset]]]]$

if $FV(of) \cap Att(e_1) \neq \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $FV(P) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$, $x_1, \ldots x_k \notin symbols(of)$

7.4 Expressive Power

In this section a brief overview of the expressive power of the Xtasy algebra will be given. First, a relational completeness result will be shown, and then we will characterize the class of XQuery queries that can be represented in the algebra.

7.4.1 Relational Completeness

Current XML query languages can be used to query both irregular data and regular data, such as relational databases. This usually happens in integration systems, which provide a uniform XML view of multiple heterogeneous data sources. Therefore, it is important to show that the Xtasy algebra can be used to query relational data sources.

Proposition 7.4.1 *Completeness for relational algebra with aggregates*

There exists an encoding scheme $\mathcal{M}$ of relational data into XML data, and an encoding scheme $\mathcal{M}'$ of relational algebraic expressions extended with aggregation into Xtasy algebraic expressions, such that for any relational database db and for any correct relational algebraic expression Q on db:

$\mathcal{M}(Q(db)) \equiv \mathcal{M}'(Q)(\mathcal{M}(db))$

7.4.2 Representation and Translation of XQuery Queries

As in object-oriented query languages, XQuery query translation into algebraic expression is performed in two phases. During the first phase, common syntactical transformations are applied to the query tree; in particular:

1. any binder occurring in the where clause (e.g., `SOME $y IN ...`) is moved to the for clause, and corresponding quantified predicates are introduced in the where clause;
2. any path expression occurring free in the where clause, in the sortby/orderby clause, or in the return clause is bound to a variable, and the corresponding binder is introduced in the for - let clauses;
3. nested queries, occurring in any clause, are bound to variables and corresponding binders are introduced in the define clause (similar to the define clause of GOM);
4. finally, common subexpressions are factorized.

The following example shows the transformations applied to a sample query.

Example 7.4.2 Consider the following query:

```
for $b in input()/book,
    $t in $b/title
where EVERY $a in $b/author SATISFIES
      $a\data() != "Vassilis Christophides"
      AND $b/publisher IN Q
return <entry> $t </entry>
```

This query returns the title of each book not written by Vassilis Christophides, whose publisher is contained into the result of a nested query Q. By applying the above transformations, this query is transformed as follows:

```
FOR $b in book,
    $t in $b/title,
    $a in $b/author
LET $p = $b/publisher
DEFINE $_var = Q
WHERE EVERY $a SATISFIES $a != "Vassilis Christophides"
      AND $p IN $_var
RETURN <entry> $t </entry>
```

■

After the first phase, for and let clauses are examined to build *filter-like* path trees, obtained by transforming each path expression into a filter-like path, and then by merging together paths referring to the same document. Next, the optional sortby/orderby clause is scanned to build an ordering predicate used in the *Sort* operator, and finally, the return clause is scanned in order to build corresponding output filters.

Thus, the output of this phase is a query satisfying the following grammar:

```
Q ::= BWSR
B ::= (F|D)+
F ::= for input_filter do
D ::= define var = Q do
W ::= where BoolExp do
S ::= order_by sort_criteria
R ::= return output_filter
```

Given a query conforming to the previous grammar, its algebraic representation is obtained by applying the following translation scheme.

$[\![BWSR]\!]\rho \equiv [\![R]\!]([\![S]\!](TupSort_{(varlist)}([\![W]\!]([\![B]\!]\rho))))$

$[\![R]\!]\rho \equiv [\![return\ \ output_filter]\!]\rho \equiv return_{output_filter}(\rho)$

$[\![S]\!]\rho \equiv [\![order_by\ \ sort_criteria]\!]\rho \equiv Sort_{sort_criteria}(\rho)$

$[\![W]\!]\rho \equiv [\![where\ \ BoolExp]\!]\rho \equiv \sigma_{BoolExp}(\rho)$

$[\![B]\!] \equiv [\![b_1, \ldots, b_n]\!]\rho \equiv [\![b_1]\!]\rho < [\![b_2]\!]\rho < \ldots < [\![b_n]\!]\rho > \ldots >>$ where:

- $[\![for\ \ input_filter\ \ do]\!]\rho \equiv path_{input_filter}(\rho)$ and
- $[\![define\ \ var = Q\ \ do]\!]\rho \equiv path_{(_,var,=)_[\emptyset]}([\![Q]\!]\rho)$

Query qualified with the UNORDERED keyword are translated by just removing the *TupSort* operator.

Chapter 8

Storage Schema

Xtasy storage scheme is based on the mix of object-oriented and relational storage techniques for XML data with techniques apt to support data updates. Element and attribute nodes, thus, are clustered according to their names; moreover, each XML node is endowed with a *physical*, persistent, unique identifier (EID), mainly used for retrieving elements and attributes from the persistent store, and with a *structural* unique label used for checking the ancestor/descendant relationship; the latter labeling scheme is inspired by the seminal work of Cohen et al. on labeling schemes for dynamic and persistent tree data [CKM02]. In the following Sections these aspects will be described in detail.

8.1 Node Clustering

Given a XML tree $\mathcal{T}$, its representation on persistent store is obtained by clustering element and attribute nodes according to their names: the node set of $\mathcal{T}$ is partitioned into as many groups as the number of distinct tags and attribute names, e.g., book, author, etc, and each group is stored into a distinct file, so the system creates as many files as groups.

This approach clearly leads to the generation of a high number of files, which in turn may lead to a potential space and time overhead[1]; despite these issues, storing each group into a distinct file allows the system to execute navigational operations such as $//l$ by just scanning the file associated to tag l, which is not feasible with a logical only clustering (e.g., Xyleme's tag indexes [ACV+00]).

Groups of nodes are stored by means of *heapfiles*: heapfiles are just heaps of records, where a record is any char-delimited string. Heapfiles, then, can store both fixed-length and variable-length records. As a result, the system creates:

- one heapfile per tag name, e.g., the heapfile `*book*` for book elements;
- one heapfile per attribute name, e.g., the heapfile `*class*` for class attributes;

[1] A study by Arnaud Sahuguet [] suggests that complex schemas have no more than 100-200 different tags, and perhaps 5-600 different attributes. Hence, the number will not be too high.

- one heapfile for storing values, i.e., the heapfile `**value**`.

The system associates a unique fileID to each heapfile (for obvious reasons); this, together with the encapsulation of tag and attribute names into heapfile names, allows the system to use integers in place of tag and attribute names, hence introducing a very modest form of data compression.

Element and attributes are represented as fixed-length record containing the following fields:

1. the structural labels of the node (see Section 8.3);
2. the pointer to the father node.

Node pointers are represented by EIDs (Element IDs), where the EID of a node n is a pair containing the fileID of the heapfile containing n and the record identifier of n; each XML node, hence, has its own unique EID, which can be used for directly accessing the node and for inspecting its *type* (element, attribute, or value): in particular, EIDs of element and attribute nodes denote also their tags or names, thus filtering a list of EIDs according to a given tag or name requires no persistent storage access.

Value nodes are, instead, represented as variable-length records containing:

1. the structural labels of the node;
2. the father EID;
3. the string representation of the value itself.

The choice of keeping values separate from their enclosing elements and attributes allows an easy representation of *mixed content* element, as shown by the following example.

Example 8.1.1 Consider the following small XML fragment:

```
<bib>
   <book class = "OpSys">
      <author> Stuart Madnick </author>
      <author> John Donovan </author>
      <title> <cap> O </cap> perating <cap> S </cap> ystems </title>
      <year> 1974 </year>
   </book>
</bib>
```

This fragment, shown in its tree representation in Figure 8.1, contains a title element with mixed content (`<title> ... <cap> ... </cap> ... </title>`), and it can be stored as depicted in Figure 8.2.

■

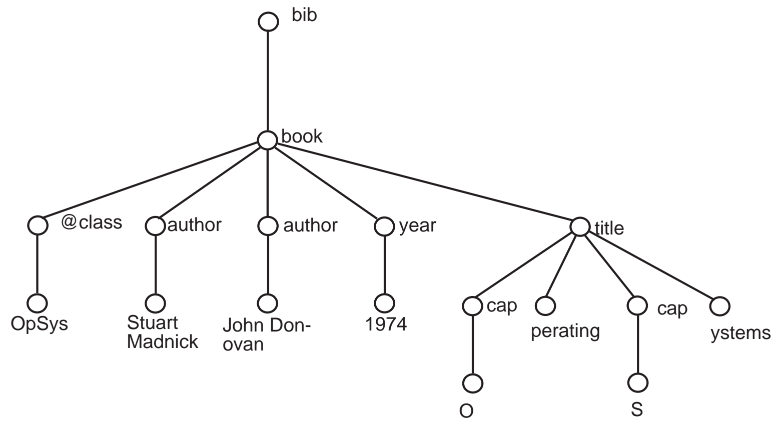


Figure 8.1: A XML tree

bib:: 1

1	18	<null,null>

@class:: 3

1	2	<2,1>

year:: 5

7	8	<2,1>

cap:: 7

9	10	<6,1>
12	13	<6,1>

book:: 2

1	17	<1,1>

author::4

3	4	<2,1>
5	6	<2,1>

title:: 6

9	15	<2,1>

value:: 8

1	<3,1>	OpSys
3	<4,1>	Stuart Mad...
6	<4,2>	John Don...
7	<5,1>	1974
9	<7,1>	O
11	<6,1>	perating
12	<7,2>	S
14	<6,1>	ystems

Figure 8.2: Storage schema for a XML tree (each table is endowed with its name and its file ID, e.g., **bib::1**)

Structural Index

<1,1>	<2,1>
<2,1>	<3,1>,<4,1>,<4,2>,<5,1>,<6,1>
<3,1>	<8,1>
<4,1>	<8,2>
<4,2>	<8,3>
<5,1>	<8,4>
<6,1>	<7,1>,<8,6>,<7,2>,<8,8>
<7,1>	<8,5>
<7,2>	<8,7>

Figure 8.3: Structural Index for a XML tree

8.2 The Structural Index

Though back pointers are sufficient to reconstruct the structure of the whole tree, evaluating path expressions with the only aid of back pointers is very expensive. For this purpose, Xtasy stores the parent/child relationship in the Structural Index, which is just a $B^+ - tree$[2] associating each element EID with the EIDs of its children nodes.

The use of a $B^+ - tree$ instead of more complex index structures has a threefold motivation. First of all, the $B^+ - tree$ is a simple and well-known data structure that can be easily managed; second, the $B^+ - tree$ allows a better handling of updates than IR-like data structures, and it can be easily adapted to support XML update languages [TIHW01]; finally, the $B^+ - tree$ allows a relatively easy dynamic indexing of elements created during nested query evaluation, which is a *must* in the case of free nesting languages as XQuery. These advantages are counterbalanced by a significant space overhead w.r.t. structures combining IR and compression techniques.

The following example shows the Structural Index for the XML fragment of Example 8.1.1.

Example 8.2.1 Referring to the previous XML fragment, its Structural Index is shown in Figure 8.3.

■

8.3 Structural Labeling

As already seen, each node x in a document $\mathcal{T}$ is associated an EID $eid(x)$, which is used for retrieving the record representing x from the persistent store as well as

[2]The $B^+ - tree$ was implemented from scratch in Pure Java.

for inspecting the kind of x (i.e., element, attribute, or value), and also, in the case of element and attribute nodes, its name.

EIDs are very useful, but they do not meet two important requirements: first, given two nodes x and y, it is not possible to test whether they are in ancestor/descendant relation by only using $eid(x)$ and $eid(y)$; second, the order relation among EIDs is not compatible with the document order.

Since any practical labeling scheme for XML data should meet these two requirements, and since EIDs are still necessary and useful, each node x must be endowed with a supplementary label.

There exist many labeling schemes for static XML documents satisfying these requirements: most of them are based on *range* or *prefix* schemes, where nodes are endowed with positional intervals, or with *prefix-free* strings.

Instead of using a labeling scheme for static data, we prefer to exploit a scheme for dynamic data, i.e., a scheme robust wrt updates. The rationale behind this choice is the will to build the grounds for an evolution of the present system supporting the management of dynamic data in a distributed context.

The labeling scheme we adopt is inspired by the schemes described in [CKM02]. In particular, the scheme associates each node x with a closed interval $[\alpha(x), \beta(x)]$ such that $\beta(x) - \alpha(x) + 1$ is the dimension of the subtree rooted in x; intervals are assigned by associating the root with $[1, N]$, where N is the dimension of the whole tree, and, given a node x with children $y_1, \ldots, y_k$, by splitting $[\alpha(x), \beta(x)]$ in k distinct and contiguous subintervals, such that $\beta(y_j) - \alpha(y_j) + 1$ is still the dimension of the subtree rooted in y_j (in a future extension of the system, $\beta(y_j) - \alpha(y_j) + 1$ will be the *estimated* size of the subtree rooted by y_j).

This labeling scheme satisfies the requirements described above. Indeed, the following properties hold.

Property 8.3.1 *Given two XML nodes x and y in $\mathcal{T}$, x is an ancestor of y if and only if $\alpha(x) \leqslant \alpha(y) \leqslant \beta(y) \leqslant \beta(x)$.*

Property 8.3.2 *Given two XML nodes x and y in $\mathcal{T}$, x precedes y in the document order of $\mathcal{T}$ if and only if $\alpha(x) \leqslant \alpha(y) \leqslant \beta(y) \leqslant \beta(x)$ or $\beta(x) \leqslant \alpha(y)$.*

The following example shows the labels associated with nodes in the XML fragment of Example 8.1.1.

Example 8.3.3 Consider the XML fragment of Example 8.1.1. The nodes of this fragment are labeled as shown in Figure 8.4

∎

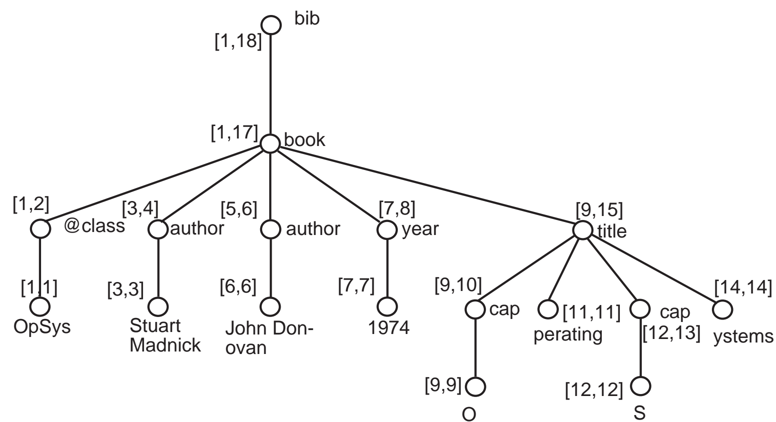


Figure 8.4: A labeled XML tree

Chapter 9

Physical Operators

This Chapter presents the physical query algebra used in Xtasy.

9.1 General Concepts

Xtasy physical operators are based on the iterative execution model [Gra93], widely used in many database systems; differently from the temporary relation model, the iterative model requires that each operator works on a single *data granule* per time, hence decreasing the memory requirements and enhancing the scalability of the system.

Given the choice of the iterative model, Xtasy operators feature a common interface, formed by five methods: open, next, close, isDone, and introduce. open is used for passing initial parameters to operators and for preparing them for the execution; next is used, instead, for obtaining a new data granule from operators; close is invoked at the end of query plan execution for releasing allocated resources, e.g., internal buffers; isDone checks if the task of the operator is finished; introduce, finally, is used for printing information about the operator, i.e., its full signature, in the GUI of Xtasy.

Physical operators communicate through method invocation, and exchange a particular kind of data granule called *unboxed tuple*. Even if its name is a true nonsense, the unboxed tuple is designed for combining tuple-based and slot-based data granules, therefore eliminating the need for two different data granule kinds: indeed, path evaluation operators work on document and indexes, hence manipulating EIDs, while other operators work on variable binding tuples.

As shown in Figure 9.1, an unboxed tuple is composed by a vector of pairs ($var_name, EID\ sequence$), and by a slot for a single free EID.

The vector part is used for hosting variable bindings collected during query evaluation, and it is accessed by most physical operators; the free EID slot, instead, is used for carrying EIDs collected during path evaluation. In particular, these EIDs are produced by path evaluation operators, and consumed by path evaluation and

Figure 9.1: An unboxed tuple

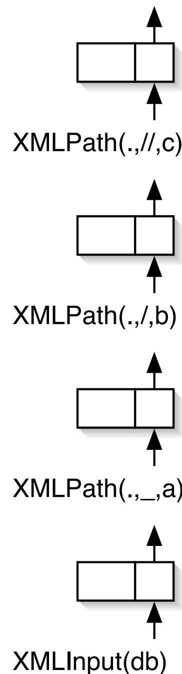


Figure 9.2: Flow of unboxed tuples along operators

binding operators.

Example 9.1.1 Consider the access plan depicted in Figure 9.2, representing an hypothetical translation of `a/b//c`.

With the only exception of the lowest operator, which feeds the query plan with the EIDs of database roots, each operator in the plan consumes the EID contained into the EID slot, and produces a new one replacing the consumed EID. ■

Xtasy operators can be roughly divided into two categories: *unary* operators, which have at most one input operator, and *binary* operators, which have two input operators. These two classes are described in the following Sections.

9.2 Unary Physical Operators

Unary physical operators have at most one input operator; in particular, all operators in the class have exactly one input, with the only exception of the XMLInput

operator, which provides the query plan with the EIDs of the database roots.

Unary physical operators can be organized in a hierarchy, as shown in Figure 9.3.

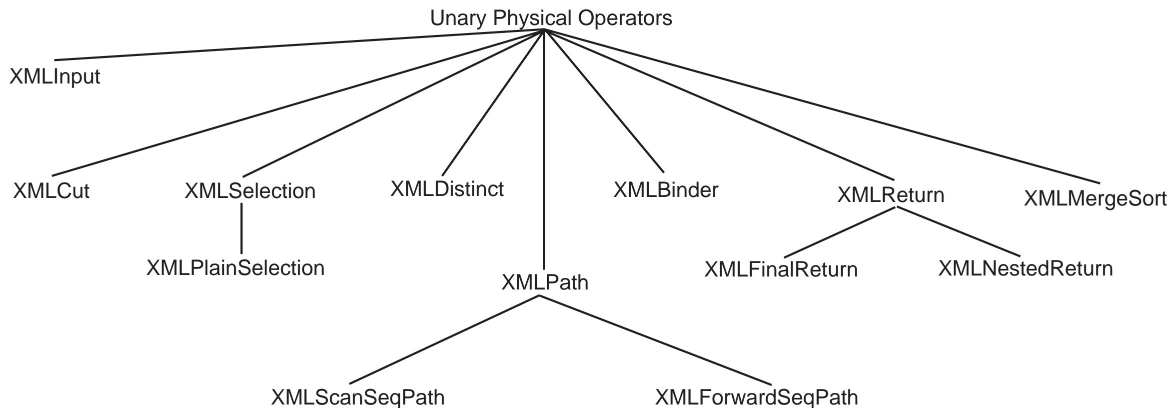


Figure 9.3: Hierarchy of unary physical operators

In the following we will briefly describe the most important ones.[1]

XMLInput

$$XMLInput: \quad Datasource \rightarrow \quad \{UnboxedTuple\}$$

XMLInput has only one task: to provide the query plan with the EIDs of the database roots. Such EIDs are stored in a reserved area in persistent store, to protect them against accidental injuries that would make the database completely unusable.

XMLCut

$$XMLCut: \quad Datasource \times \{String\} \times \{UnboxedTuple\} \rightarrow \quad \{UnboxedTuple\}$$

XMLCut performs a projection of the vector part of unboxed tuples over a set o variable names, just as the usual projection operator of relation databases. During field elimination, no duplicate removal is executed, i.e., $\pi_{\vec{A}}(e)$ cannot be directly translated into XMLCut(ds,A,e). If some fields in A (the string vector) are missing, the operator returns an error tuple, which propagates up to the root of the query plan and generates a run-time error message.

XMLDistinct

$$XMLDistinct: \quad Datasource \times \{UnboxedTuple\} \rightarrow \quad \{UnboxedTuple\}$$

XMLDistinct eliminates duplicates from a list of unboxed tuples.

[1] Physical operator signatures will be described as if physical operators manipulate sequences of tuples; we will abstract from the iterative implementation model.

XMLSelection

$$XMLSelection: \quad Datasource \times \{Conditions\} \times \quad \{UnboxedTuple\} \\ \rightarrow \{UnboxedTuple\}$$

XMLSelection is used for evaluating predicates over unboxed tuples. XMLSelection comes in two flavors. XMLPlainSelection is invoked when the logical condition applies to single tuples, e.g., $data(\$x) > 5$, while XMLQuantifiedSelection comes into play when a quantified condition is being evaluated, e.g., $\forall \$x \in \$p/author : data(\$x) = "McInerney"$.

XMLBinder

$$XMLBinder: \quad Datasource \times String \times String \times \quad \{UnboxedTuple\} \\ \rightarrow \{UnboxedTuple\}$$

XMLBinder creates variable bindings by consuming EIDs contained in the EID slot of unboxed tuples. As a result, XMLBinder increases the size of unboxed tuples.

XMLBinder supports two kinds of binders: *iterative* binder (XQuery's `for...in`), and *grouping* binder (XQuery's `let...:=`). When the iterative binder is applied, XMLBinder just takes the EID contained into the EID slot of the current tuple, puts it into a variable field, and passes the resulting tuple to its father operator. As it can be noted, this kind of binder creates nearly no memory management problems or scalability limitations.

The grouping binder, used for translating XQuery let clauses, requires XMLBinder to entirely evaluate its sub-plan, i.e., to accumulate all results produced by its sub-plan. Thus, XMLBinder calls its input operator as long as possible, and stores the EIDs contained in EID slots into an internal buffer; when the input operator has finished, XMLBinder copies the buffer content into the variable field, and passes the new (single) tuple to its father operator. Buffer overflow issues are treated by using traditional set manipulation techniques, i.e., by creating run files on persistent store and by inserting physical pointers into the EID sequence.

XMLReturn

$$XMLFinalReturn: \quad Datasource \times OutputFilter \times \quad \{UnboxedTuple\} \rightarrow XML$$

$$XMLNestedReturn: \quad Datasource \times OutputFilter \times \quad \{UnboxedTuple\} \\ \rightarrow TempXML \times \\ TempStats$$

XMLReturn is used for creating new XML fragments by filling a XML skeleton with variable values. XMLReturn comes in two versions. XMLFinalReturn produces the final result of a query, which can be serialized on screen or on disk. Therefore,

XMLFinalReturn fills skeleton gaps with the XML trees pointed by the EIDs extracted from tuples; these trees are reconstructed on the fly, which makes this operator quite expensive.

XMLNestedReturn, instead, produces the result of a nested query, and it is described in detail in Section 9.4.

XMLSeqPath

$$XMLSeqPath: \quad Datasource \times String \times String \times Var \times \quad \{UnboxedTuple\} \\ \rightarrow \{UnboxedTuple\}$$

XMLSeqPath is used for evaluating single-step path expressions, while multiple-step path expressions or twigs are translated by combining XMLSeqPath operator with join operators.

XMLSeqPath makes no use of path indexes or other XML specific access structures ("Seq"), hence exploiting only the ability to scan heap-files and to look up the Structural Index.

XMLSeqPath comes in two flavors: XMLForwardSeqPath and XMLScanSeqPath. XMLForwardSeqPath looks up the Structural Index for traversing the data tree, and for finding children and descendants of a given node. XMLScanSeqPath, instead, exploits the clustering of attributes and elements according to their tags, therefore performing a simple scan of the heapfile associated to a given label. The evaluation of // operations by XMLScanSeqPath involves the check for the ancestor/descendant relationship, which is performed by relying on the structural labeling scheme described in the previous Chapter. Since XMLScanSeqPath just scans heapfiles, it is not suitable for evaluating *wild-card* steps such as $\mid *$, $/*$, and $//*$, while XMLForwardSeqPath can be used in any context ($\mid l$ just denotes a set filtering operation).

The following example shows a sample query plan involving unary operators only.

Example 9.2.1 Consider the following query on a bibliographic database.

```
for $b in input()/book
where $b/year < 2001
return $b
```

This query just returns all books published before 2001. The system translates the query in the following algebraic expression.

$$return_{\nu\$b}(\sigma_{\$y<2001}(\\ path_{(/,\$b,in)book[(/,\$y,in)year[\emptyset]]}(db_1)))$$

which is in turn translated in the query plan of Figure 9.4.

■

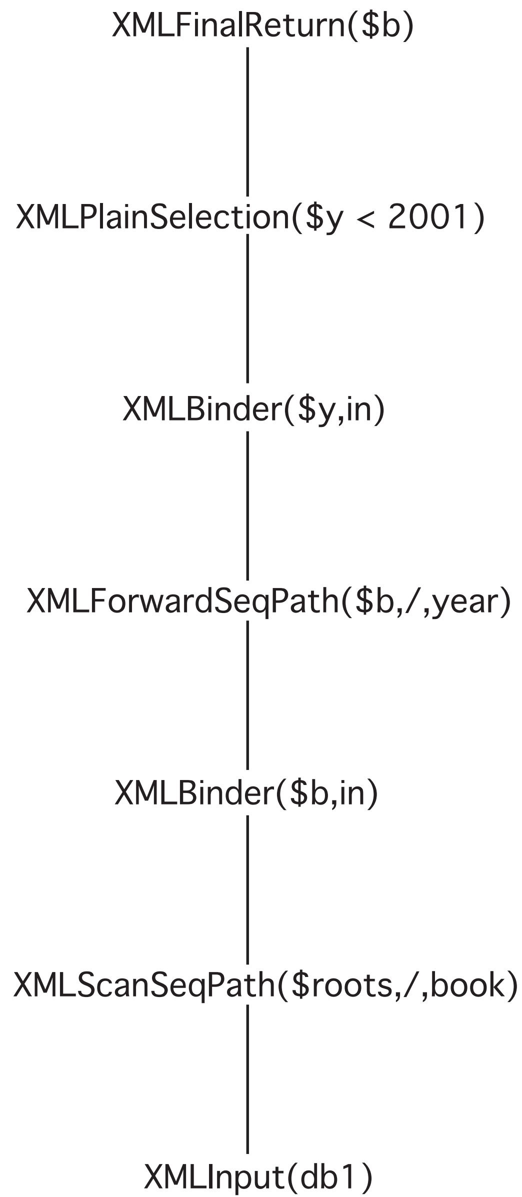


Figure 9.4: Sample query plan

9.3 Binary Physical Operators

Binary physical operators have exactly two input operators, and they are used for connecting query sub-plans as well as for performing some set-based operations.

Binary physical operations can be organized in a hierarchy as shown in Figure 9.5.

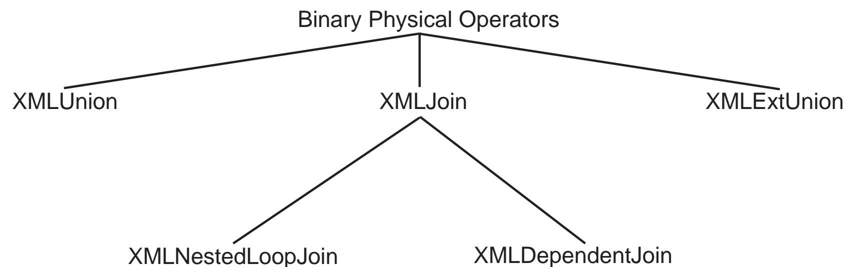


Figure 9.5: Hierarchy of binary physical operators

XMLNestedLoopJoin

$$XMLNestedLoopJoin: \quad Datasource\times \quad Predicate \times \\ \{UnboxedTuple\}\times \quad \{UnboxedTuple\} \rightarrow \{UnboxedTuple\}$$

This operator performs a nested loop between the results of its left input and those of its right input. Combined tuples are then filtered by evaluating an appropriate join predicate. This operator assumes that the left input and the right input are independent, it cannot be used for evaluating djoins induced by nested queries..

XMLDependentJoin

$$XMLDependentJoin: \quad Datasource\times \quad \{UnboxedTuple\} \times \\ \{UnboxedTuple\} \rightarrow \{UnboxedTuple\}$$

This operator behaves as the previous operator, with the only exception that tuples from the left input are passed to the right operand, and used for instantiating and evaluating the right input.

XMLUnion

$$XMLUnion: \quad Datasource \times \{UnboxedTuple\}\times \quad \{UnboxedTuple\} \\ \rightarrow \{UnboxedTuple\}$$

XMLUnion is used to combine two streams of tuples. As usual, no duplicate elimination is performed, the streams being just concatenated. If the streams have different structures, an error tuple is returned.

XMLExtUnion

$$XMLExtUnion: \quad Datasource \times \{UnboxedTuple\} \times \quad \{UnboxedTuple\} \rightarrow \{UnboxedTuple\}$$

XMLExtUnion is used to combine and make homogeneous two streams of tuples. Unlike XMLUnion, XMLExtUnion extends tuples to have a common structure, supplementary fields being filled by using empty EIDs.

The following example shows a sample query plan involving both unary and binary query plans.

Example 9.3.1 Consider the following query on a bibliographic database:

```
for $b in input()/book,
    $a in $b/author,
    $y in $b/year
where $a = "Francis Scott Fitzgerald" AND ($y > 1935)
return $b/title
```

This query returns the titles of the books written by Francis Scott Fitzgerald after 1935. The system translates this query in the following logical algebraic expression.

$$return_{\nu\$t}(\sigma_{\$a=\text{"}FrancisScottFitzgerald\text{"}\wedge\$y>1935}(path_{(/,\$b,in)book[(/,\$a,in)author[\emptyset],(/,\$y,in)year[\emptyset],(/,\$t,in)title[\emptyset]]}(db_1)))$$

This algebraic expression, then, can be translated in the query plan shown in Figure 9.6.

■

9.4 Evaluation of Nested Queries

As previously noted, the system uses XMLNestedReturn for producing results of nested queries. Given the relevance of nested queries in free nesting languages, and, in particular, in XQuery, we chose to pay particular attention to this issue.

Before describing our approach, a brief analysis of the kind of results nested queries produce. The following example shows a typical use of nested queries.

Example 9.4.1 Consider a bibliographic database and the following query, which returns authors and the titles of their works.

```
for $a in input()/*/author
let $title_list := for $b in input()/*
                       $t in $b/title
                   where $a = $b/author
                   return <pubtitle> data($t) </pubtitle>
return <ref> { $a, $title_list} </ref>
```

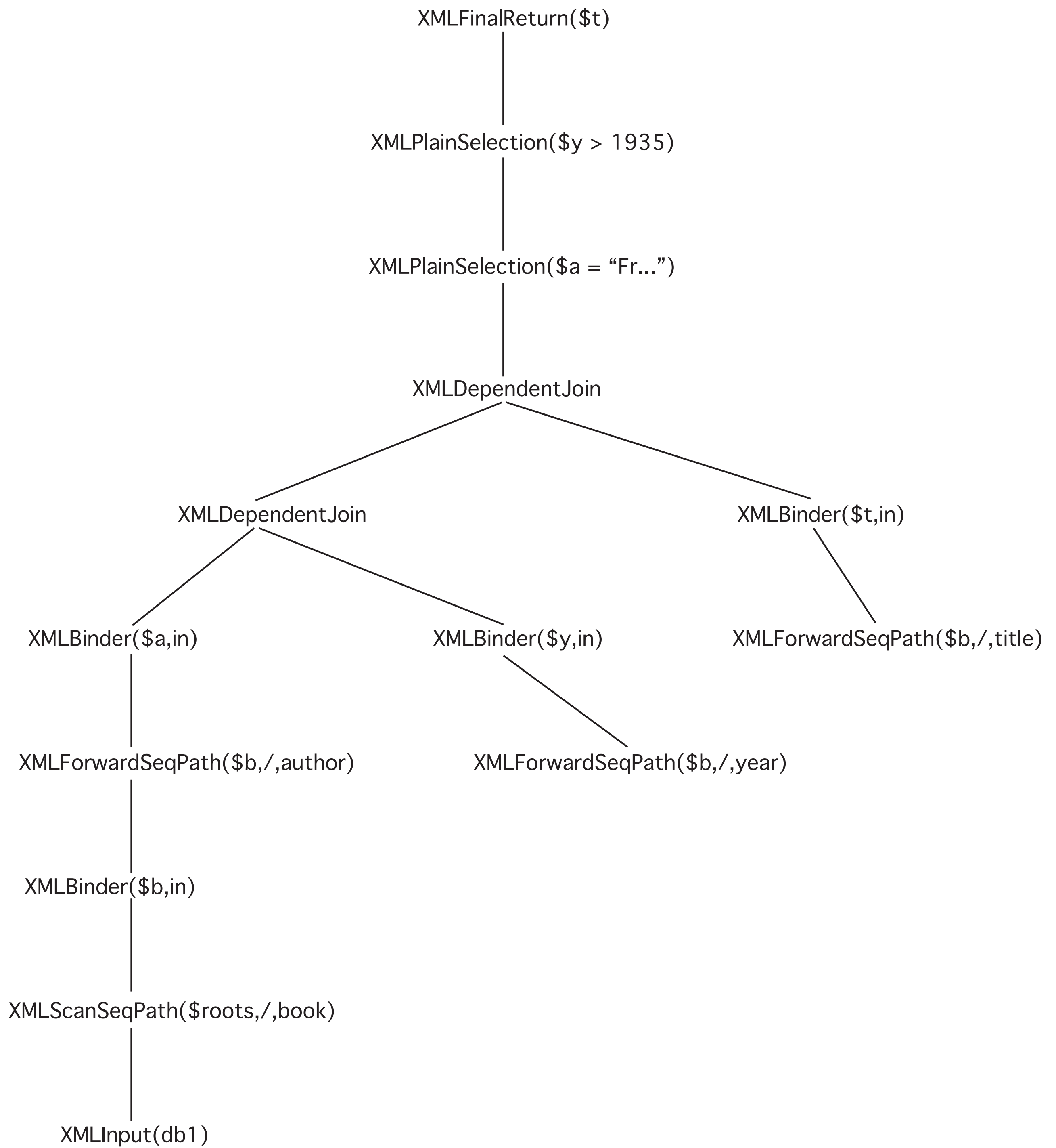


Figure 9.6: Another sample query plan

The inner query selects the titles of works published by any given author, and returns their values enclosed in the new tag pubtitle. ■

This sample query shows that nested queries may return newly created XML nodes (e.g., pubtitle elements), as well as nodes directly copied from the existing database (e.g., `data($t)`). Hence it is worth to classify XML nodes according to the way the are built; this classification will greatly simplify the discussion.

Proposition 9.4.2 *XML nodes (regardless of their type) can be classified as follows:*

- natural *nodes: nodes existing in the persistent database;*
- artificial *nodes: nodes existing in the persistent database and copied into query results;*
- synthetic *nodes: newly created nodes by nested queries.*

For the sake of simplicity, we will consider only element and attribute nodes, and we will not take into account reference semantics RETURN clauses.

For supporting synthetic nodes, the system must set them apart from natural nodes, since they do not survive the execution of the outer query. To this end, the use of simple EIDs, as shown in Chaper 8, is not sufficient, because they discriminate nodes by their name only, and not by their provenance. As a consequence, we associate each query execution with a unique ID, which is then used for extending EIDs. QueryIDs uniquely identify query executions (the same nested query can be instantiated and invoked many times during outer query evaluation), and they are assigned partly at compile time and partly at run time: during query parsing, the system creates a unique ID for each query block; at run time, then, each execution of the query generates a new dynamic ID (*invocation ID*). Hence, each query has associated a set of QueryIDs of the form $\{queryBlock.queryExec\}$.

QueryIDs are volatile, since they perish after outer query execution. The following example shows the assignment of QueryIDs.

Example 9.4.3 Consider the following query, returning book authors grouped with the titles of their books and with their publisher.

```
for $a in input()/book/author
let $title_list := for $b in input()/book,
                       $t in $b/title
                   where $a = $b/author
                   return <pubtitle> data($t) </pubtitle>,

let $p_list := for $b in input()/book,
                   $p in $b/publisher
               where $a = $b/author
               return <pubpublisher> data($p) </pubpublisher>

return <ref> {$a, $title_list, $p_list} </ref>
```

As the system parses the query, it assigns ID qid1 to the first query, and then assigns IDs qid2 and qid3 to the inner queries. During query execution, the system generates invocation IDs for the set of queries; as a result, assuming 10 execution for query block qid2 and 15 executions for query block qid3, the system generates the following list of QueryIDs:

$$
\begin{array}{lll}
qid1.1 & qid2.1 & qid3.1 \\
 & qid2.2 & qid3.2 \\
 & \ldots & \ldots \\
 & qid2.10 & qid3.10 \\
 & & \ldots \\
 & & qid3.15
\end{array}
$$

■

QueryIDs are used for extending EIDs. An extended EID (ExtEID in the following) is a triple $< QueryID, fileID, Rid >$, where $QueryID$ is the ID of the originating query (0 for persistent elements), $fileID$ is the ID of the file containing the element representation, and Rid is the record id of the element representation. Extended attribute IDs can be defined in the same way.

Synthetic nodes must be stored in a way that allows the operators of the outer query plan to access and manipulate them. The simplest way to do so is to build for synthetic nodes exactly the same tabular representation as for natural nodes. Synthetic nodes, hence, are represented as records grouped by tag or attribute name, the hierarchical structure being captured by a new Structural Index fragment. Both the node records and the Structural Index fragment are stored in a reserved storage area, distinct from that of the natural nodes or other query nodes (they do not survive the execution of the outer query, so dirtying the representation of persistent nodes has no justification). As a further distinctive feature, synthetic node records are assigned negative fileIDs, obtained by complementing the fileID assigned to their natural counterparts: if book elements are stored in file $fileID1$, then synthetic book elements are stored in file $-fileID1$. For newly created tags or attribute names, new (negative) fileIDs are generated.

For improving the performance of the system during nested query execution, synthetic nodes (as well as their Structural Index fragment) are stored into *virtual* files: these files are allocated in main memory, and flushed to secondary storage only when their size exceeds the capacity of the allocated memory: this policy, similar to the storage policy of hybrid hash join, allows the system to reduce the use of secondary storage during query execution as much as possible.

While synthetic nodes are explicitly stored by the system, artificial nodes (i.e., persistent nodes copied in nested query results) are *shared*, i.e., no new records are created for them, and their EIDs are refreshed by using a special-purpose hash

table; this hash table contains one entry for each artificial subtree root, and maps the original EID to the new full QueryID.

Artificial nodes are shared primarily for performance and scalability reasons; without node sharing, artificial nodes should be "duplicated", hence leading to potential unpleasant consequences (e.g., copying the whole database).

Example 9.4.4 Consider the following simple query:

```
for $b in input()/book
let $ops := for $c in input()/book
            where $b != $c
            return $c
return <oddPair> {$b,$ops} </oddPair>
```

This query returns each book in the database, as well as the list of all non-matching book. Without node sharing, executing the inner query would bring to the materialization of the whole database. ■

9.4.1 Assigning structural labels to nested query results

In the presence of copy-semantics RETURN clauses, the result of nested queries is a new document, distinct from any other documents previously created, even if can contain nodes coming from the persistent store. As a consequence, nodes in the nested query result should carry a proper structural label, conforming to the structural labeling scheme described in Chapter 8. While synthetic nodes can be labeled by exploiting an algorithm much close to the labeling algorithm used during database loading, the same does not apply to artificial nodes; as a matter of fact, artificial nodes are kept unmaterialized, so we need a way to convert the structural labels in persistent store into structural labels referring to the new document.

The following example briefly illustrates this point.

Example 9.4.5 Consider the following query:

```
for $b in input()/book,
    $a in $b/author
let $t_list := for $bb in input()/book,
               where $a isin $bb/author
               return <titlelist> $bb/title </titlelist>
return <authortitle> {$a, $t_list} </authortitle>
```

This query returns authors paired with the titles of their books. The inner query, in particular, returns, for any given author, the list of titles of their books. title elements are extracted from the database, together with their structural labels, and inserted in a new context, from which they should inherit new structural labels.

■

The structural labeling scheme adopted in Xtasy derived from those described in [CKM02], which are designed for supporting leaf insertions, and are not robust to movements of subtrees and to node sharing, hence existing structural labels cannot be *reused*. A simple way to solve this issue is to materialize the root of each shared tree with updated structural labels, and, if needed, to use this labels for dynamically computing new labels for the root descendant.

The following example illustrates this technique.

Example 9.4.6 Consider again the query of Example 9.4.5, and suppose that the structural labels of the title element $title_i$ are $[257, 258]$ (we assume that the content of a title element is just a string). Assume w.l.o.g. that the inner query encloses $title_i$ into the seventeenth titlelist element $titlelist_{17}$, as shown below:

```
<titlelist> ... </titlelist>
...
<titlelist> title_i </titlelist>
...
```

Assuming that the system assigns labels $[49, 51]$ to $titlelist_{17}$, $title_i$ should receive labels $[49, 50]$ and its beloved child string labels $[49, 49]$.

As a consequence, during query execution the system materializes a copy of element $title_i$ with labels $[49, 50]$, which are then used for computing, if needed, the labels of the child string. ■

Part III

The Result Size Estimator

Chapter 10

Estimation Framework

This Chapter starts the description of the result size prediction model of Xtasy. Unlike other models for estimating the cardinality of XML queries, the prediction model of Xtasy was designed from the beginning to support the estimation for arbitrarily complex FLWR expressions.

The model was developed by first defining a general framework for XML query estimation models, and then by specializing this framework with specific statistics and algorithms: the framework acts as a *metamodel*, on top of which specific models, as the Xtasy model, can be built.

After a brief recall of the main issues in result size estimation, this Chapter will show in detail the general framework as well as the algorithms associated with this framework. The following Chapters will describe the statistics being used in Xtasy, and, finally, the prediction algorithms.

10.1 Issues in Result Size Estimation

Referring to the fragment of XQuery supported by Xtasy (FLWR expressions), the most problematic aspects concern the estimation of path and twig cardinality, the estimation of predicate selectivity, the estimation of group cardinality (let binder of XQuery), as well as the estimation of the cardinality of nested queries. While path and twig estimation is a peculiar issue of XML and semistructured query languages, predicate, group, and nested queries cardinality estimation are well-known problems in database theory and practice. Nevertheless, these problems receive new strength from the irregular nature of XML, as briefly discussed above.

Irregular tree or forest structure XML data can be seen as node-labeled trees or forests; these trees, being commonly used for representing semistructured data, usually have a deeply nested structure, and are far from being well-balanced. Moreover, the same tag may occur in different parts of the same document with a different semantics, e.g., the tag name under person and the tag name under city.

```
<root>
  <persons>
     <person>
        <name>
           <fullname> Caius Julius Caesar
           </fullname>
           <gensname> Julia </gensname>
        </name>
     </person>
  </persons>
  <cities>
    <city>
       <name>
          <ancientName> Roma </ancientName>
          <modernName> Roma </modernName>
       </name>
       <nick> Caput Mundi </nick>
       <nick> Eternal City </nick>
    </city>
    <city>
       <name>
          <ancientName> Pisae </ancientName>
          <modernName> Pisa </modernName>
       </name>
    </city>
    <city>
       <name> New York </name>
       <nick> The Big Apple </nick>
    </city>
  </cities>
</root>
```

Figure 10.1: A sample XML document

The irregular and overloaded structure of XML documents influences cardinality estimation, since the location of a node inside a tree may determine its semantics, and, then, its relevance in operations like path and predicate evaluation. For example, consider the XML document shown in Figure 10.1.

The structure of the name element under person is quite different from the semantics of New York's name, hence the evaluation of any query operation starting from name elements should take this into account.

Non-uniform distribution of tags and values The irregular structure of XML data, together with their hierarchical tree-shaped nature, leads to the *non-uniform* distribution of tags and values in XML trees. XML non-uniformity is further strengthened by the presence of structural dependencies among elements (e.g., name depends on person, etc). As a consequence, a prediction model should track the

provenance of estimated matching elements.

These typical features of XML influence the nature and the "complexity" of the previously cited estimation problems, and give rise to new requirements for prediction models. Hence, a closer look to these problems is necessary.

Path and twig cardinality estimation Path and twig expressions are used in XQuery and in many other XML query languages for retrieving nodes from a XML tree, and for binding them to variables for later use.

The main difficulties in cardinality estimation for path and twig expressions come from the need to reduce the prediction errors induced by joins (paths and twigs are usually translated in sequences of joins), and, for twigs only, from the need to correlate results coming from different branches.

Predicate selectivity estimation The estimation of predicate selectivity is a well-known problem in database theory and practice. The most effective and accurate solutions rely on histograms for capturing the distribution of values in the data, and on the use of the uniform distribution when nothing is known about the data involved in the predicate.

In the context of XML, predicate selectivity estimation poses new challenges. First, XML data are usually distributed in a (very) non-uniform way, hence the use of the uniform distribution can lead to many potential errors. Second, the selectivity of a predicate such as $data(\$n)\ \theta\ value$ depends not only on θ and $value$, but also on a) the nodes bound to $\$n$, which may be heterogeneous, b) the semantics of those nodes (e.g., `name` under `person` is quite different from `name` under `city`), and c) the "region" of the document where those nodes appear.

Many existing prediction models (including [CJK$^+$01], [WPJ02], and [AAN01]), while very sophisticated and accurate, return raw numbers as result of the estimation. Raw numbers, denoting the cardinality of matching nodes in the data tree, do not carry sufficient information for the estimation of subsequent predicates being accurate, hence making the enclosing models not so accurate.

Groups As noted about nested queries, XQuery misses explicit constructs for performing `groupby`-like operations.[1] Nevertheless, the `let` binder can be used for creating heterogeneous sets of nodes, hence for building, together with nested queries, groups and partitions. The `let` binder, unlike the `for` binder, accumulates each node returned by its argument into a set, which is then bound to the binding variable. For example,

[1] We are aware of proposals, both public and private to the W3C XQuery Working Group, for extending XQuery with explicit `group-by` constructs. We delay the extension of the framework until they become more stable.

```
for $c in input()//city,
let $n_list := $c/name,
```

returns, for each city, the list of its names (ancient as well as modern ones).

Estimating the cardinality of the let binder requires the system to a) estimate the number of distinct groups created, b) correlate each group to the variables on which it depends (*$n_list* depends on *$c*), and c) estimate the distribution of nodes and values into each group. This information is necessary since the groups created by the let binder can be used as starting point for further navigational operations, as argument for aggregate functions or for predicates.

The estimation of group cardinality is one of the missing points in current XML prediction model. For what is known to the author, no existing model for XML query languages faces this problem, hence the support of group cardinality estimation at the framework level becomes a *must*.

Nested queries XQuery, as many other XML query languages, is a *free nesting* language, where nested queries are primarily used for reshaping or regrouping elements. Since the result of nested queries may be the input for navigational or filtering operations in the outer query, predicting the size of nested query results requires the system not only to estimate the raw cardinality, but also to build temporary statistics for them. These statistics, as well as the way to generate them, are very model-dependent, and cannot be ascribed to a general framework.

10.2 The Framework

10.2.1 Basics

The main idea behind the framework is to estimate the distribution of data into the result of any query subexpression. Hence, an estimation function based on the framework takes as input a query subexpression (e.g., a node of the query AST, or, as in the model of Xtasy, a node of the physical query plan), as well as a data structure, called ETLS, describing the distribution of the input data, and it returns a new ETLS for the result: this ETLS is obtained by recursively traversing the query subexpression, and by using path and predicate statistics for interpreting for, let, and where clauses. ETLS structures will be described in Section 10.2.3. Inside ETLSs, data distribution is described by means of sequences of *match occurrences*, which are themselves illustrated in Section 10.2.3.

We stress that the framework is in some way independent from specific statistics: it acts as a *metamodel*, on top of which specific models, as the Xtasy model, can be built; models conforming to the framework can benefit from the services offered by the framework.

10.2.2 Tagged Region Graph

Statistic models for database queries usually rely on the partitioning of the database into *regions*, which are used for limiting the scope of aggregated statistics, and, then, for increasing their accuracy. In the proposed framework, regions are defined as follows.

Definition 10.2.1 *Given a document $\mathcal{T}$, a region partitioning scheme for $\mathcal{T}$ is a pair $(\mathcal{R}, \mathcal{F})$, where $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, and $\mathcal{F}$ is a function mapping $\mathcal{T}$ nodes into regions $\mathcal{R}_i$ such that, for any node $x \in \mathcal{T}$ $\mathcal{F}(x) = \mathcal{R}_i \in \mathcal{R}$.*

The notion of region is wide enough to accomplish the needs of different prediction models: it may be the type of the node (*intensional* region), the type of the node together with its location in the originating document (*mixed* region, e.g., the node type and its bucket in the corresponding *structural* histogram in StatiX [FHR+02]), or the location of the node in the originating database (*extensional* region, e.g., the grid cell in the related *position* histogram in TIMBER [WPJ02]). Regions, hence, can express both intensional and extensional concepts, and are the main tool for describing the distribution of data.

Depending on the kind of partitioning used, regions may overlap: for instance, in a purely intensional partitioning based on a type system with subtyping, person regions may contain student regions. When extensional information comes to play, regions are defined as disjoint, in order to ease the construction of proper histograms.

Regions can be further specialized by partitioning them according to the element tags they contain.

Definition 10.2.2 *Given a region $\mathcal{R}_i$ of the document $\mathcal{T}$, $\mathcal{R}_i$ can be split into a set of* tagged regions $\bar{\mathcal{R}} = \{(l_1, \mathcal{R}_i), \ldots, (l_p, \mathcal{R}_i)\}$*, such that $l_1, \ldots, l_p$ are the tags of the elements occurring in $\mathcal{R}_i$, and $\mathcal{R}_i = \cup_{l_j}(l_j, \mathcal{R}_i)$.*

Tagged regions carry more information than regions, since element labels are explicitly indicated. The explicit indication of the label may seem unnecessary, since the region itself may identify the main characteristics of the nodes. This is only partially true. If the partitioning scheme used is intensional and based on DTD-like types, where a 1-1 correspondence between tags and types exists, the label is useless; instead, if there is no 1-1 correspondence between tags and types, the label component becomes necessary.

Tagged regions can be organized to form a graph, the Tagged Region Graph, which is the main statistical structure of the framework. Graph edges are determined by the actual region partitioning scheme as well as by the statistics used in the specific model. In the model of Xtasy, for instance, we use parent/child path statistics, e.g., we store the average number of nodes in the tagged region (l_2, r_2) being sub-elements of nodes in (l_1, r_1) $(N_{afo}((l_1, r_1), (l_2, r_2)))$. Hence, we can define the Tagged Region Graph as follows.

```
type DB = root[persons[Person*],cities[City*]]
type Person = person[PersonName]
type PersonName = name[fullname[String],
                       gensname[String]?]
type City = city[OldCityName,OldCityNick*|
                 NewCityName,NewCityNick*]
type OldCityName = name[ancientName[String]+,
                        modernname[String]]
type OldCityNick = nick[String]
type NewCityNick = nick[String]
type NewCityName = name[String]
```

Figure 10.2: A schema for the sample document

Definition 10.2.3 *Given a document $\mathcal{T}$ and a region partitioning scheme $(\mathcal{R}, \mathcal{F})$, the* tagged region graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *is a directed graph, where nodes are labeled with tagged regions (l, r), and $((l, r), (l', r')) \in \mathcal{E} \iff$ there exists a node $y \in (l', r')$ and a node $x \in (l, r)$ such that y is a child of x in $\mathcal{T}$.*

The following example shows a sample tagged region graph.

Example 10.2.4 Consider the sample document of Figure 10.1, obeying the schema of Figure 10.2.4. Assume that we build an intensional partitioning scheme based on such schema. The corresponding tagged region graph $\mathcal{G}$, then, is shown in Figure 10.3; statistical information related to these graph is shown in Tables 10.1 and 10.2.[2]

(root, DB)	1
(persons, Any)	1
(person, Person)	1
(name, PersonName)	1
(fullName, Any)	1
(gensName, Any)	1
(cities, Any)	1
(city, City)	3
(name, OldCityName)	2
(ancientName, Any)	2
(modernName, Any)	2
(nick, OldCityNick)	2
(name, NewCityName)	1
(nick, NewCityNick)	1

Table 10.1: N_{el} table (Xtasy model) for the tagged region graph of Figure 10.3

[2]Tags without explicit types are mapped into the *Any* type.

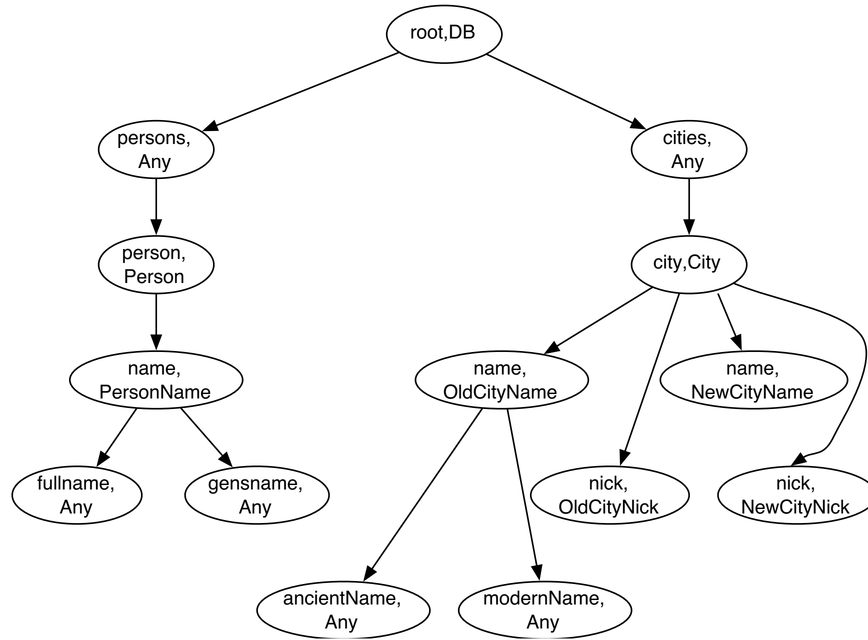


Figure 10.3: Tagged region graph with /-edges

(root, DB)	(persons, Any)	1
(root, DB)	(cities, Any)	1
(persons, Any)	(person, Person)	1
(person, Person)	(name, PersonName)	1
(name, PersonName)	(fullName, Any)	1
(name, PersonName)	(gensName, Any)	1
(cities, Any)	(city, City)	3
(city, City)	(name, OldCityName)	.66
(city, City)	(nick, OldCityNick)	.66
(city, City)	(name, NewCityName)	.33
(city, City)	(nick, NewCityNick)	.33
(name, OldCityName)	(ancientName, Any)	1
(name, OldCityName)	(modernName, Any)	1

Table 10.2: N_{afo} table (Xtasy model) for the tagged region graph of Figure 10.3

The tagged region graph plays a key role in the framework, since many algorithms work on it, and it is the natural repository for *region-related* statistics; for instance, the Xtasy model associates each tagged region with its cardinality ($N_{el}(l, r)$), as well as with parent/child statistics ($N_{afo}((l, r), (l', r'))$).

10.2.3 Match Occurrences, ECLSs, and ETLSs

Estimation functions estimate data distribution in query result by means of sequences of match occurrences, which are formally defined as follows.

Definition 10.2.5 *A match occurrence o is a pair $((l, r), m)$, where (l, r) is a tagged region, and m is the multiplicity of the occurrence; a match occurrence $o = ((l, r), m)$, then, says that m nodes labeled l and belonging to region r are part of the result.*[3]

Estimation functions manipulate sequences of match occurrences, called ECLSs, which can be further organized in more complex structures called ETLSs.[4]

Definition 10.2.6 *An ECLS ecls is a list of match occurrences $ecls = \{o_1, \ldots, o_n\}$, such that $\nexists i, j \in 1, \ldots, n \mid o_i = (l_i, r_i, m_i), o_j = (l_i, r_i, m_j), i \neq j$.*

Definition 10.2.7 *An ETLS (Extended Tuple Label Sequence) is a list of pairs $(\$v, \{e\})$, where $\$v$ is a distinct variable symbol, and $\{e\}$ is a list of ECLSs.*

ETLSs collect estimations about all tuples produced by the system. Two key points must be noted: first, each variable is bound to a sequence (possibly, a singleton) of ECLSs, in order to support the cardinality estimation of groups; second, the ECLS associated with a variable contains all the match occurrences found for the variable, hence only one ETLS is generated during query cardinality estimation.

The following example shows sample ECLSs and ETLSs.

Example 10.2.8 Consider the following query clause:

```
for $c in input()//city,
    $n in $c/name
```

[3]For the sake of simplicity, match occurrences will be denoted in the form (l, r, m) (instead of $((l, r), m)$).

[4]The names of these data structures come from early version of Xtasy, where result size estimation was performed by relying on sequences of labels only.

The estimation function first estimates the result distribution for the path expression `input()//city`, hence returning the following ECLS:

$$\{(city, City, 3)\}$$

This ECLS is then bound to the $\$c$ variable symbol, hence generating the following ETLS:

$$\{(\$c : \{\{(city, City, 3)\}\})\}$$

The estimation function, then, scans the match occurrences bound to $\$c$ to find those whose tagged regions in the graph have `name` children. The multiplicity of these new occurrences is computed by using the statistical information taken from the graph (e.g., N_{afo} and N_{el} in the case of the model of Xtasy). The resulting ETLS is the following:

$$\begin{aligned}\{(\$c : \quad & \{\{(city, City, 3)\}\}),\\ (\$n : \quad & \{\{(name, OldCityName, 2),\\ & (name, NewCityName, 1)\}\}\}\end{aligned}$$

■

10.2.4 Cardinality Notions

One key point in any estimation model is what cardinality notion is used, i.e., how the size measures returned by the model should be interpreted. The most common operational model for XML queries (at least, for queries involving variables) is based on the construction of tuples carrying the values bound to variables [ABS99]; since usual optimization heuristics are based on the minimization of the number of granules generated during query evaluation, the *natural* cardinality notion is the number of such granules (e.g., [FHR+02]).

The proposed framework embodies the vision of intermediate tuple generation, hence it supports the number of generated tuples as cardinality notion. The way this number is computed from ETLSs depends on the specific model being considered. However, the framework contains a general purpose cardinality function $\| \cdot \|$.

The following example shows how tuple cardinality can be computed from ETLSs.

Example 10.2.9 Consider the query of the previous example, and the resulting ETLS, which is reported below.

$$\begin{aligned}\{(\$c : \quad & \{\{(city, City, 3)\}\}),\\ (\$n : \quad & \{\{(name, OldCityName, 2),\end{aligned}$$

$$(name, NewCityName, 1)\}\}\}$$

This ETLS estimates the distribution of data into bound variables; variables are organized in twigs, which may eventually be simple paths (as in this case). The number of generated tuples can be estimated as the number of distinct twig instances, which is computed by multiplying the multiplicity of match occurrences bound to leaf variables of the same twig. In this case, the tuple cardinality is the cardinality of the variable $\$n$, hence the framework correctly predicts that three tuples will be generated. ∎

10.2.5 Correlation

The correlation problem refers to the need of correlating estimations coming from distinct branches of the same twig. Consider, for example, the following query clause:

```
for $x in input()/a,
    $y in $x/b,
    $z in $y/c,
    $w in $y/d
```

This clause matches a two-branch twig against an hypothetical document; in order to correctly predict the number of tuples in the result it is necessary to correlate the estimation for the branch `b/c` with the estimation for the branch `b/d`. Without such a correlation, computing the number of tuples represented by a given ETLS would require the model to multiply the multiplicity of $\$z$ with that of $\$w$ (cross product hypothesis), hence introducing many potential errors.

Twig branch correlation can be performed by using regions. The idea is the following. Once estimated the cardinality of the twig branches, the number of generated tuples can be obtained from the resulting ETLS by identifying the variables having a common root variable, and multiplying the multiplicity of those match occurrences sharing the same parent region.

Example 10.2.10 Consider the following query fragment:

```
for $c in input()//city,
    $n in $c/name,
    $nick in $c/nick,
```

This query fragment retrieves, for each `city` element in the database, its name and the list of its nicknames. By evaluating this clause on the sample document of

Figure 10.1, the framework produces the following ETLS:

$$\begin{array}{ll} \{\$c : \{\{ & (city, City, 3)\}\}, \\ \$n : \{\{ & (name, OldCityName, 2)\}, \\ & (name, NewCityName, 1)\}\}\} \\ \$nick : \{\{ & (nick, OldCityNick, 2), \\ & (nick, NewCityNick, 1)\}\} \end{array}$$

Without any correlation, the predicted number of tuple would be 6, which is clearly wrong (the right number is 3). By correlating nick elements and name elements on the basis of their parent region, the framework can correctly estimate the number of tuples as 3. ■

Correlation affects the tuple cardinality computing function, as well as other facilities of the framework: the cardinality of an ETLS is computed by multiplying the multiplicity of correlated match occurrences only, independent variables (e.g., variables bound to different documents) being considered as fully correlated (each match occurrence is correlated to any other).

10.2.6 Group cardinality estimation

Group cardinality estimation refers to the problem of estimating the dimension of sets created by the let binder, and, more generally, by the use of free path expressions outside the binding clauses for and let. In Section 10.1 three main issues were identified about groups: the estimation of the number of distinct groups; the correlation between each group and the variable instance, which it depends on; and the estimation of the distribution of data into each group.

The number of groups created by the let binder is equal to the multiplicity of the variable on which the groups depend. Consider, for example, the query fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
```

For each city node bound to $\$c$, a distinct $\$n_list$ group is created.

Thus, the framework computes the number of groups by using the following function, which sums multiplicity of match occurrences for a single variable:[5]

$$\|etls(\$c)\| = \begin{cases} \Sigma_{(l,r_l,m_l)\in e} m_l & \text{if } etls(\$c) = \{e\} \text{ and} \\ & \$c \in ungrouped(etls) \\ n & \text{if } etls(\$c) = \{e_1, \ldots, e_n\} \\ & \text{and } \$c \in grouped(etls) \end{cases}$$

[5] $grouped(etls)$ is the set of variables in $etls$ bound by a let clause, while $ungrouped(etls)$ is the set of variables in $etls$ bound by a for clause.

The second case in the definition is necessary when the root variable of the let binder is itself the result of the application of another let binder, as in the fragment shown below:

```
for $c in input()//city,
let $n_list := $c/name,
let $notes := $n_list/notes,
```

In this particular case, the number of groups is equal to the number of groups of the root variable ($\$n_list$).

Once computed the number of groups, the framework must create each group and estimate the data distribution inside it. The group creation algorithm is based on the correlation function, and, performs the following steps: the algorithm, first, collects all match occurrences of the let right member into a set o in $\mathcal{S}$, and creates m empty groups, where m is the estimated number of groups; then, it correlates each occurrence o in $\mathcal{S}$ with the root match occurrences, and distributes them accordingly. The following example illustrates the group creation process.

Example 10.2.11 Consider our well-known query fragment:

```
for $c in input()//city,
let $n_list := $c/name,
```

By estimating for clause cardinality on the sample document of Figure 10.1, the framework generates the following ETLS:

$$\{\$c : \{\{(city, City, 3)\}\}\}$$

The list of match occurrences collected for the let path expression is the following:

$$\{(name, OldCityName, 2),\\ (name, NewCityName, 1)\}$$

The framework creates three groups, and distributes match occurrences as shown below.

$$\{\$n_list : \{\{ \quad (name, OldCityName, 1)\},\\ \{(name, OldCityName, 1)\}\}\\ \{(name, NewCityName, 1)\}\}$$

■

10.2.7 Predicate selectivity estimation

The estimation of predicate selectivity for XML queries shows two main issues. The first issue concerns the estimation process itself as well as the nature of selectivity factors. As already discussed, XML documents have an irregular structure, where tags and values are distributed in a way far from being uniform. As a consequence, the uniform distribution hypothesis is not suitable for predicates on XML data. Moreover, queries can contain *value* predicates (e.g., `data($y) > 1982`, where $\$y$ is bound to year elements) and *structural* predicates about the existence of children nodes with a given label (e.g., `book[publisher]`); finally, even if we consider only value predicates, variables can be bound to heterogeneous nodes, for instance $\$a$ bound to author and publisher nodes. Hence, the selectivity factor for a given predicate P should be a function of structural information.

Given that, the framework supports selectivity factors as functions of the label and the region of a given match occurrence. This choice allows the framework to take structural information (even type information if an intensional partitioning is used) into account, hence increasing the accuracy of the cardinality estimation. The following example clarifies this point.

Example 10.2.12 Consider the following query fragment:

```
for $c in input()//city,
    $n in op:union($c//ancientName,
                   $c//modernName)
where data($n) = "Monticello"
```

The predicate `data($n) = "Monticello"` applies to ancientName and modernName elements, and its selectivity factor depends on the tag of the subject node, since values can have different distributions in ancientName and modernName elements (e.g., "Monticello" is a quite common name for small Italian villages, even though their Latin or medieval names were slightly different). ■

Selectivity factors for unary predicates can be defined as follows.

Definition 10.2.13 *Given a unary predicate P, the selectivity factor of P $psf[P]$ is a function*

$$psf[P] : label \times region \to [0,1]$$

that, given a label l and a region r, returns a real number belonging to [0,1].

Selectivity factors defined according to the previous definition associate each tagged region with a real number comprised between 0 and 1. As a consequence, this definition naturally leads to histogram-based selectivity factors, where buckets correspond to tagged regions, even though the model designer is free to choose the preferred way of collecting and storing statistics. Histograms can be built and managed by using well-known techniques, without the need for particular changes.

The following example illustrates unary predicate selectivity factors.

Example 10.2.14 Consider the query fragment of the previous example, where variable $\$n$ is bound to ancientName as well as modernName elements. Assuming that statistics are gathered from a bigger document than that of Figure 10.1, the selectivity factor for the predicate $P \equiv data(\$n) = "Monticello"$ could be the following:

$$psf[P] = \begin{cases} (ancientName, r_1) \rightarrow 0.2 \\ \ldots \\ (ancientName, r_5) \rightarrow 0.07 \\ (modernName, r_6) \rightarrow 0.75 \\ \ldots \\ (modernName, r_9) \rightarrow 0.67 \end{cases}$$

■

The second main issue about predicate selectivity estimation concerns the way these factors are applied to ETLSs in the framework. For instance, given the predicate `data($n) = "Monticello"`, the values returned by $psf[P]$ are used for decreasing the multiplicity of match occurrences bound to $\$n$. Still, this is not sufficient, as the predicate cuts the number of twig instances collected on data; hence, the multiplicity decrease should be applied also to any directly or indirectly dependent variable ($\$c$ only in the example). For applying this decrease and preserving accuracy, multiplicity decrease propagation should be based on the correlation mechanism previously described.

As a consequence, the framework offers a predicate selectivity factor application function, which, starting from the predicate variable, scans match occurrences, applies the right factor, and reapplies the transformation to directly or indirectly dependent variables. The following example shows how selectivity factors are applied.

Example 10.2.15 Consider again the query fragment of Example 10.2.12, and the selectivity factor of Example 10.2.14; assume that the for clause estimation returns the following ETLS:

$$\begin{aligned} \{\$c : \{\{ \quad & (city, r_1, 2), (city, r_3, 1), (city, r_4, 1)\}\}, \\ \$n : \{\{ \quad & (ancientName, r_5, 2), (modernName, r_6, 1), \\ & (modernName, r_9, 1)\}\}\} \end{aligned}$$

By applying the selectivity factor of Example 10.2.14, the framework generates the following ETLS:

$$\begin{aligned} \{\$c : \{\{ \quad & (city, r_1, .14), (city, r_3, .75), (city, r_4, .67)\}\}, \\ \$n : \{\{ \quad & (ancientName, r_5, .14), (modernName, r_6, .75), \\ & (modernName, r_9, .67)\}\}\} \end{aligned}$$

(we assume that r_5 correlates to r_1, and that r_6 and r_9 correlate to r_3 and r_4 respectively). ■

The following example concludes the description of the estimation approach: the next Section will present the main algorithms of the framework.

Example 10.2.16 Consider the following query.

```
for $c in input()//city,
    $n in op:union($c//ancientName,
                   $c//modernName),
    $nick in $c/nick
where data($n) != "Roma"
return <otherNick> $nick </otherNick>
```

This query returns the nicknames of cities whose ancient or modern name is different from "Roma" (we assume that there exists a proper selectivity factor).

The size estimator first collects matches for the for clause of the query, hence returning the following ETLS:

$$\begin{array}{ll} \{\$c : \{\{ & (city, City, 3)\}\}, \\ \$n : \{\{ & (ancientName, Any, 2)\}, \\ & (modernName, Any, 2)\}\}\} \\ \$nick : \{\{ & (nick, OldCityNick, 2), \\ & (nick, NewCityNick, 1)\}\} \end{array}$$

The size estimator, then, applies the selectivity factor, corresponding to the predicate `data($n) != "Roma"`, to this ETLS: the algorithm cuts $\$n$ occurrences by an half, and then tries to propagates the selectivity factor to directly or indirectly dependent variables; since only occurrences in $(nick, OldCityNick)$ correlate with occurrences in $(ancientName, Any)$ or in $(modernName, Any)$, occurrences in $(nick, NewCityNick)$ are cut off from the resulting ETLS, which is the following:

$$\begin{array}{ll} \{\$c : \{\{ & (city, City, 1.5)\}\}, \\ \$n : \{\{ & (ancientName, Any, 1)\}, \\ & (modernName, Any, 1)\}\}\} \\ \$nick : \{\{ & (nick, OldCityNick, 1), \end{array}$$

By applying the tuple cardinality computing function $\| \cdot \|$, the framework estimates the cardinality of this query as $1 * (1 + 1) = 2$, instead of 1: the overestimation error comes from the inaccuracy generated by intensional regions.

■

```
1  correlated(Match (l, r, m), Match (l', r', m'), Var $root):
3  begin
4    boolean corr = false;
5    for e_i ∈ ecls($root) do
6        if (e_i ⋂ correlationTable((l, r), (l', r'))
7          then corr = true
8                 break
9        fi
10   od
11   return corr
12 end
```

Figure 10.4: Correlation function

10.3 Framework Algorithms

In this section we present the most important algorithms in the framework, namely the correlation function, the selectivity factor propagation algorithm, the group cardinality estimation algorithm, and the cardinality computing function. The description of each algorithm is completed by the discussion about time, and, to a less extent, space complexity.

Correlation function The correlation function, whose algorithm is shown in Figure 10.4, determines whether two match occurrences o_1 and o_2 are correlated by a common ancestor in the ETLS associated to a parent variable $\$root$. The problem is equivalent to the problem of finding a common ancestor in a portion $\mathcal{R}$ of a graph $\mathcal{G}$, where nodes are labeled with pairs $(label, region)$, and edges are determined by the structure of the document and by the region partitioning scheme. Due to the constraint represented by $\mathcal{R}$, NCA (Nearest Common Ancestor) algorithms should be modified to be used in this context. During statistics collection each tagged region (l, r) is endowed with its reachability sets (both ancestors and descendants) in the tagged region graph; moreover, a correlation table is created, associating each pair of tagged regions $((l, r)$,$(l', r'))$ with the list of their common ancestors. The correlation function, then, reduces to the check for the non-emptyness of the intersection between the ECLS of the root variable and the common ancestor list of $((l, r)$,$(l', r'))$; this task can be performed in linear time (in the number of tagged regions of the graph), if tagged regions are replaced by integers, and lists are kept ordered.

The correlation table can be stored by means of bit arrays: each pair $((l, r)$,$(l', r'))$ has associated a n-bit array, where n is the number of nodes in the tagged region graph. As a result, the correlation table space requirement is $O(n^3 logn)$.

Selectivity factor propagation algorithm The selectivity factor propagation algorithm, shown in Figure 10.5, takes as input an ETLS *etls*, a predicate selectivity factor $psf[P]$, and the variable $\$var$ being selected. The algorithm updates the multiplicity of match occurrences associated to $\$var$, as well as the multiplicity of any match occurrence associated to variables occurring in the same twig as $\$var$. This task is performed by relying on a function *twigs*, which, given a query Q, returns the list of twigs contained into Q (it is created during query parsing at no extra cost).

The algorithm first updates the multiplicity of each match occurrence (l, r, m) in the ECLS associated with $\$var$; factors used for updating such multiplicities are taken from the input histogram, e.g., for the match occurrence (l, r, m) is used $psf[P](l, r)$. The algorithm, then, builds a small hash table associating each tagged region with a selectivity factor, which is finally propagated to any match occurrence in the tagged region; these factors come again from the histogram $psf[P]$: a tagged region (l', r') is associated with the factor $psf[P](l, r)$ if (l', r') is in the reachability sets of (l, r) (ancestors or descendants). Reachability conflicts, e.g., (l', r') is in the reachability sets of both (l_1, r_1) and (l_2, r_2) are solved on first come first serve basis.

The cost of the algorithm relies in the construction of this hash table; as a consequence, the algorithm has $O(n^2)$ worst case time complexity, where n is the number of tagged regions in the tagged region graph.

Group cardinality estimation algorithm The group cardinality estimation algorithm (shown in Figure 10.6), given the set of match occurrences returned by the right side of a let expression (*ecls*), creates as many groups as the cardinality of the root variable (lines 4-5), and distributes these occurrences in such groups. If a match occurrence is produced by distinct match occurrences of the root variable, then it is splitted and distributed in the related groups accordingly (lines 11-13)[6]. The algorithm has $O(n^2)$ worst case time complexity, where n is the number of match occurrences in the input ETLS.

Cardinality computing function This function, whose algorithm is shown in Figure 10.7, computes the cardinality of a given ETLS, i.e., the number of tuples represented by this ETLS. The computation is performed under the hypothesis that the statistical model propagates the multiplicity of the collected match occurrences down to the leaves of the twigs, e.g., if the model collects three book elements with, respectively, one, two, and three author elements, then the total number of author occurrences should be 6. If this hypothesis is not satisfied by a given model, or the model supports a different cardinality notion, the model designer is free to define its own cardinality function.

The main idea behind the cardinality function is to isolate the twigs contained in a query Q, to compute the cardinality of each twig query, and then to multiply these

[6] The function *unnest* flattens a sequence of ECLSs into a single ECLS.

```
1  propagation(ETLS etls, PSF psf[P], Var $var):
3  begin
4    Twig twig = Q.twigs($var)
5    Table ancTable = new Table()
6    Table descTable = new Table()
7    Var $root = twig.getRoot()
8    for e_i ∈ etls($var) do
9         for (l, r, m) ∈ e_i do
10             m = m * psf[P](l, r)
11             for (l', r') ∈ ancestor(l, r) do
12                  ancTable.put((l', r'), psf[P](l, r))
13             od
14        od
15   od
16   for e_i ∈ etls($root) do
17        for (l, r, m) ∈ e_i do
18             m = m * ancTable.get(l, r)
19             for (l', r') ∈ descendant(l, r) do
20                  descTable.put((l', r'), ancTable.get(l, r))
21             od
22        od
23   od
24   for $v ∈ twig.varList() \ {$var, $root} do
25        for e_j ∈ etls($v) do
26             for (l, r, m) ∈ e_j do
27                  m = m * descTable(l, r)
28             od
29        od
30   od
31 end
```

Figure 10.5: Selectivity factor propagation algorithm

```
1  groups(ETLS etls, ECLS ecls, Var $root, Var $tgt):
3  // etls is the input ETLS, ecls contains the match occurrences of the result,
5  // $tgt is the variable being bound,
7  // and $root is the variable which $tgt depends on
8  begin
9    int partNum = ||etls($root)||
10   Seq letSeq =< {}, ..., {} >_partNum
11   for o_i = (l, r, m) ∈ ecls do
12       List corrList = ∅
13       for o_j = (l', r', m') ∈ unnest(etls($root)) do
14           if (l, r) ∈ descendants(l', r') then corrList = corrList + {j} fi
15       od
16       for j ∈ corrList do
17           letSeq(j) = letSeq(j) + o_i[m ← m/corrList.size()]
18       od
19   od
20 end
```

Figure 10.6: Group cardinality estimation algorithm

cardinalities. Twig query cardinality is estimated by multiplying the multiplicity of correlated match occurrences: given the above hypothesis, it is sufficient to multiply the multiplicity of occurrences bound to leaf variables, i.e., the variables on the leaves of the twig.

Given the use of the correlation function, the algorithm has $O(n^2m)$ worst case time complexity, where n is the number of match occurrences in the input ETLS, and m is the number of tagged regions in the tagged region graph.

```
1  ‖ · ‖(ETLS etls):
3  begin
4    double rawCard = 1
5    List twigs = Q.Twigs()
6    Array twigCard
7    // twigCard hosts the cardinality of each query twig
8    for t_i ∈ twigs do
9        twigCard(t_i) = twigCardinality(t_i)
10   od
11   rawCard = ∏_{t_i ∈ twigs} treeCard(t_i)
12   return rawCard
13 where
14 funct twigCardinality(Twig t) ≡
15   ⌈
16    List leaves = leafList(t)
17    /* we assume leaves = {v_1, ..., v_h} */
18    Array varCard = {0, ..., 0}_h
19    for i := 1 to h − 1 do
20        ETLS etls_i = etls($v_i)
21        if $v_i grouped
22           then varCard($v_i) = 1
23            else
24                 for j := i + 1 to h do
25                     ETLS etls_j = etls($v_j)
26                     for o_p ∈ unnest(etls_i) do
27                         int localCard = 1
28                         if $v_j ungrouped
29                            then
30                                 for o_q ∈ unnest(etls_j) do
31                                     if correlated(o_p, o_q, commonRoot($v_i, $v_j))
32                                        then localCard = m_p * mq * localCard
33                                     fi
34                                 od
35                         fi
36                         varCard($v_i) = varCard($v_i) + localCard
37                     od
38                 od
39        fi
40        treeCard(t_i) = ∑_{$v_i ∈ leafList(t_i)} localCard($v_i)
41    od
42   ⌋
43 .
45 end
```

Figure 10.7: Cardinality estimation function

Chapter 11

Statistics

The prediction model of Xtasy obeys the general framework described in the previous Chapter, and it can be considered an instance of such framework. The prediction model exploits various kinds of statistics for estimating the cardinality of XML queries; these statistics comprehend the *tagged region graph* (together with the reachability sets), the *correlation table*, the *path* statistics as well as the predicate selectivity factors. The following Sections describe these statistics in detail.

11.1 Tagged Region Graph

As shown in the previous Chapter, the tagged region graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph whose vertexes are labeled with tagged regions (l, r), and whose edges are determined by the characteristics of the document being modeled as well as by the statistical policy. Before seeing how graph edges are built in Xtasy, it is necessary to focus the attention on regions.

The notion of region supported by Xtasy is purely extensional, and regions are defined as follows.

Definition 11.1.1 *A region r is defined as a pair $(h, [p, p + \delta_p])$, where:*

- *h denotes a level in the data tree, i.e., level 1 is associated to the tree root, level 2 is associated to the root children, etc, and*
- *$[p, p + \delta_p]$ is an interval of node positional labels (as shown in Chapter 8, XML nodes are endowed with positional labels).*

By this definition it follows that a data tree $\mathcal{T}$ is covered by regions obtained by horizontally splitting $\mathcal{T}$, and then by further dividing the slices according to position intervals. By varying the δ_p width, the user can adjust the number of regions covering the document (with lower bounds represented by the maximum tree height and by the number of distinct tags in the document), hence she can trade accuracy for statistics size.

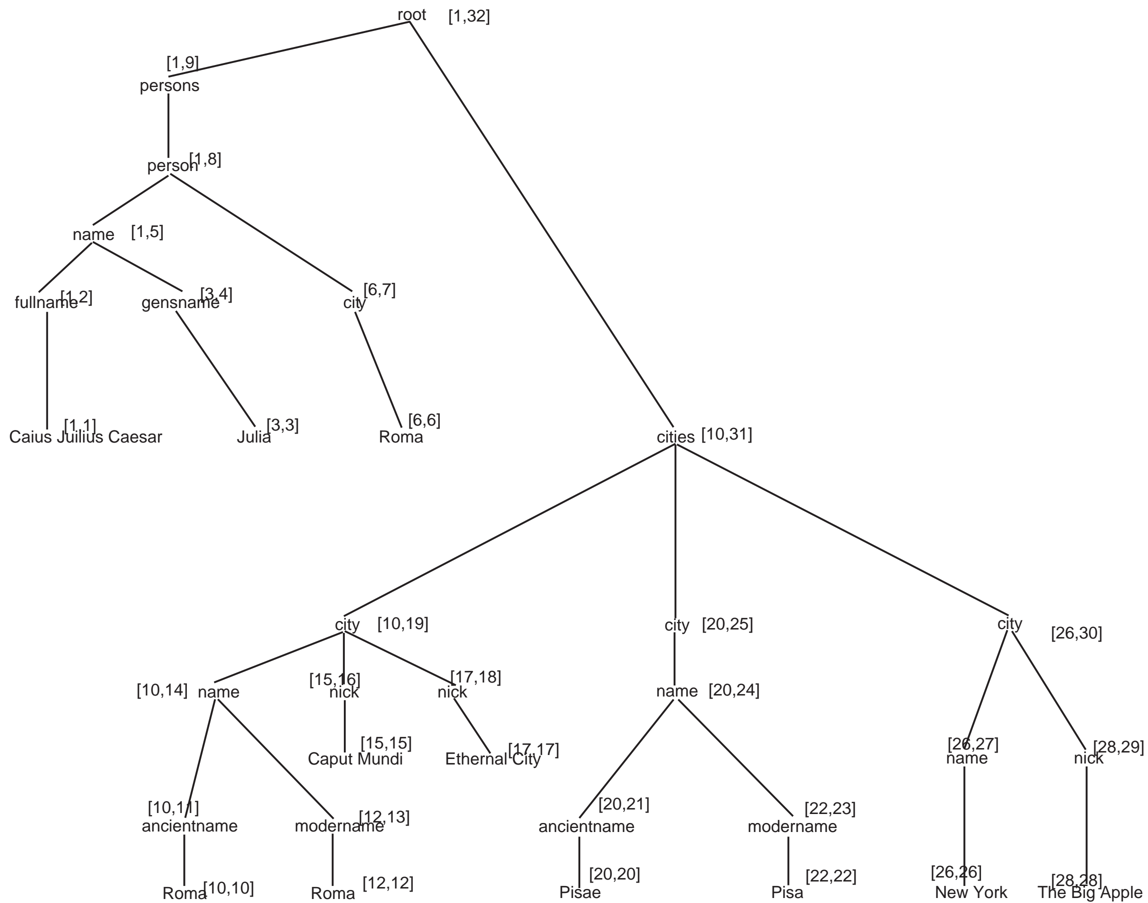


Figure 11.1: A sample tree with positional information

In Xtasy regions are connected by /-edges only, since, as it will be shown in Section 11.3, Xtasy path statistics take into account single-step paths only.

The following example shows how the tagged region graph for the sample document of the previous Chapter appears.

Example 11.1.2 Consider the sample document of Figure 10.1, whose nodes can be labeled with positional information as shown in Figure 11.1.

The document contains 32 nodes, with 9 distinct tags, distributed into 6 levels. By choosing $\delta_p = 10$, the document can be splitted into 16 non-empty regions, enumerated in Figure 11.2: for each region, we indicate the level component (e.g., $h = 2$), as well as the interval of positional labels (e.g., $[31, 40]$).

■

As shown by the previous example, the region partitioning scheme adopted by Xtasy may lead to a high number of regions, with a potential significant space overhead, which may prevent the system from loading the full pool of statistics into main memory. To avoid such nasty situation, Xtasy introduces some minor forms

$(h = 1, [31, 40])$
$(h = 2, [1, 10])$ $(h = 2, [31, 40])$
$(h = 3, [1, 10])$ $(h = 3, [11, 20])$ $(h = 3, [21, 30])$
$(h = 4, [1, 10])$ $(h = 4, [11, 20])$ $(h = 4, [21, 30])$
$(h = 5, [1, 10])$ $(h = 5, [11, 20])$ $(h = 5, [21, 30])$
$(h = 6, [1, 10])$ $(h = 6, [11, 20])$ $(h = 6, [21, 30])$

Figure 11.2: Non-empty tagged regions

of statistic compression: first each tagged region is associated with a unique integer, which is used in place of the pair (l, r), the correspondence between integers and tagged regions being stored into a hash table; second, each region is represented as a bit string $h.p$, where h is the bit representation of a short denoting the tree level of the region, and p is the bit representation of an integer denoting the lower bound of a positional interval.[1]

To further reduce the space requirements for tagged region graph, which is the core statistical structure of the model, the reachability sets are represented by means of bit arrays.

11.2 Correlation Table

As already stated, the correlation table is a hash table associating each pair of tagged regions with the ordered list of common ancestors in the tagged region graph. In the previous Chapter, an implementation of the correlation table based on bit arrays was suggested. Unfortunately, this implementation, as well as any other technique we tried to use, proved to be too expensive in terms of memory allocation: in particular, the size of the correlation table rapidly grows with the growth of region number, hence, making its use nearly impossible.

As a consequence, the Xtasy system does not really generate the correlation table; instead, the correlation function dynamically intersects the ancestor bit sets of its arguments, and checks whether the bit representing the root match occurrences is on or off. This solution leads to an increased time complexity for the correlation function, which is now $O(m + n)$, where n is the number of tagged regions in the graph, and m is the number of match occurrences for the root variable.

11.3 Path Statistics

Path statistics are the core of most XML query cardinality estimation models: they are used for just estimating the cardinality of path expressions, or for predicting the size of twig queries (we found no model directly supporting statistics at the

[1] As Xtasy is implemented in Java, shorts are represented on 16 bits, and integers on 32 bits.

$$
\begin{aligned}
N_{el}((l,r)) &= \text{number of nodes in } (l,r) \\
N_{son}((l_p,r_p),(l,r)) &= \text{number of children nodes in } (l,r) \text{ of nodes in } (l_p,r_p) \\
N_{afo}((l_p,r_p),(l,r)) &= \frac{N_{son}((l_p,r_p),(l,r))}{N_{el}((l_p,r_p))}
\end{aligned}
$$

Figure 11.3: Xtasy path statistics

twig level). Xtasy statistic model is not an exception, path statistics being used for predicting the size of both path and twig queries.

Xtasy path statistics are quite simple, and record, for each tagged region in the graph, the average number of children in any given tagged region; this is equivalent to the use of Markov tables of length $m = 2$, and, as shown in [AAN01], it offers a very high degree of accuracy in path cardinality prediction. More formally, Xtasy path statistics are defined as shown in Figure 11.3. $N_{el}((l,r))$ represents the cardinality of the tagger region (l,r), i.e., the number of nodes in (l,r); $N_{son}((l_p,r_p),(l,r))$, instead, is the number of nodes in (l,r), whose father node is in region (l_p,r_p); $xnafol_pr_plr$, finally, is the average number of children in (l,r) for any node in (l_p,r_p).

Xtasy path statistics are stored together with the tagged region graph, each tagged region being endowed with an array for N_{afo}. This solution eliminates the need for duplicate information, and makes the tagged region graph the core statistical structure of Xtasy.

11.4 Predicate Selectivity Factors

Xtasy predicate selectivity factors strictly obeys the definition given for the estimation framework shown in Chapter 10. Hence, the selectivity factor for a unary predicate P is defined as follows.

Definition 11.4.1 *Given a unary predicate P, the selectivity factor of P $psf[P]$ is a function*

$$psf[P] : TaggedRegion \rightarrow [0,1]$$

that, given a tagged region (l,r), returns a real number belonging to [0,1].

The system does not automatically estimate selectivity factors for predicates: the user must specify what predicates should be estimated: if a query Q contains a predicate for which no selectivity factor was estimated, the system uses for its estimation the usual combination of magic numbers and uniform distribution.

Predicate selectivity factors are stored by means of histograms, each histogram bucket corresponding to a tagged region graph in the tagged region graph. Xtasy histograms, thus, should be considered *equi-width* histograms; while this solution may reduce accuracy w.r.t. *equi-depth* histograms, it allows the system to apply selectivity factors to match occurrences without further manipulations.

11.5 Statistics Collection Algorithms

This Section presents the main algorithms used for statistics collection, namely the tagged region graph creation algorithm, and algorithms for computing reachability sets and path statistics. These algorithms are executed once the XML document has been loaded into the system, hence they are applied to the internal representation of XML data (illustrated in Chapter 8), and not on raw XML documents.

Tagged region graph creation algorithm This algorithm, shown in Figure 11.4, creates the tagged region graph associated to a given document; during graph generation, the algorithm also estimates path statistics, namely the cardinality of each tagged region (N_{el}) as well as the average number of children (N_{afo}).

The algorithm works as follows. First, it gets the EIDs for the datasource roots (line 4), and retrieves the Structural Index entries for the EIDs: these entries are used for obtaining the EIDs of the root children, that are then collapsed to form the list of the EIDs of all elements appearing at the second level in the datasource (lines 8-10); whenever a new tagged region has to be created, the CreateRegion algorithm, shown in Figure 11.5, is invoked. Second, the algorithm estimates the cardinality of the created regions, retrieves Structural Index entries for the EIDs in the current context (i.e., the set EIDs being currently examinated), estimates the average number of children, and then reapplies this procedure until no more EIDs exist in the database (lines 15-25).

The algorithm examines each EID just one time, hence it has $O(n)$ worst case time complexity, where n is the number of database elements.

Reachability set computing algorithms These algorithms are used for computing, for any tagged region in the graph, the sets of descendants and ancestors of the given. The first algorithm, shown in Figure 11.6, computes the set of descendants by means of a recursive exploration of the graph; during the search, each graph edge is visited only one time (lines 12-13), so the algorithm has $O(m+n)$ time complexity, where n is the number of tagged regions in the graph, and m is the number of edges.

The algorithm for computing ancestor sets, shown in Figure 11.7, is based on a different principle. Since graph edges are stored in a forward way, they can be traversed from the root to the target only: this is sufficient for generating descendant sets, while ancestor set generation needs back edges instead. As a consequence,

```
1  TaggedRegionGraph(Datasource ds, int δ_p):
3  begin
4    List root = roots(db)
5    int h = 0
6    List context = new List()
7    // context is a list of pairs ({eid}, r), where
9    // {eid} is the list of EIDs to be processed,
11   // and r is the tagged region containing their father
13   List doubleContext = new List()
14   for eid ∈ root do
15       TaggedRegion r = CreateRegion(eid, h)
16       context = context ∪ (eid.children(), r)
17   od
18   repeat
19           h++
20           for (eid, r) ∈ context do
21               TaggedRegion reg = CreateRegion(eid, h)
22               N_son(r, reg)++
23               N_afo(r, reg) = N_afo(r, reg) + 1/N_el(r)
24               Edge edge = newEdge(r, reg)
25               context = context \ {(eid, r)}
26               doubleContext = doubleContext ∪ (eid.children(), reg)
27           od
28           context = doubleContext
29   until context = ∅
30 end
```

Figure 11.4: Tagged region graph generation algorithm

```
1  CreateRegion(Datasource ds, int δ_p, EID eid, int h):
3  begin
4    int pos = eid.getPos()
5    int regPos = ⌊pos/δ_p⌋ + 1
6    Region reg
7    if ∄ Region(eid.getTag(), h, regPos)
8      then reg = Region(eid.getTag(), h, regPos)
9           G.addRegion(reg)
10     else reg = G.getRegion(eid.getTag(), h, regPos)
11   fi
12   N_el(reg)++
13 end
```

Figure 11.5: Region creation algorithm

```
1  Descendants(TaggedRegionGraph G):
3  begin
4    List rootList = G.rootList()
5    rootList() returns the list of root tagged regions in G
7    for (l, r) ∈ rootList do
8        descendants(l, r) = recDescendants((l, r))
9    od
10 where
11 funct recDescendants(TaggedRegion(l, r)) ≡
12   ⌈
13    for unmarked((l, r), (l′, r′)) ∈ G.edges() do
14        mark((l, r), (l′, r′))
15        descendants(l, r) = descendants(l, r) ∪ {(l′, r′)} ∪ recDescendants((l′, r′))
16    od
17    return descendants(l, r)
18   ⌋
19 .
20 end
```

Figure 11.6: Reachability set computation algorithm: descendants

```
1  Ancestors(TaggedRegionGraph G):
3  begin
4    List regionList = G.regions()
5    for (l, r) ∈ regionList do
6        for (l', r') ∈ descendants(l, r) do
7            ancestors(l', r') = ancestors(l', r') ∪ {(l, r)}
8        od
9    od
10 end
```

Figure 11.7: Reachability set computation algorithm: ancestors

ancestor sets cannot be computed by exploring the tagged region graph, as for descendant sets; they are, instead, generated by scanning descendant sets associated to graph nodes, and by reversing them. Hence, the algorithm has $O(n^2)$ time complexity, where n is the number of tagged regions in the graph.

Chapter 12

Estimation Model

Result size estimation is an important feature of any database system. Predicting the size of a given query allows the system to provide the user with an early feedback about the dimension of the expected results (e.g., if the predicted size of a SQL query is about one billion records, the user can stop monitoring the console, and starts watching her favorite DivX movie). Moreover, the estimation of the size of a query can be very useful when using a database programming language, i.e., it can be used as hint to the compiler for an optimized resource allocation. Finally, result size is an important part of the query cost evaluation process: indeed, most size and cost equations rely on the cardinality or size of previous operations, therefore, errors in the size estimation may propagate to the cost estimation, and significantly affect the choices of the query optimizer.

This Chapter presents the estimation model of Xtasy: the model is based on the framework described in Chapter 10, and exploits the statistics shown in the previous Chapter.

12.1 Basic Concepts

The estimation model of Xtasy was primarily designed as a tool for supporting query cost estimation, hence, the model estimates the size of the expected result of a given physical query plan. This does not represent a limitation in the use of the model: given a query Q, its expected size can be computed by generating a query plan with a random query plan generator, i.e., a plan generator that, at any choice point, performs random choices, and then estimating the size of the result of the query plan.[1]

The following example briefly illustrates this point.

[1] It should be noted that different query plans for the same query, while producing the same result, may lead to different size estimation; this is due to error propagation issues, which, for the sake of simplicity, are ignored in this model.

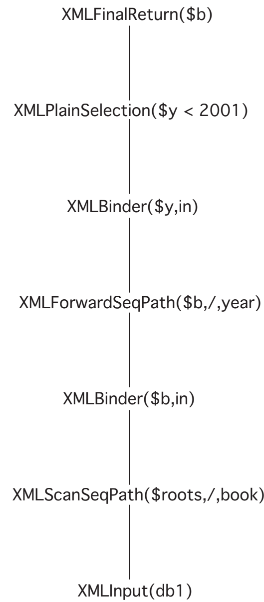


Figure 12.1: Sample query plan

Example 12.1.1 Consider the following query (also shown in Example 9.2.1). This query retrieves `book` elements representing books published before 2001.

```
for $b in input()/book
where $b/year < 2001
return $b
```

To estimate the size of this query, Xtasy generates a random query plan for the query, as shown in Figure 12.1; this query plan represents the input for the result size estimator of Xtasy.

■

The prediction model of Xtasy consists of a polymorphic estimation function ES (Estimated Size). ES takes as input a physical operator op as well as the ETLSs produced by its input operators, and it returns the ETLS for op. Since Xtasy has only unary and binary physical operators, ES has the following signatures:

$$ES : UnaryPhysicalOperator \times ETLS \longrightarrow ETLS$$
$$ES : BinaryPhysicalOperator \times ETLS \times ETLS \longrightarrow ETLS$$

Before seeing how ES is defined for physical operators in the physical query algebra, it is worth to illustrate how ES is used by the result size estimator. The

```
1  SizeEstimator(QueryPlan qp): ETLS
3  begin
4     Node root = qp.Root()
5     HashTable sizeTable = new HashTable()
6     estimation(root)
7     return sizeTable.get(root)
8  where
9  funct estimation(Node node) ≡
10    ⌈
11     NodeList children = node.getChildren()
12     if children = ∅
13       then
14             if node instanceof XMLInput
15               then sizeTable.put(node, ES(node, ∅))
16                else sizeTable.put(node, ES(node, node.getContextETLS()))
17             fi
18        else
19             for node_i ∈ children do
20                  estimation(node_i)
21             od
22             sizeTable.put(node, ES(node, sizeTable.get(children)))
23     fi
24    ⌋
25 .
26 end
```

Figure 12.2: Result size estimation algorithm

result size estimator of Xtasy performs a depth first traversal of the query plan, as shown in Figure 12.2. [2]

The algorithm performs a recursive depth first search of the query plan through the support function *estimation*. Once the ETLS for a given node has been estimated, it is stored into a hash table; while the hash table is not necessary when performing size estimation only, it can be very useful when the size estimation is performed as part of the query optimization process.

[2] Function $getContextETLS()$ extracts the ETLS corresponding to the tuples used by XMLDependentJoin to instantiate its right operand.

$$
\begin{aligned}
|\,\centerdot\,| \quad &: \quad ECLS \rightarrow MatchOccurence \\
|e_i| \quad &= \quad \begin{cases} (l,m,r) & \text{if } e_i = \{(l,m,r)\} \\ (_,0,\emptyset) & \text{otherwise} \end{cases} \\
& \quad\ \text{where } \emptyset \text{ denotes the null region} \\
\uplus \quad &: \quad ECLS \times ECLS \rightarrow ECLS \\
ecls_1 \uplus ecls_2 \quad &= \quad \{(l, m_1+m_2, r) \mid (l,m_1,r) \in ecls_1 \wedge (l,m_2,r) \in ecls_2\} \cup \\
& \quad\ ecls_1 \setminus \{(l,m_1,r) \mid (l,m_1,r) \in ecls_1 \wedge (l,m_2,r) \in ecls_2\} \cup \\
& \quad\ ecls_2 \setminus \{(l,m_2,r) \mid (l,m_1,r) \in ecls_1 \wedge (l,m_2,r) \in ecls_2\} \\
\odot \quad &: \quad ETLS \times ETLS \rightarrow ETLS \\
etls_1 \odot etls_2 \quad &= \quad etls_2 \cup (etls_1 \setminus \{(_, \langle ecls \rangle) \mid (_, \langle ecls \rangle) \in etls_1\})
\end{aligned}
$$

Figure 12.3: ECLS and ETLS manipulation functions

12.2 ECLS and ETLS Manipulation Functions

These functions, shown in Figure 12.2, are used throughout the model to manipulate ECLSs and ETLSs, hence representing the building blocks of the whole model.

$|\,\centerdot\,|$ maps a singleton ECLS into its only element. $\uplus$, instead, takes as input two ECLSs, and returns a new ECLS obtained by merging together the input ECLSs: match occurrences referring to the same tagged region are merged together, while the remaining occurrennces are left untouched. The algorithm used in the model for $\uplus$ has $O(n+m)$ worst case time complexity, where n is the number of match occurrences in $ecls_1$, and m is the number of match occurrences in $ecls_2$; the algorithm exploits the fact that match occurrences are kept ordered in the ECLS, the order relation being defined as the ordered relation among integers representing tagged regions.

12.3 Estimation for Unary Physical Operators

This Section presents the fragment of the estimation model concerning unary physical operators. For most operators, size equations will be provided, together with the time complexity of the corresponding algorithms used in Xtasy; for XMLSeqPath, the estimation algorithm, together with its proof of termination, will be described.

$$PathES(XMLSeqPath(_, *, v), ecls) \equiv ecls$$
$$PathES(XMLSeqPath(_, l, v), ecls) \equiv \{(l, m, r) \mid (l, m, r) \in ecls\}$$

Figure 12.4: Context filtering size equations

XMLInput As stated in Chapter 9, XMLInput provides access to the roots of the datasource, hence its estimation function just returns an ETLS describing such roots.

$$ES(XMLInput(Datasource\ \ ds), \emptyset) \equiv \{(_, \langle\{\ldots\}\rangle)\}$$

XMLSeqPath XMLSeqPath is used, in both its incarnations, to evaluate single-step path expressions; these path expressions may contain wildcards as well as closure operators.

For the purpose of size estimation, three main cases can be identified: context filtering; /; and //. In all these cases, size estimation is performed by using an auxiliary function $PathES$, which directly manipulates ETLSs instead of ETLSs.

$$PathES : PathOp \times ECLS \rightarrow ECLS$$

ES is connected to $PathES$ by the relation shown below.

$$ES(XMLSeqPath(op, l, v), etls) \equiv etls \odot$$
$$\{(_, \langle \bullet_{i=1}^{n} PathES(XMLSeqPath(op, l, v), e_i)\rangle) \mid etls(v) = e_1, \ldots, e_n\}$$

This equivalence says that the result size of a path operator can be evaluated by pushing the evaluation on the ECLSs bound to the root variable.

Size equations for context filtering are shown in Figure 12.3.

As shown by the equations, context filtering size can be evaluated by scanning the sequence of ECLSs bound to the root variable; as a consequence, this estimation can be performed in $O(n)$ worst case time, where n is the number of match occurrences in the input ETLS.

Size estimation for / navigational operations is a bit trickier, since it requires the system to lookup the Tagged Region Graph for finding, for each input match occurrences, its outgoing edges, as shown in Figure 12.3.

$$PathES(XMLSeqPath(/, l, v), ecls) \equiv \uplus_{(l_p,m_p,r_p)\in ecls}\{(l, m * N_{afo}((l_p, r_p), (l, r)), r) \mid ((l_p, r_p), (l, r)) \in G.edges()\}$$
$$PathES(XMLSeqPath(/, *, v), ecls) \equiv \uplus_{(l_p,m_p,r_p)\in ecls}\{(l, m * N_{afo}((l_p, r_p), (l, r)), r) \mid ((l_p, r_p), (l, r)) \in G.edges()\}$$

Figure 12.5: Slash size equations

Both in the case of $/l$ and $/*$, the system scans, for each match occurrence in the input ECLS, the N_{afo} array to find the average fan-out for the corresponding tagged region; the only difference is that, in the case of $/l$, this scan can be limited to the portion of the array containing tagged regions with label l. Despite this, the worst case time complexity for $/l$ is the same as for $/*$, and it is equal to $O(n^2)$, where n is the number of match occurrences bound to the root variable in the input ETLS.

Size estimation for // navigational operations is more difficult than for context filtering and / operations. Indeed, the evaluation of // operations requires the system to explore the data subtrees rooted in the context nodes, in the search for nodes with the proper label. In the same way, since Xtasy path statistics capture the parent/child relationship, the size estimator must traverse the portions of the beloved Tagged Region Graph rooted in the input match occurrences, to find the tagged regions with the proper label; furthermore, as a general Tagged Region Graph may contain cycles, it is necessary to explore the graph in a way that ensures the termination of the algorithm.

The size equations for // navigational operations are shown in Figure 12.3.

The idea behind these size equations is to fully explore the subgraphs rooted in the input match occurrences ($Path(c_i)$ equations), accumulating the found matches occurrences in a new ECLS, which is then filtered to find those with the proper label (they remain unfiltered in the case of $//*$ operations).

Before further commenting these equations, it is worth to see the algorithm used in Xtasy for estimating // cardinality (see Figure 12.7).

The algorithm scans the list of match occurrences bound too the root variable (lines 5-9), and, for each match occurrence, explores the relevant portion of the Tagged Region Graph through the support function $DescendantTree$. This function performs a breadth first search of the subgraph rooted in its input, and returns the match occurrences collected during the exploration.

$$
\begin{aligned}
PathES(XMLSeqPath(//, *, v), ecls) &\equiv \uplus_{(l_p,m_p,r_p)} \uplus_{i \geqslant 0} Path(c_i) \\
PathES(XMLSeqPath(//, l, v), ecls) &\equiv \{(l, m, r) \in \\
&\quad PathES(XMLSeqPath(//, *, v), ecls)\} \\
Path(c_0) \equiv \{(l_x, m_x, r_x) &\mid ((l_p, r_p), (l_x, r_x)) \in G.edges(), \\
&\quad m_x = m_p * N_{afo}((l_p, r_p), (l_x, r_x))\} \equiv c_1 \\
Path(c_1) \equiv \uplus_{(l_y,m_y,r_y) \in c_1}\{(l_x, m_x, r_x) &\mid ((l_y, r_y), (l_x, r_x)) \in G.edges(), \\
&\quad m_x = m_y * N_{afo}((l_y, r_y), (l_x, r_x))\} \equiv c_2 \\
Path(c_i) \equiv \uplus_{(l_y,m_y,r_y) \in c_i}\{(l_x, m_x, r_x) &\mid ((l_y, r_y), (l_x, r_x)) \in G.edges(), \\
&\quad m_x = m_y * N_{afo}((l_y, r_y), (l_x, r_x))\} \equiv c_{i+1}
\end{aligned}
$$

Figure 12.6: Double slash size equations

The termination property of this algorithm relies on the following lemma.

Lemma 12.3.1 *The Tagged Region Graph $\mathcal{G}$ generated by Xtasy is acyclic.*

Proof: Consider a tagged region (l, r) in $\mathcal{G}$, and assumes that (l, r) is part of a cycle C. Without loss of generality, we can think of C as $C = \{(l, r), (l_1, r_1), \ldots, (l_n, r_n)\}$, where $((l, r), (l_1, r_1)) \in \mathcal{G}.edges()$, $((l_i, r_i), (l_{i+1}, r_{i+1})) \in \mathcal{G}.edges()$ $i = 1, \ldots, n-1$, and $((l_n, r_n), (l, r)) \in \mathcal{G}.edges()$.

By the definition of graph edges in Xtasy, it follows that $((l_x, m_x, r_x), (l_y, m_y, r_y)) \in \mathcal{G}.edges() \iff \exists n_x \in (l_x, m_x, r_x)$ and $\exists n_y \in (l_y, m_y, r_y)$ such that n_y is a child of n_x in the data tree. Since Xtasy regions have the form $(h, [p, p + \delta_p])$, where h is a tree level, it follows that $h_y = h_x + 1$. This relations should hold for any edge in the cycle, and, in particular, for $((l_n, m_n, r_n)$,(l,r)): $h = h_n + 1$. Nevertheless, $h_n = h_{n-1} + 1 = \ldots = h_1 + n - 1 = h + n$, hence $h_n > h$. This contradiction proves the thesis.

Theorem 12.3.2 *The algorithm DOSEstimation terminates.*

Proof: The proof is based on the fact that *DescendantTree* explores the subgraph level by level; since the previous lemma states the absence of cycles in the graph, the exploration terminates. As the function *DescendantTree* is called once for each match occurrence bound to the root variable, the algorithm terminates.

The algorithm for estimating the cardinality of // operations has $O(mn^2)$ worst case time complexity, where m is the number of match occurrences bound to the

```
1  DOSEstimation(XMLSeqPth(//,l,v), ETLS etls): ETLS
3  begin
4    List partialResult = new List()
5    for e_i ∈ etls($v) do
6        for (l_p, m_p, r_p) ∈ e_i do
7            partialResult = partialResult ⊎ DescendantTree((l_p, m_p, r_p))
8        od
9    od
10   if l =' *'
11     then return etls ⊙ {(_, ⟨partialResult⟩)}
12     else List result = new List()
13          for (l_x, m_x, r_x) ∈ partialResult do
14              if l_x = l
15                then result = result ⊎ {(l_x, m_x, r_x)}
16              fi
17          od
18          return etls ⊙ {(_, ⟨result⟩)}
19   fi
20 where
21 funct DescendantTree(MatchOccurrence (l_p, m_p, r_p)) → ECLS ≡
22   ⌈
23    List result = new List()
24    List context = ⟨(l_p, r_p)⟩
25    List doubleContext = new List()
26    while (context not empty) do
27          for (l, m, r) ∈ context do
28              for ((l, r), (l_x, r_x)) ∈ G.edges() do
29                  doubleContext.add((l_x, m * N_afo((l, r), (l_x, r_x))))
30              od
31          od
32          result = result ⊎ context
33          context = doubleContext
34    od
35    return result
36   ⌋
37 .
38 end
```

Figure 12.7: // cardinality estimation algorithm

$$ES(XMLBinder(v, in), etls) \equiv etls[v/_]$$
$$ES(XMLBinder(v, =), etls) \equiv groups(etls, etls(_).unnest(), root(v), v)$$

Figure 12.8: XMLBinder size equations

root variables, and n is the number of regions in the tagged region graph. Indeed, *DOSEstimation* loops over match occurrences bound to $\$v$, and, for each match occurrence, it calls *DescendantTree*. *DescendantTree* consists of a threefold loop, used to explore the subgraph rooted in its argument level by level. The worst case happens when the argument of *DescendantTree* is a match occurrence corresponding to a root region in the graph, and it requires the algorithm to fully explore the graph.

XMLCut XMLCut projects the stream of unboxed tuples coming from its only input operator over a list of variables. Hence, the size estimator just removes from the input ETLS the entries for the variables being cut. The size equation for XMLCut is shown below.

$$ES(XMLCut(\$v_1, \ldots, \$v_n), etls) \equiv etls(\$v_1) \bullet etls(\$v_2) \bullet \ldots \bullet etls(\$v_n)$$

XMLBinder XMLBinder takes as input a variable symbol, a binder, and a stream of input tuples. It returns the input stream endowed with a new field for the variable symbol, which is bound to the EID contained in the EID slot.

When the binder is *in*, the system just extracts the EID from the EID slot, and moves it to the variable field; when the binder is =, instead, the system accumulates the EIDs contained in the input tuple EID slots, and moves them to the variable field.

Size equations for XMLBinder are shown in Figure 12.3.

The first equation is straightforward, the system just replacing _ with v; the second equation, instead, exploits the framework group cardinality estimation function, where $root(v)$ is the root variable for v (see Figure 10.6 in Chapter 10 for the details about the group cardinality estimation function).

XMLSelection As for XMLBinder, XMLSelection size estimation is performed by relying on the facilities offered by the almighty framework described in Chapter 10.

```
1  applypsf(Predicate P, ETLS etls): ETLS
3  begin
4    open statistics repository
5    PSF psf = new PSF()
6    Var root = new Var()
7    if selectivity factor for P there exists
8      then psf = SelectivityFactorTable(P).psf()
9            root = SelectivityFactorTable(P).startVar()
10     else psf = new MagicHistogram(P)
11           root = random(P.vars())
12   fi
13   return propagation(etls, psf, root)
14 end
```

Figure 12.9: Predicate selectivity factor application algorithm

The current version of Xtasy does not support universally quantified predicates, i.e., XMLQuantifiedSelection is not supported.

XMLSelection size estimation consists of the application of a selectivity factor to the input ETLS; this application is performed by the *applypsf* algorithm, shown in Figure 12.9, which also has the duty to verify the existence of a selectivity factor for the given predicate.

applypsf first checks whether there exists a selectivity factor for the predicate P (line 7). If a selectivity factor has been defined, then *applypsf* retrieves it from the statistics repository; otherwise, the algorithm assumes that the uniform distribution can be *safely* applied. In any case, *applypsf* calls the propagation algorithm of the framework with the chosen factor.

The complexity of the algorithm is that of the propagation algorithm.

XMLNestedReturn XMLNestedReturn is the physical operator devoted to produce the result of a nested query, and to bind this result to a proper variable. Since the result of nested queries may be the starting point of further navigations and/or computations, a proper result size estimation for nested queries is necessary. Nested queries have the nice property of making database statistics pretty unuseful, hence the system must guess on the fly new statistic information for the result of nested queries; in particular, the system computes the following information:

- a new temporary tagged graph, holding the newly created regions as well as the links with the regions in the original graph;
- the cardinality of each region in the temporary graph, as well as the N_{afo} information;
- the reachability sets for the new regions.

```
1  SyntheticRegions(OutputFilter of, ETLS etls)
3  begin
4    int h = 0;
5    TaggedRegionGraph newGraph = new TaggedRegionGraph()
6    synReg(of, h, etls)
7    tempDescendants(newGraph)
8    tempAncestors(newGraph)
9    computeNafos(newGraph)
10 end
```

Figure 12.10: Algorithm for computing temporary statistics

The resulting size equation for XMLNestedReturn is shown below.

$$ES(XMLNestedReturn(of, v), etls) \equiv etls \odot \{(v, \langle tempGraph.roots() \rangle)\}$$

where *tempGraph* is the temporary tagged region graph.

The algorithm for computing *tempGraph*, and, more generally, the new statistical information is shown in Figure 12.10.

This algorithm first creates the temporary graph, and computes the cardinality of each region; then, it estimates the N_{afo} information for the new regions, and, finally, it computes descendants and ancestors for the new regions. The algorithm exploits the support functions *synreg*, *tempDescendants*, *tempAncestors*, and *computeNafos*, shown, respectively, in Figure 12.11 in Figure 12.12, in Figure 12.13, and in Figure 12.14, as well as the *createTempRegion* function, which creates a new synthetic region (l, h), and, if already created, just increases its cardinality.

Temporary regions have no positional information; this simplification is justified by the small number of newly created regions as well as by the need to lower the time taken to compute the new statistics. Furthermore, the N_{afo} information is computed by just dividing the cardinality of the target region by the cardinality of the root region: this simplification obeys the need to lower the computational requirements of the estimation.

The size estimator does not reevaluate selectivity factors for predicates over data coming from nested queries, hence the magic number approach is used for them.

12.4 Estimation for Binary Physical Operators

XMLNestedLoopJoin XMLNestedLoopJoin takes as input two tuple streams and a predicate P; these tuples are concatenated, and those satisfying the predicate become part of the result. From a size estimation point of view, the result size of a XMLNestedLoopJoin operation can be estimated by concatenating the input ETLSs,

```
1  synReg(OutputFilter of,int h, ETLS etls): List
3  begin
4    List vector = new List()
5    case of
7    1)of = of_1, ..., of_n
8    for i := 1 to n step 1 do
9        List regionVector = synReg(of_i, h, etls)
10       vector.add(regionVector)
11   od
12   2)of = v_B
13   TaggedRegionreg = createTempRegion('value', h)
14   vector.add(reg)
15   newGraph.add(reg)
16   3)of = label[of']
17   TaggedRegion r = createTempRegion(label, h)
18   newGraph.add(r)
19   vector.add(r)
20   List regionVector = synReg(of', h + 1)
21   for reg ∈ regionVector do
22       newGraph.addTempEdge(r, reg)
23       vector.add(reg)
24   od
25   4)of = $var
26   for e_i ∈ etls($var) do
27       for (l, m, r) ∈ e_i do
28           TaggedRegionreg = createShadowReg(l, h)
29           N_el(reg) = m
30           newGraph.add(reg)
31           vector.add(reg)
32       od
33   od
35   return vector
36 end
```

Figure 12.11: Algorithm for computing synthetic regions

```
1  tempDescendants(TaggedRegionGraph G)
3  begin
4    List rootList = G.rootList()
5    for (l, r) ∈ rootList do
6         descendants(l, r) = recDescendants((l, r))
7    od
8  where
9  funct recDescendants(TaggedRegion(l, r)) ≡
10   ⌈
11    for unmarked((l, r), (l', r')) ∈ G.edges() do
12         mark((l, r), (l', r'))
13         if (l', r') instanceof TempTaggedRegion
14           then
15                 descendants(l, r) = descendants(l, r) ∪ {(l', r')}∪
16                 ∪ recDescendants((l', r'))
17            else
18                 descendants(l, r) = descendants(l, r) ∪ {(l', r')}∪
19                 ∪ descendants((l', r'))
20           od
21           return descendants(l, r)
22         ⌋
23    .
24   end
```

Figure 12.12: Algorithm for computing descendants for synthetic regions

```
1  tempAncestors(TaggedRegionGraph G)
3  begin
4    List tempRegionList = G.regionList()
5    for (l, r) ∈ tempRegionList do
6         for (l', r') ∈ descendants(l, r) do
7              ancestors(l', r') = (l', r') ∪ {(l, r)}
8         od
9    od
10 end
```

Figure 12.13: Algorithm for computing ancestors for synthetic regions

```
1  computeNafos(TaggedRegionGraph G)
3  begin
4     List tempRegionList = G.regionList()
5     for (l, r) ∈ tempRegionList do
6         for ((l, r), (l', r')) ∈ G.edges() do
7             N_afo((l, r), (l', r')) = N_el((l', r'))/N_el(l)(r)
8         od
9     od
10 end
```

Figure 12.14: Algorithm for computing N_{afo} information for synthetic regions

```
1  DJoinEstimation(PhyOp op): ETLS
3  begin
4     Node leftInput = op.getLeft()
5     Node rightInput = op.getRight()
6     ETLS etls_1 = estimation(leftInput)
7     rightInput.passThrough(leftInput)
8     return estimation(rightInput)
9  end
```

Figure 12.15: Size estimation algorithm for XMLDependentJoin (the function *passThrough* is used for pushing down input ETLSs in the query plan)

and by invoking the predicate estimation algorithm.[3] The size equation for the operator is shown below.

$$ES(XMLNestedLoopJoin(P), etls_1, etls_2) \equiv applypsf(P, etls_1 \odot etls_2)$$

XMLDependentJoin XMLDependentJoin is a physical operator directly implementing the *DJoin* operator of the logical query algebra. It mainly acts as a tuple flow controller, by using the tuples produced by its left input to instantiate and evaluate its right input. This reflects on the size estimation algorithm, which uses the left input ETLS to instantiate the result size estimation of its right input. The algorithm performing the size estimation for XMLDependentJoin is shown in Figure 12.15.

[3] This requires the size estimator to access the selectivity factor for the join predicate: as for any predicate estimation, if no selectivity factor for the join predicate is available, the estimation algorithm exploits magic numbers.

XMLUnion XMLUnion takes as input two homogeneous streams of tuples, and it returns their union. For evaluating the size of the result of this operation, it is necessary to join the sequences of ECLSs in the two input ETLSs. The size equation for XMLUnion is shown below.

$$ES(XMLUnion(), etls_1, etls_2) \equiv$$
$$\bullet_{v_i \in ungrouped(etls_1)} \{(v_i : \langle unnest(etls_1(v_i)) \uplus unnest(etls_2(v_i)) \rangle\} \bullet$$
$$\bullet_{v_i \in grouped(etls_1)} \{(v_i : etls_1(v_i) \cup etls_2(v_i))\}$$

The sequences of ECLSs bound to *grouped* variables, i.e., variables with binder =, are concatenated, in order to preserve groups; for ungrouped variables, instead, the system flattens such sequences, and combines ECLSs via the $\uplus$ support function.

The time complexity of the corresponding algorithm is determined by the union of ungrouped variables, and it is equal to $O(m+n)$, where n is the number of match occurrences in $etls_1$, and m is the number of match occurrences in $etls_2$.

XMLExtUnion XMLExtUnion takes as input two, possibly heterogeneous, tuple streams, and it returns their external union. The size equation for XMLExtUnion is similar to that of XMLUnion, with the only difference that variables appearing in only one ETLS must be treated in a different way. The size equation for XMLExtUnion is shown below.

$$ES(XMLExtUnion, etls_1, etls_2) \equiv$$
$$\bullet_{v_i \in ungrouped(etls_1) \setminus ungrouped(etls_2)} \{(v_i : etls_1(v_i))\}$$
$$\bullet_{v_i \in ungrouped(etls_2) \setminus ungrouped(etls_1)} \{(v_i : etls_2(v_i))\}$$
$$\bullet_{v_i \in grouped(etls_1) \setminus grouped(etls_2)} \{(v_i : etls_1(v_i))\}$$
$$\bullet_{v_i \in grouped(etls_2) \setminus grouped(etls_1)} \{(v_i : etls_2(v_i))\}$$
$$\bullet_{v_i \in ungrouped(etls_1) \cap ungrouped(etls_2)} \{(v_i : \langle unnest(etls_1(v_i)) \uplus unnest(etls_2(v_i)) \rangle)\} \bullet$$
$$\bullet_{v_i \in grouped(etls_1) \cap grouped(etls_2)} \{(v_i : etls_1(v_i) \cup etls_2(v_i))\}$$

The corresponding algorithm has $O(n+m)$ worst case time complexity, where n is the number of match occurrences in $etls_1$, and m is the number of match occurrences in $etls_2$.

Chapter 13

Experimental Results

This Chapter presents experimental results about the estimation model of Xtasy. These experimental results were gained by performing experiments on two distinct classes of XML databases: the DBLP database [dbl], and the XMark datasets [SWK+01].

The Chapter first will show experimental results about the space requirements of Xtasy statistics (the Tagged Region Graph and the path statistics). Then, six classes of benchmark queries for size estimators will be presented (see Appendix C for their code). Finally, accuracy tests on these classes of queries will shown.

13.1 Configuration

All experiments were performed on three datasets. The first and the second dataset are the Tiny (11.1 MBytes) and the Standard (110 MBytes) test documents of XMark [SWK+01], while the third dataset is the DBLP database [dbl] (127.6 MBytes). The DBLP document has a relatively flat structure, while XMark datasets have a quite rich and complex structure.

For all experiments, the reference machine was a PowerMac G4 733 with 384 MBytes of onboard memory, and a 5400 rpm 40 GBytes IDE hard drive; the system was running Mac OS X 10.2.6 and Java 1.4.1.

13.1.1 Space Requirement Experiments

The first experiments we performed explore the relation between the number of regions in the Tagged Region Graph (and their cardinality), and the δ parameter, which defines the width of the positional interval of a region (recall that Xtasy regions have the form $(h, [p, p + \delta_p])$, where h is a level in the tree, and $[p, p + \delta_p]$ is a positional interval).

As shown inf Figures 13.1 (XMark tiny dataset), 13.2 (XMark standard dataset),

and 13.3 (DBLP dataset)[1], only big values of δ_p make regions populous enough to be easily managed: for instance, the average region cardinality in the XMark Standard dataset reaches 1000 only for $\delta_p > 5000$. This, in turn, affects the number of regions in the graph: this number is small enough to keep the Tagged Region Graph size small only for big values of δ_p. These phenomena are due to the fact that the number of tagged regions in the graph is in $\Omega(h_{max} + t)$, where h_{max} is the maximal height of the tree, and t is the number of distinct tags in the tree. However, regions too populous may lead to **many** estimations errors.

The next class of graphs, shown in Figures 13.4 (XMark tiny dataset), 13.5 (XMark standard dataset), and 13.6 (DBLP dataset), illustrates the storage requirements for the Tagged Region Graph (statistics exceeding the allocated memory were flushed on secondary storage). As expected, low values of δ_p (which implies a high number of very small regions) produce big graphs, while great values δ_p produce very small and tractable region graphs.

These experiments show that Xtasy statistics are *space expensive*. This is due partly to the statistical model itself, and partly to the Java implementation. In particular, the programmer has no real control on what Java is really storing on secondary storage, hence common tricks used in C/C++ programs cannot be exploited. Nevertheless, we believe that the choice of purely extensional regions naturally leads to space expensive statistics, so the investigation of a mixed solution, where regions carry extensional as well as intensional information, is necessary.

13.1.2 Path Statistics

Experimental results for the size requirement of path statistics only are shown in Figures 13.7 (XMark tiny dataset), 13.8 (XMark standard dataset), and 13.9 (DBLP dataset). These graphs show that, in most cases, Xtasy path statistics are too big to be efficiently managed in main memory. This is due to the limitations imposed by the Java implementation, and to the characteristics of the statistical model. As shown in [AAN01], path statistics space requirements can be dramatically decreased by cutting off the least frequently used paths from the statistics.

13.2 Accuracy Tests

This Section presents accuracy tests for the model being described in this Thesis. In these experiments, benchmark queries were executed over the three previously described sample datasets, and the predictions provided by the result size estimator were compared with the actual query result size.

[1] All graphs shown in this Section use the logarithmic scale on both axes.

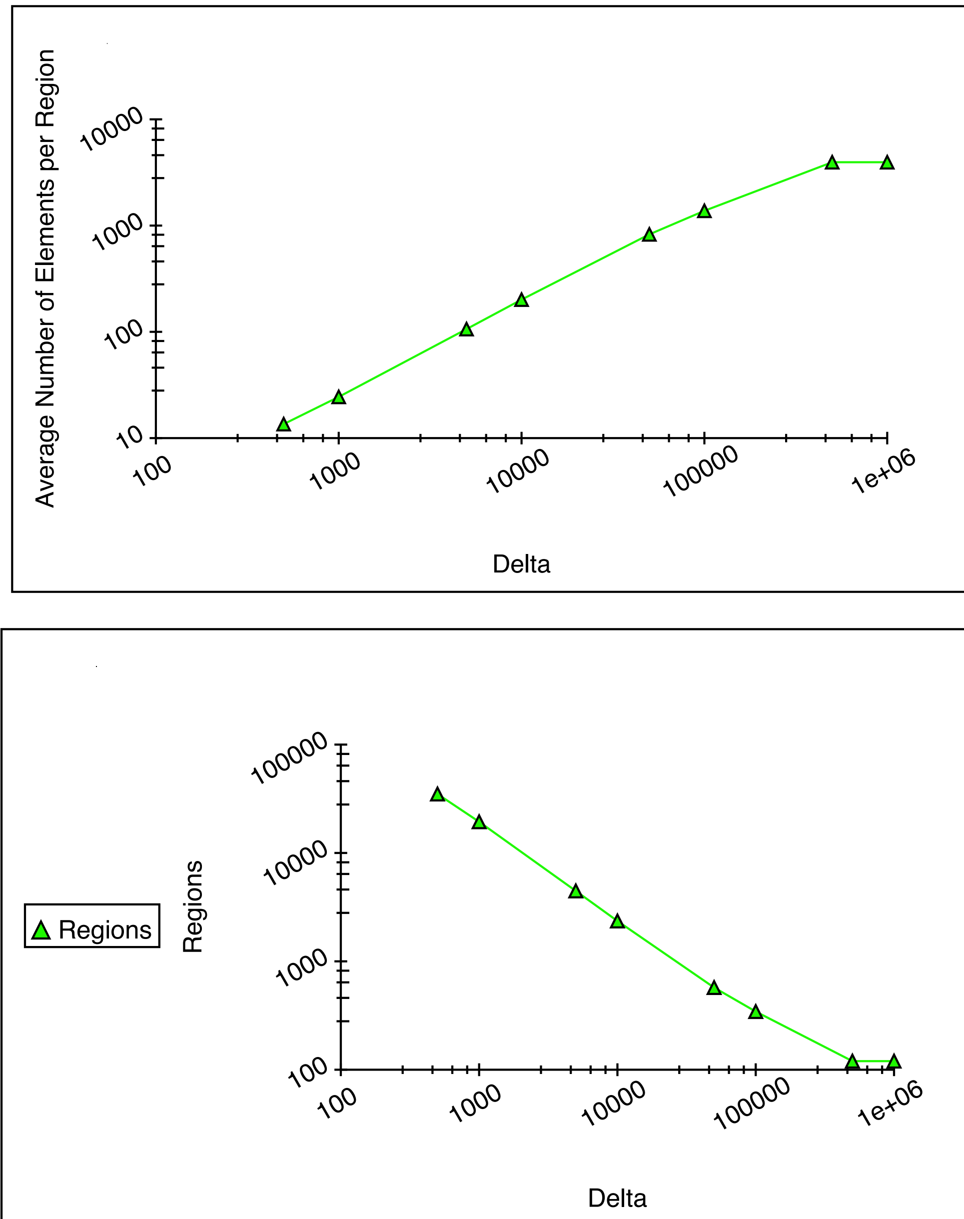


Figure 13.1: Region number and average N_{el} for the XMark tiny document

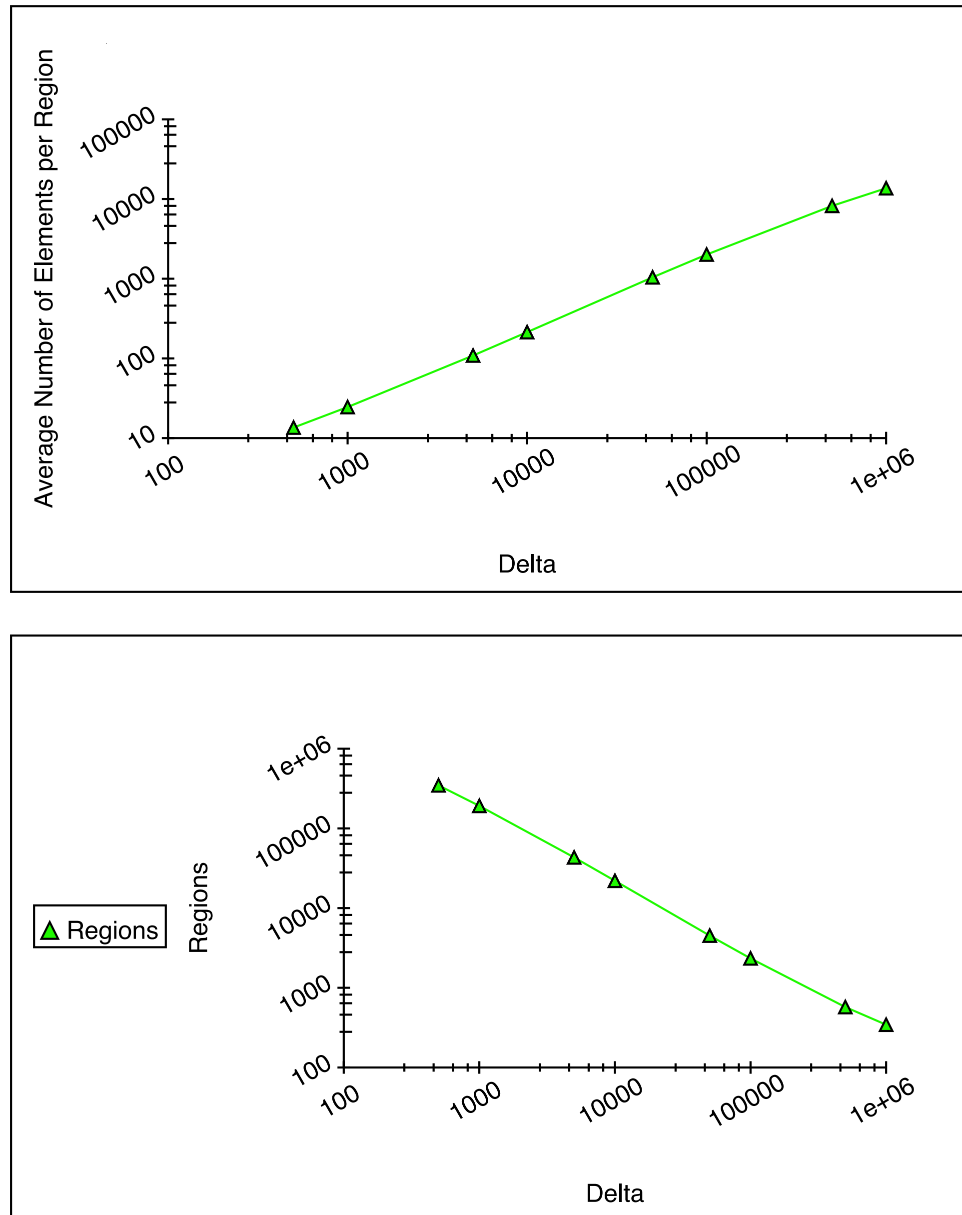
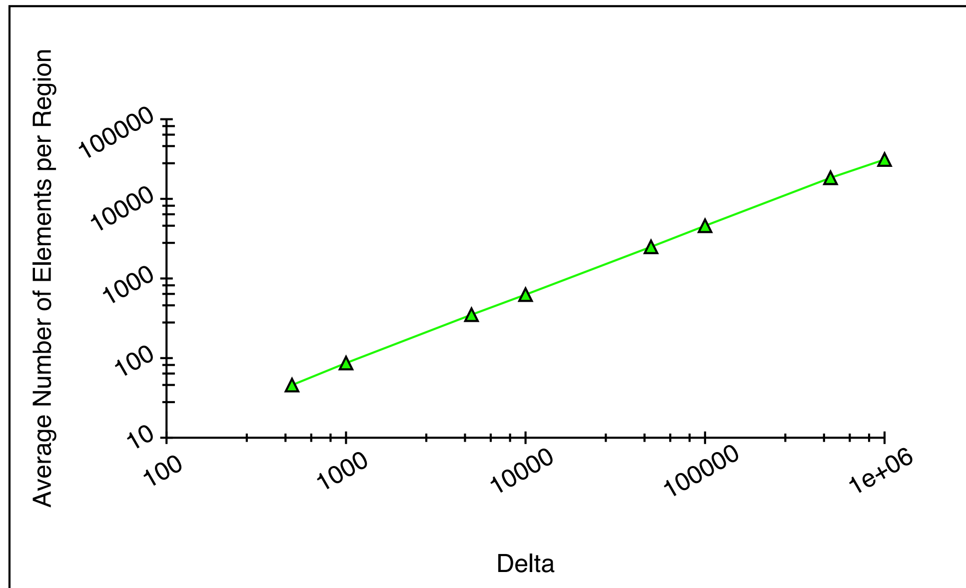


Figure 13.2: Region number and average N_{el} for the XMark standard document



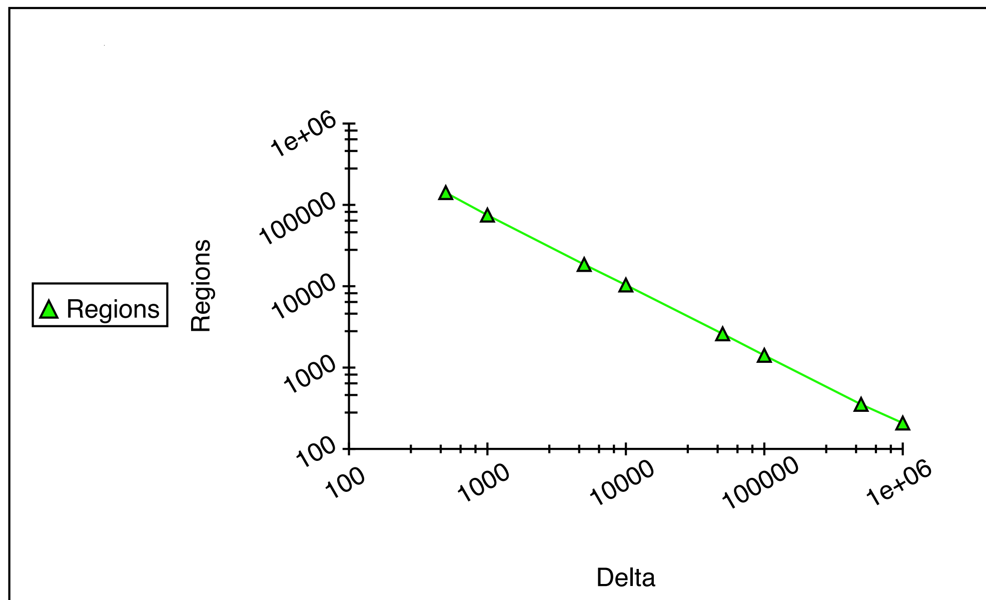


Figure 13.3: Region number and average N_{el} for the DBLP document

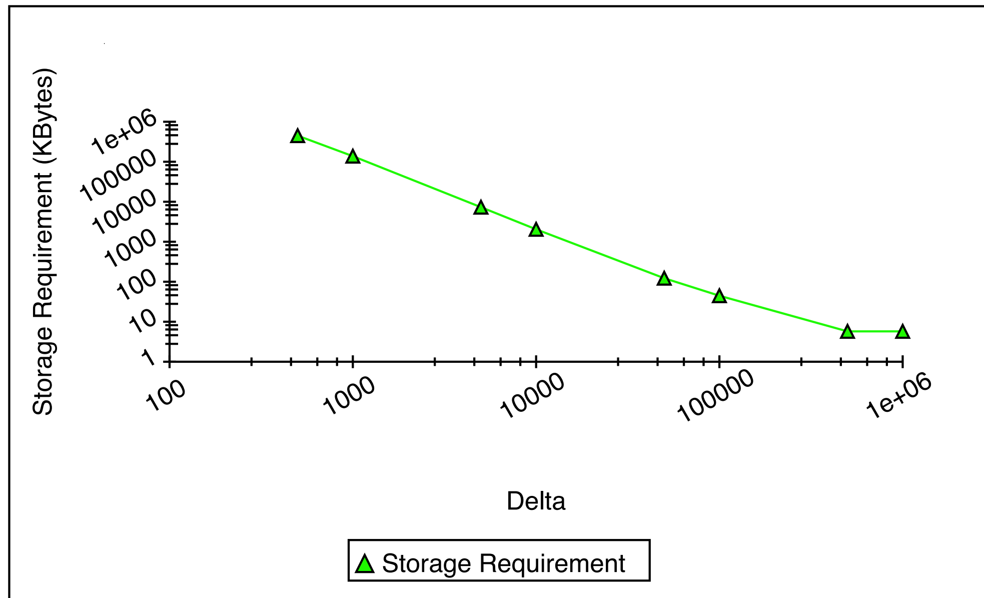


Figure 13.4: Space requirements of the Tagged Region Graph for the XMark Tiny document

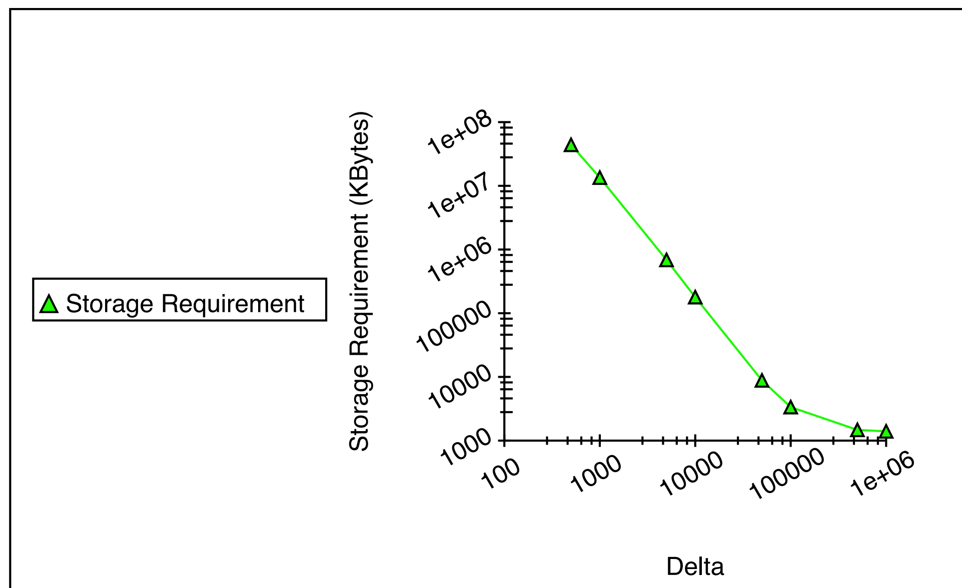


Figure 13.5: Space requirements of the Tagged Region Graph for the XMark Standard document

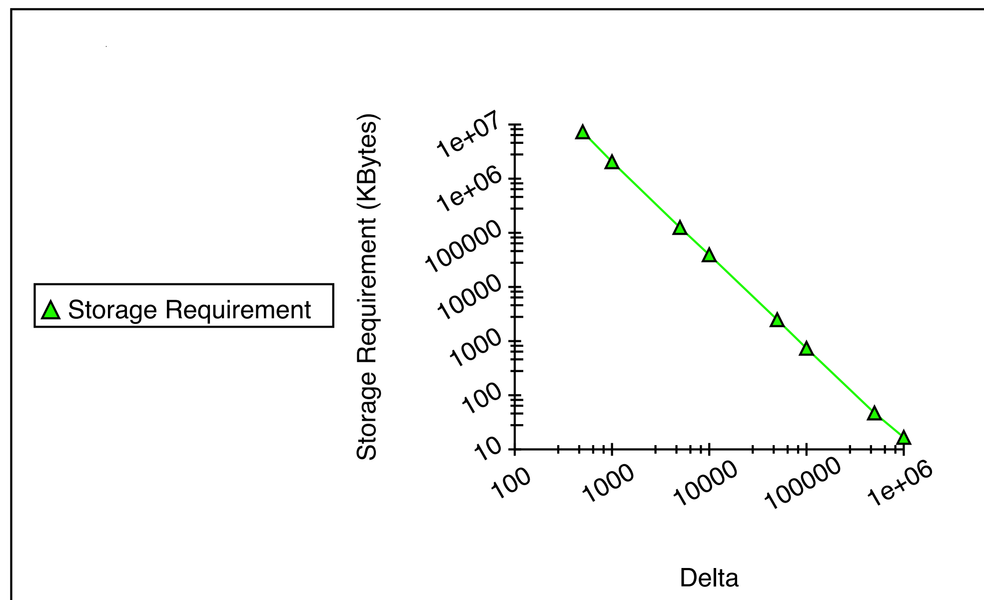


Figure 13.6: Space requirements of the Tagged Region Graph for the DBLP document

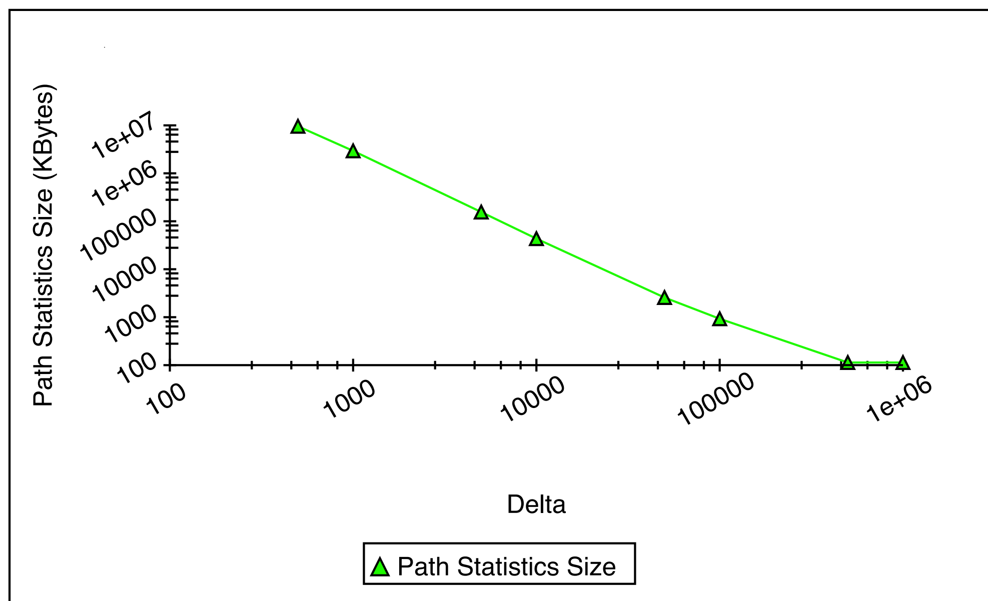


Figure 13.7: Path statistics size for the XMark tiny database

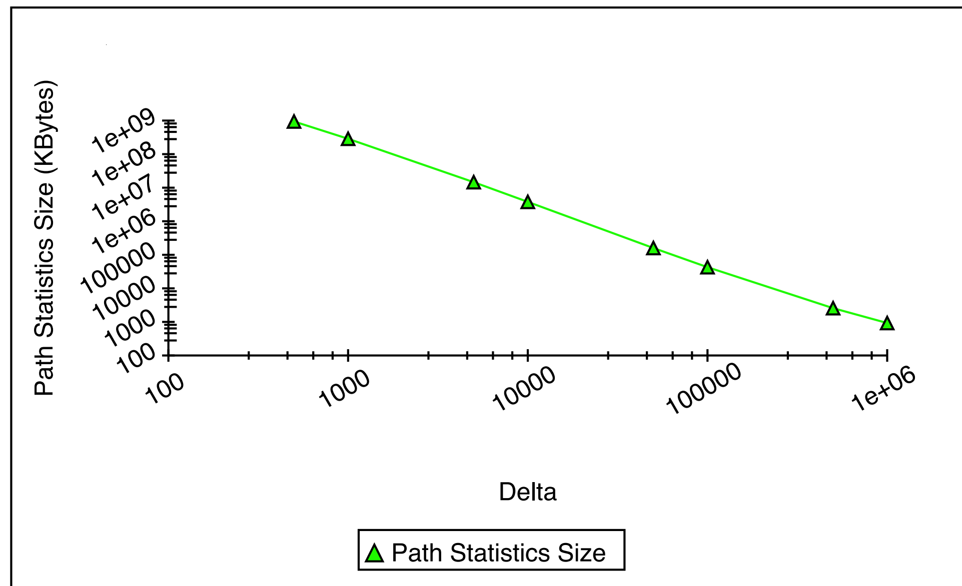


Figure 13.8: Path statistics size for the XMark standard database

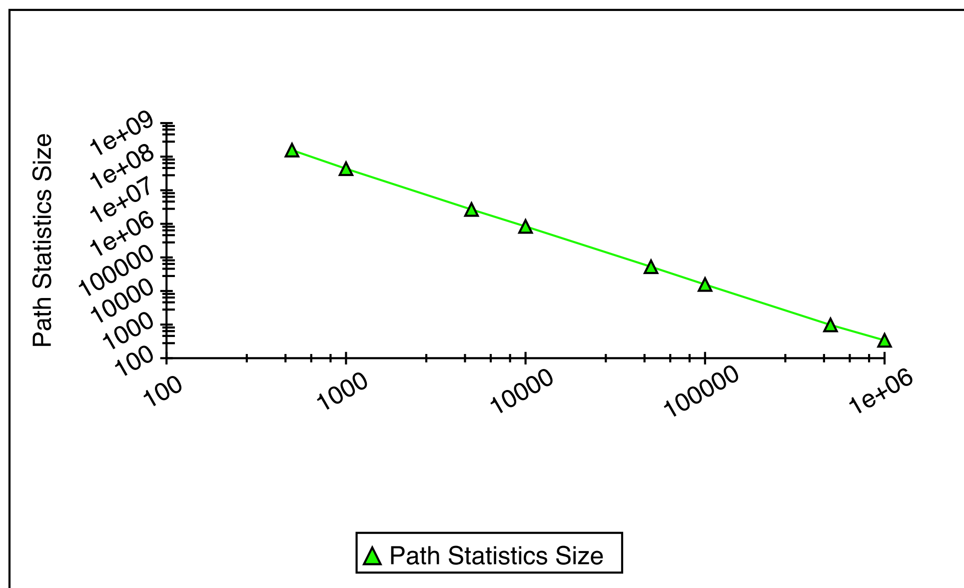


Figure 13.9: Path statistics size for the dblp database

13.2.1 Benchmark queries

The experiments performed to validate the prediction model are based on a set of benchmark queries, shown in full detail in Appendix C. Unlike usual XML database benchmarks (see [SWK+01] for instance), these queries were designed with the purpose of testing the accuracy of size predictions, hence they may turn to be inadequate for the purpose of speed testing.

The benchmark contains six sets of queries, each of them related to different size prediction issues. The first five sets contain *positive* queries, i.e., queries having non-empty result, while the last class is formed by *negative* queries, i.e., queries **with** empty result; this design choice, inspired by [CJK+01], is motivated by the will to test the accuracy of the model even in the worst conditions. We prefer to use only one class of negative queries since there is no unanimous consensus about the significance of this class of experiments (most papers on result size estimation show positive queries only).

Here follows a brief description of the benchmark query classes: the text of the query, for both the XMark datasets and the DBLP dataset, can be found in Appendix C.

Path queries These queries evaluate linear path expressions, both fully specified and with wildcards and closure operators.

Q11: linear path expression with no wildcards and closure operators;

Q12: linear path expression with wildcards but no closure operators;

Q13: linear path expression with closure operators but no wildcards;

Q14: linear path expression with wildcards and closure operators.

Twig queries These queries evaluate twig expressions, both fully specified and with closure operators; the twigs do not contain the grouping binder.

Q21: twig expression with no closure operators;

Q22: twig expression with closure operators.

Twig queries with groups These queries evaluate complex twig queries with the grouping binder (`let ... :=`).

Q31: twig expression with a grouping binder on the twig leaves;

Q32: twig expression with a grouping binder on an intermediate twig node.

Queries with predicates The queries in this class apply unary predicates to the result of twig expressions.

Q41: twig expression with a range predicate;

Q42: twig expression with an equality predicate.

Nested queries These queries contain nested queries used for reshaping elements and nodes.

Q51: query with a single inner query;

Q52: query with predicates and multiple inner queries.

Negative queries These queries are used for estimating the accuracy of size predictions on empty queries.

Q61: structurally empty query, i.e., no match for the twig in the query;

Q62: query with an empty predicate, i.e, no match satisfies the predicate.

13.2.2 Error metrics

We use two error metrics for evaluating the accuracy of the size model: the *relative* error for positive queries, and the *absolute* error for negative queries, as shown below.

$$\begin{aligned} RE(Q) &= \frac{ES(Q) - \|Q\|}{\|Q\|} \\ AE(Q) &= ES(Q) - \|Q\| \end{aligned}$$

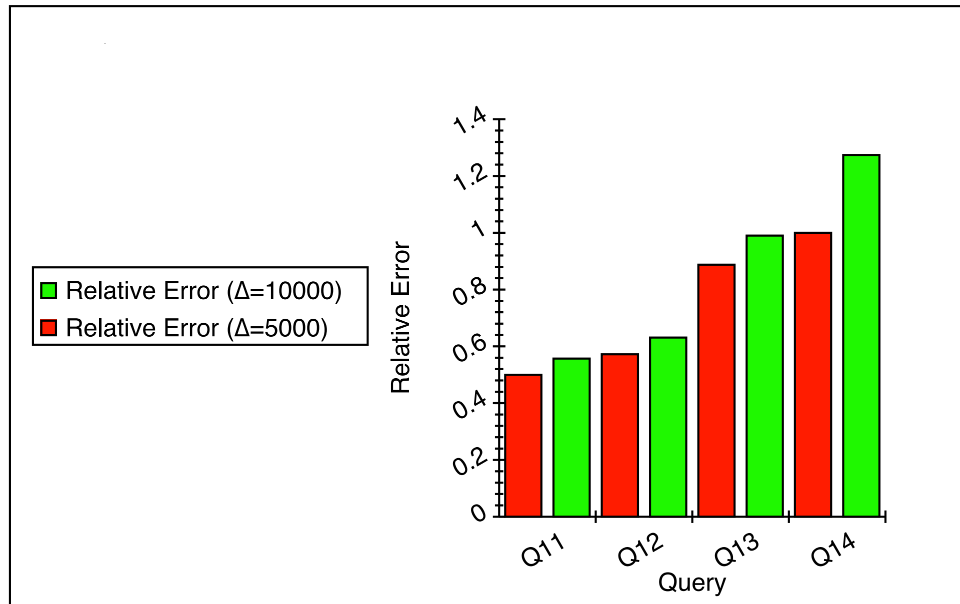
13.2.3 Experimental results

For each dataset, we perform the benchmark queries on two distinct statistical configurations: for the XMark standard document and for the DBLP dataset, we use $\delta_p = 100000$ and $\delta_p = 500000$, while for the tiny dataset we use $\delta_p = 5000$ and $\delta_p = 10000$. For predicate queries, we estimate the accuracy of the size model with and without proper histograms. In the following graphs, relative errors will be shown in the form of percentage relative errors.

Path queries Experimental results for path queries are shown in Figure 13.10 (XMark tiny and standard dataset), and in Figure 13.11 (DBLP database).

The size estimator virtually introduces no error for queries without //; for queries with //, instead, the system produces some minor errors in the most compressed statistical configurations. Other existing models show similar results.

Twig queries Experimental results for twig queries are shown in Figure 13.12 (XMark tiny and standard dataset), and in Figure 13.13 (DBLP database). Unlike path queries, twig queries are more prone to estimation errors, due to the need to perform correlation checks: in particular, when regions contain many nodes, the system overestimates the number of correlated regions, hence leading to a significant estimation error (7.578%).



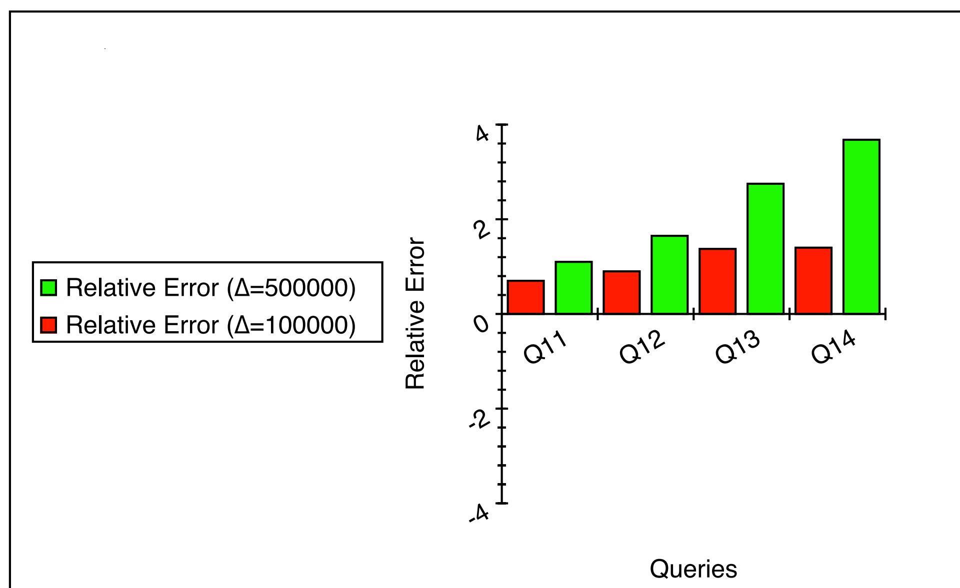


Figure 13.10: Accuracy tests for path queries (XMark tiny and standard documents)

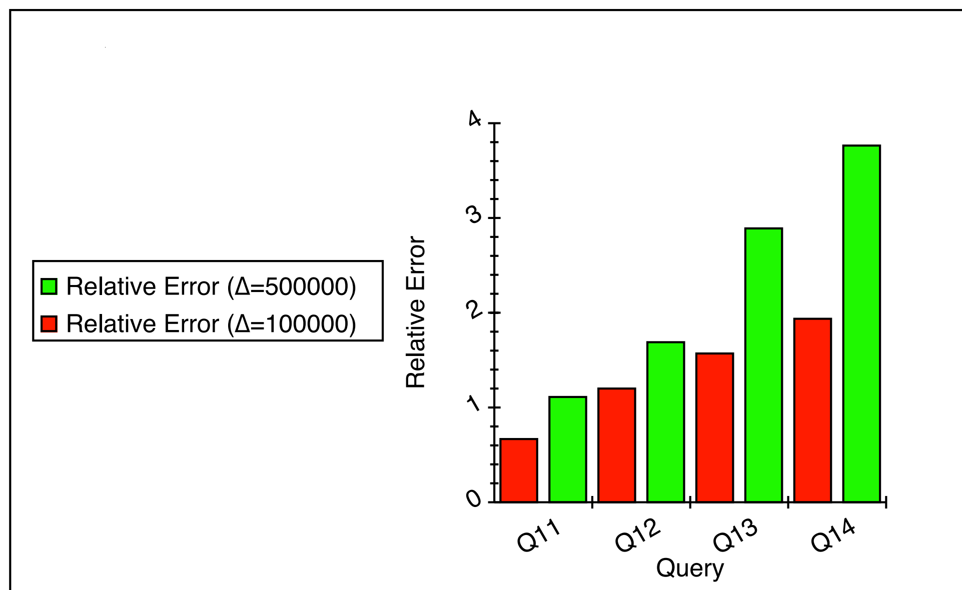


Figure 13.11: Accuracy tests for path queries (DBLP document)

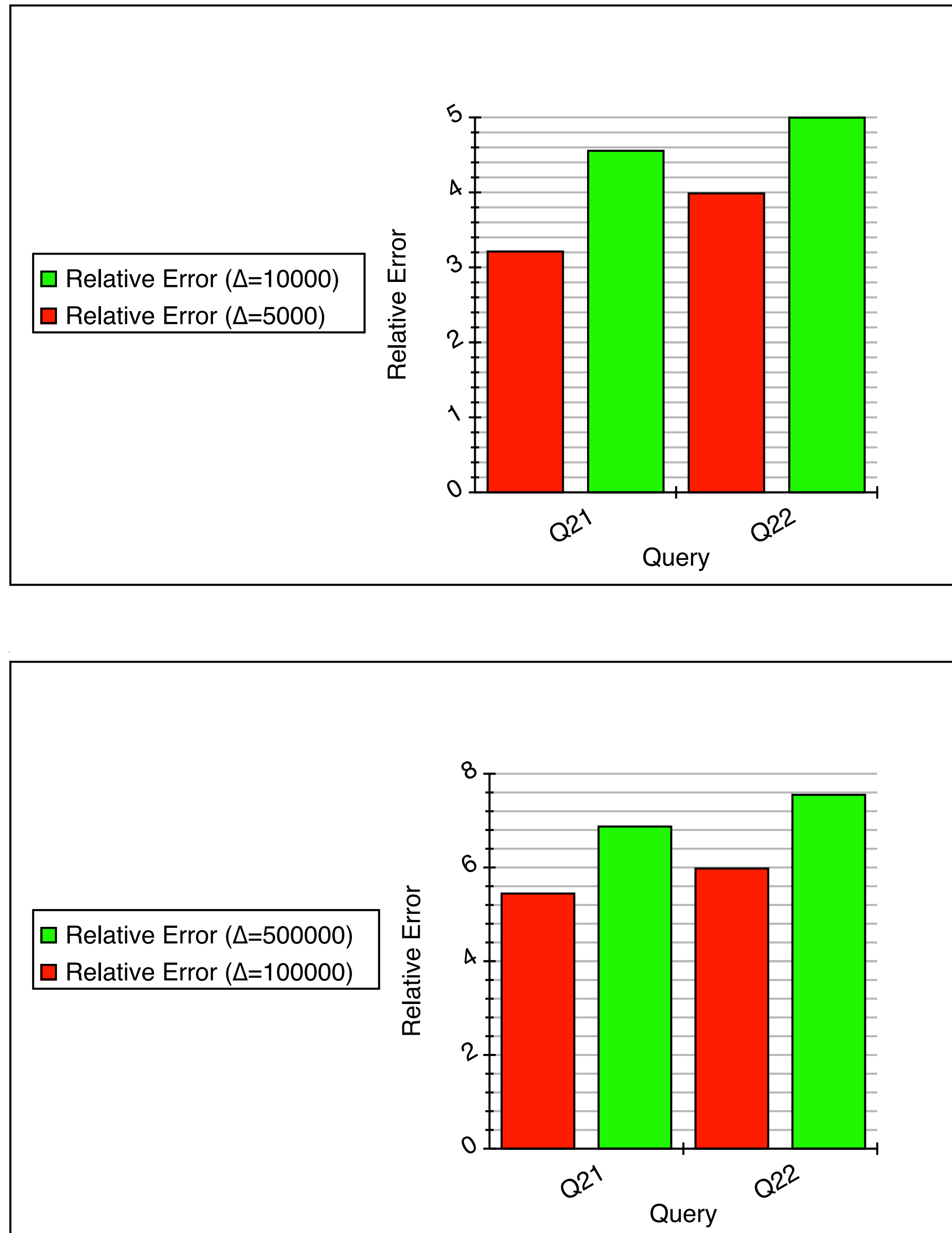


Figure 13.12: Accuracy tests for twig queries (XMark tiny and standard documents)

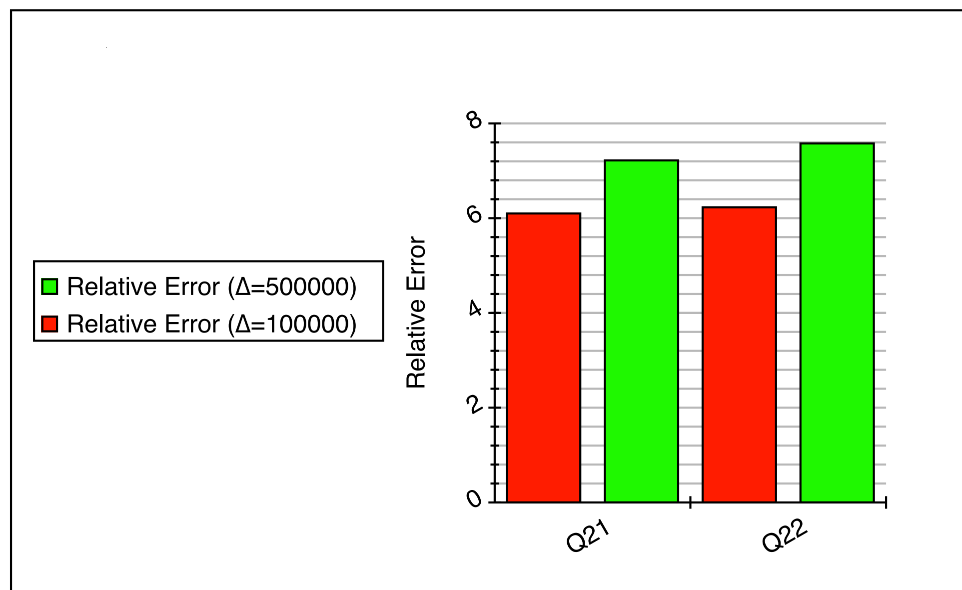


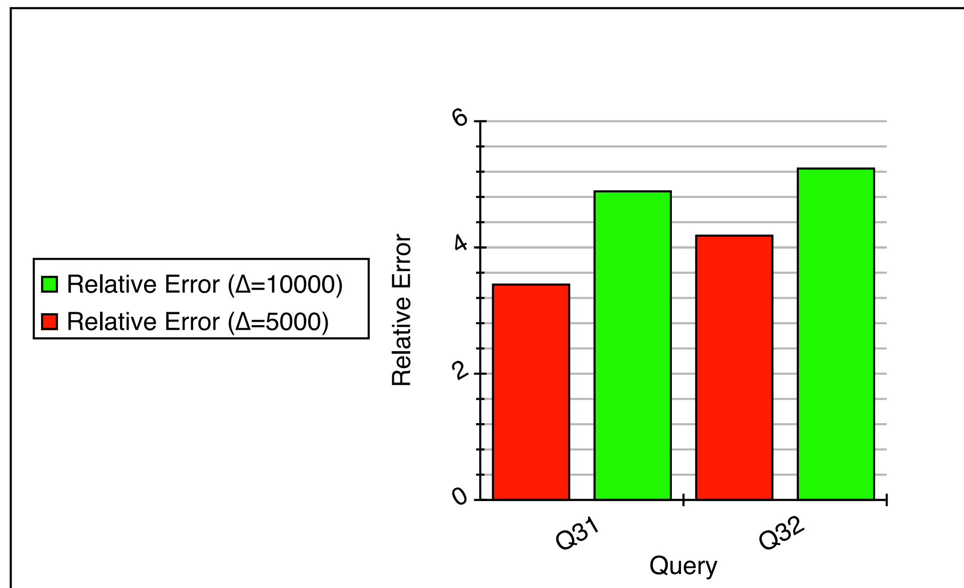
Figure 13.13: Accuracy tests for twig queries (DBLP document)

Twig queries with groups While the previous tests show a *uniform* statistical behavior of the model, i.e., the model tends to overestimate query result size, this query workload shows that the distribution of match occurrences into groups may lead to unexpected underestimation and overestimation of the size of single, which in turn may lead to query cardinality errors. Underestimation errors for groups are not visible in the diagrams, since they are balanced by overestimation errors of other groups, hence they do not appear in aggregate error metrics, as those shown in Figure 13.14 (XMark tiny and standard dataset), and in Figure 13.15 (DBLP database).

Queries with predicates Experimental results for queries with predicates are shown in in Figure 13.16 (XMark tiny and standard dataset), and in Figure 13.17 (DBLP database). For each dataset, we tested these queries with or without a proper histogram. When an histogram is available, the system virtually introduces no further error in the estimation of `where` clauses; when no histogram is available, instead, the system generates huge estimation errors. These errors are determined by the use of magic numbers (i.e., 0.1 for equality predicates, and 0.33 for other predicates), which are not adequate in the XML setting.

Nested queries Experimental results for nested queries are shown in Figure 13.18 (XMark tiny and standard datasets) and in Figure 13.19 (DBLP database). Due to the complexity of these queries, we test them only with proper histograms for predicates. Experiments show that, while still accurate, the process for synthesizing new statistics can propagate estimation errors, in particular those errors related to the evaluation of twigs and *value-based* joins on heavy compressed statistical configurations.

Negative queries Experimental results for negative queries are shown in Figure 13.20 (XMark tiny and standard dataset), and in Figure 13.21 (DBLP database). In the case of the structurally empty query (**Q61**), the system correctly predicts a zero result size; in the case of emptiness induced by the `where` clause, instead, the absence of a proper histogram leads to an intolerably high error.



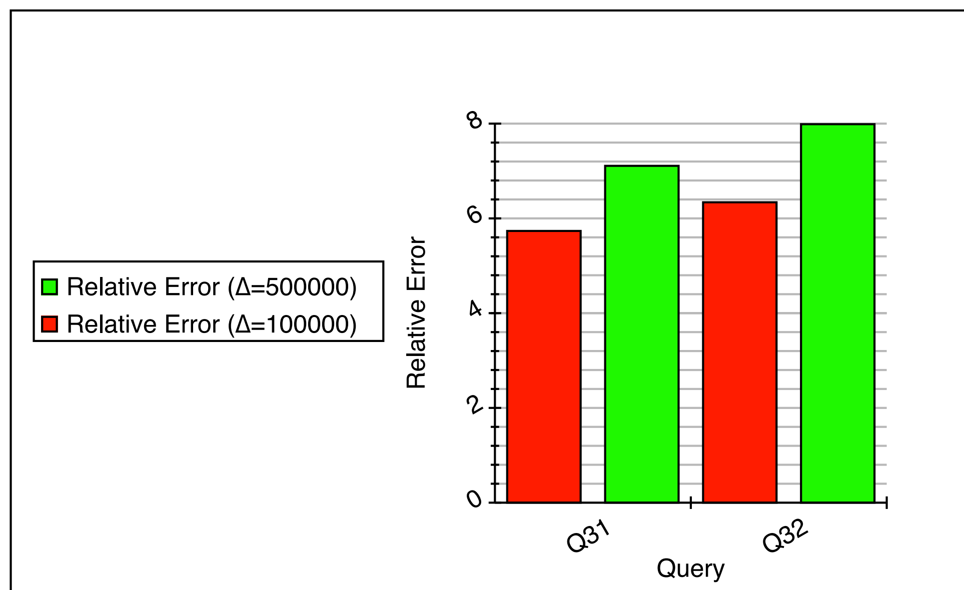


Figure 13.14: Accuracy tests for twig queries with groups (XMark tiny and standard documents)

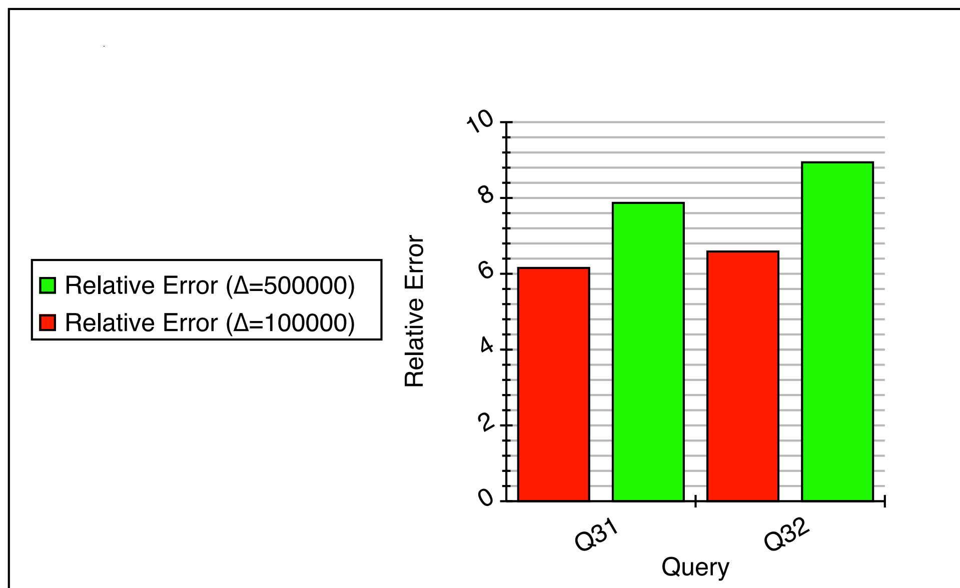


Figure 13.15: Accuracy tests for twig queries with groups (DBLP document)

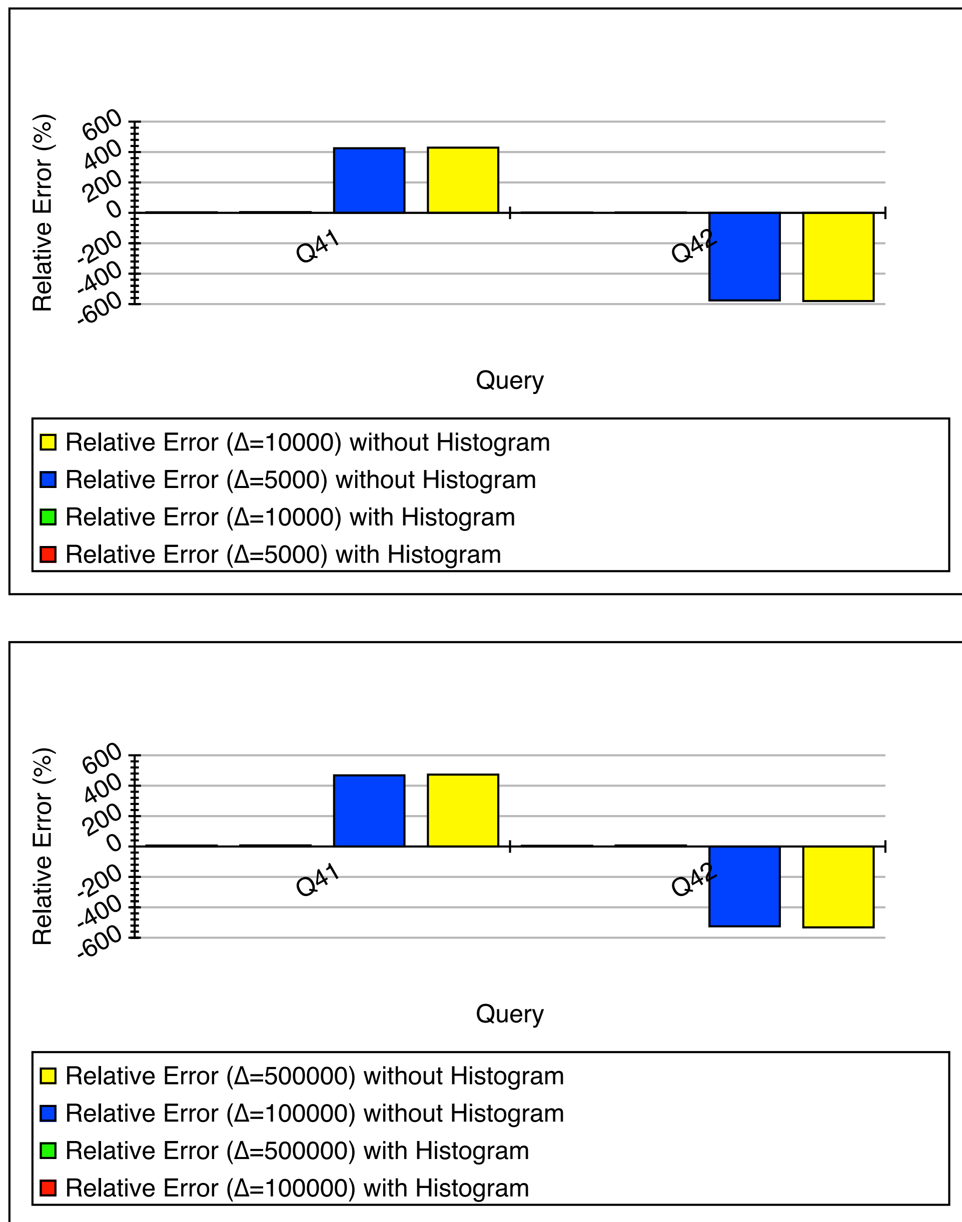


Figure 13.16: Accuracy tests for queries with predicates (XMark tiny and standard documents)

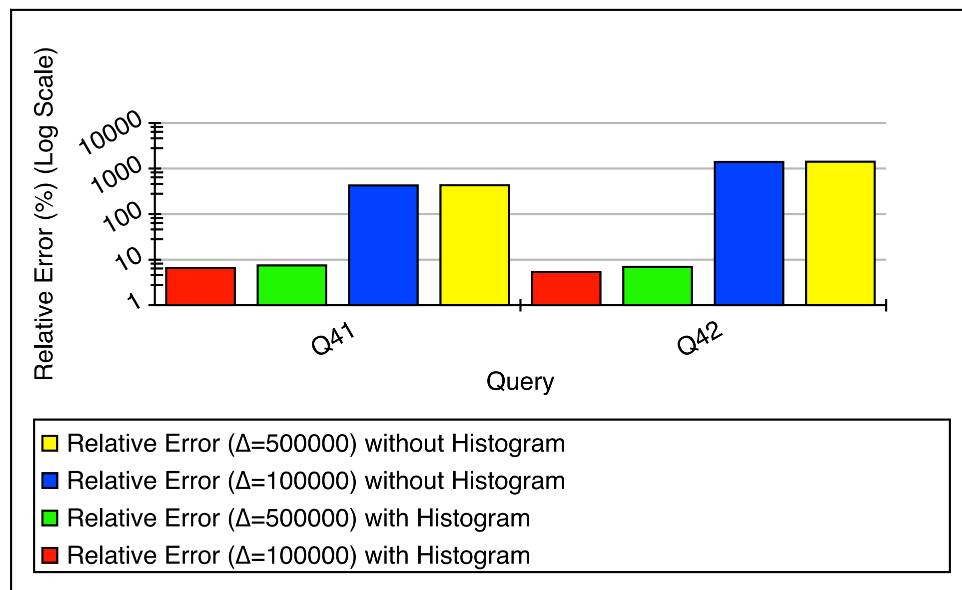


Figure 13.17: Accuracy tests for queries with predicates (DBLP document)

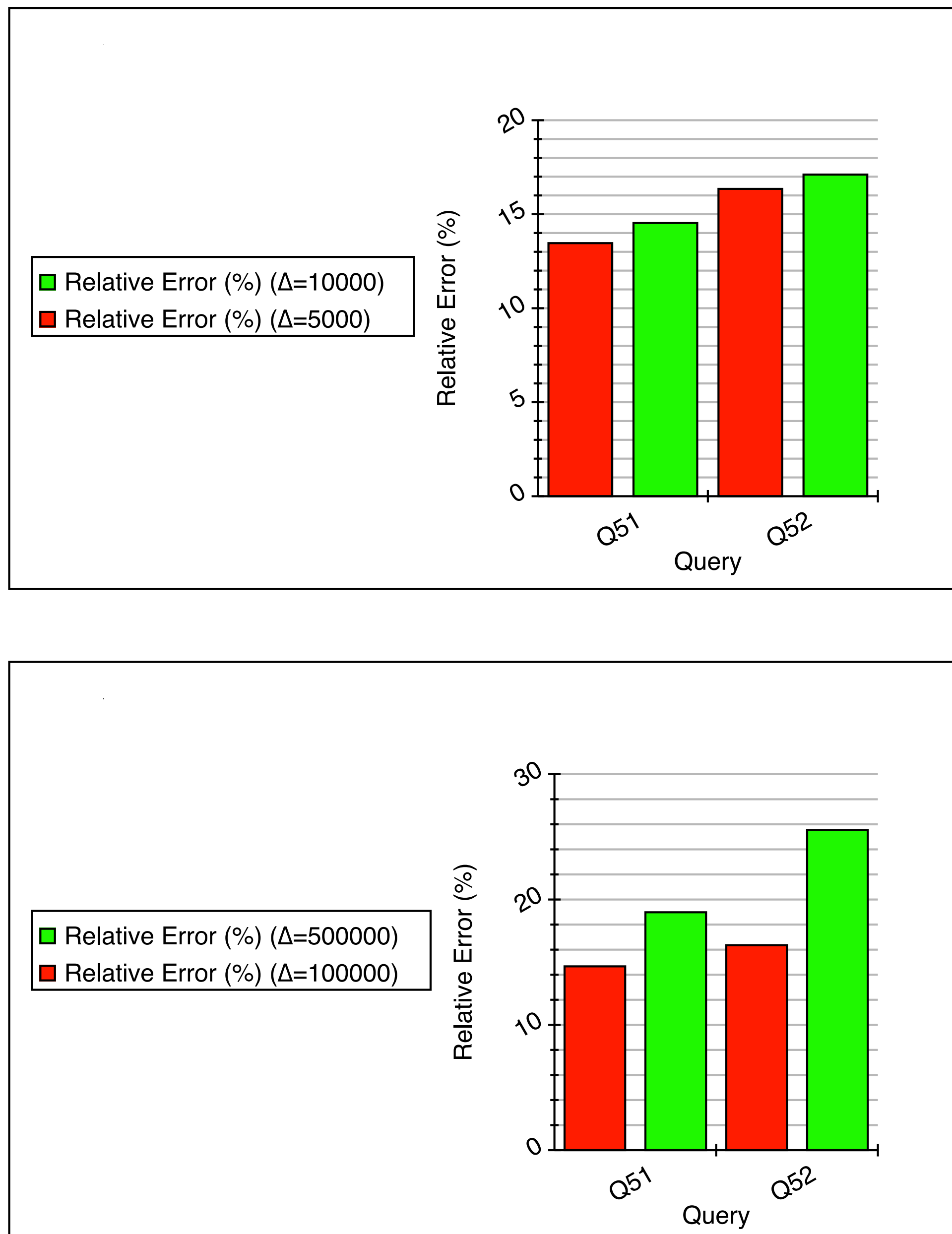


Figure 13.18: Accuracy tests for nested queries (XMark tiny and standard documents)

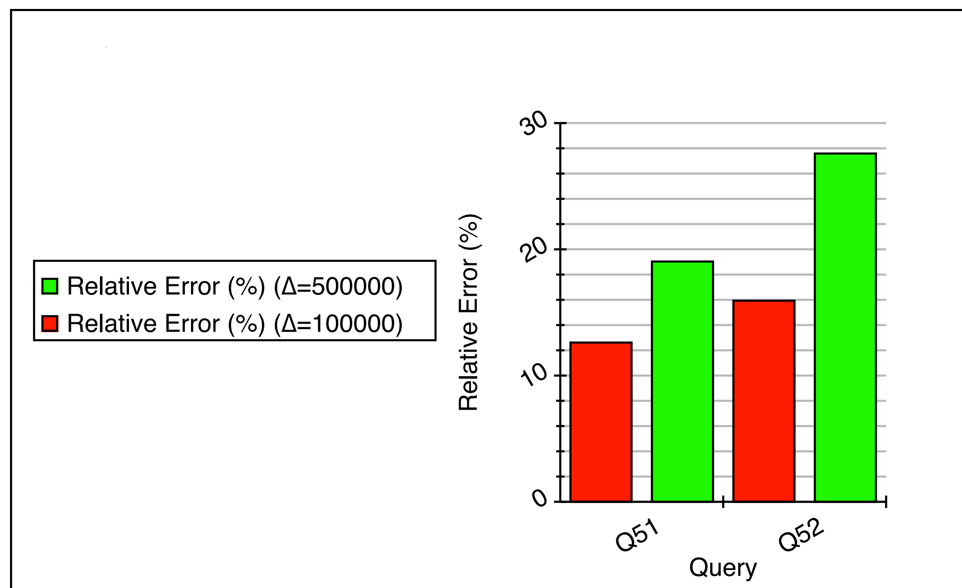


Figure 13.19: Accuracy tests for nested queries (DBLP document)

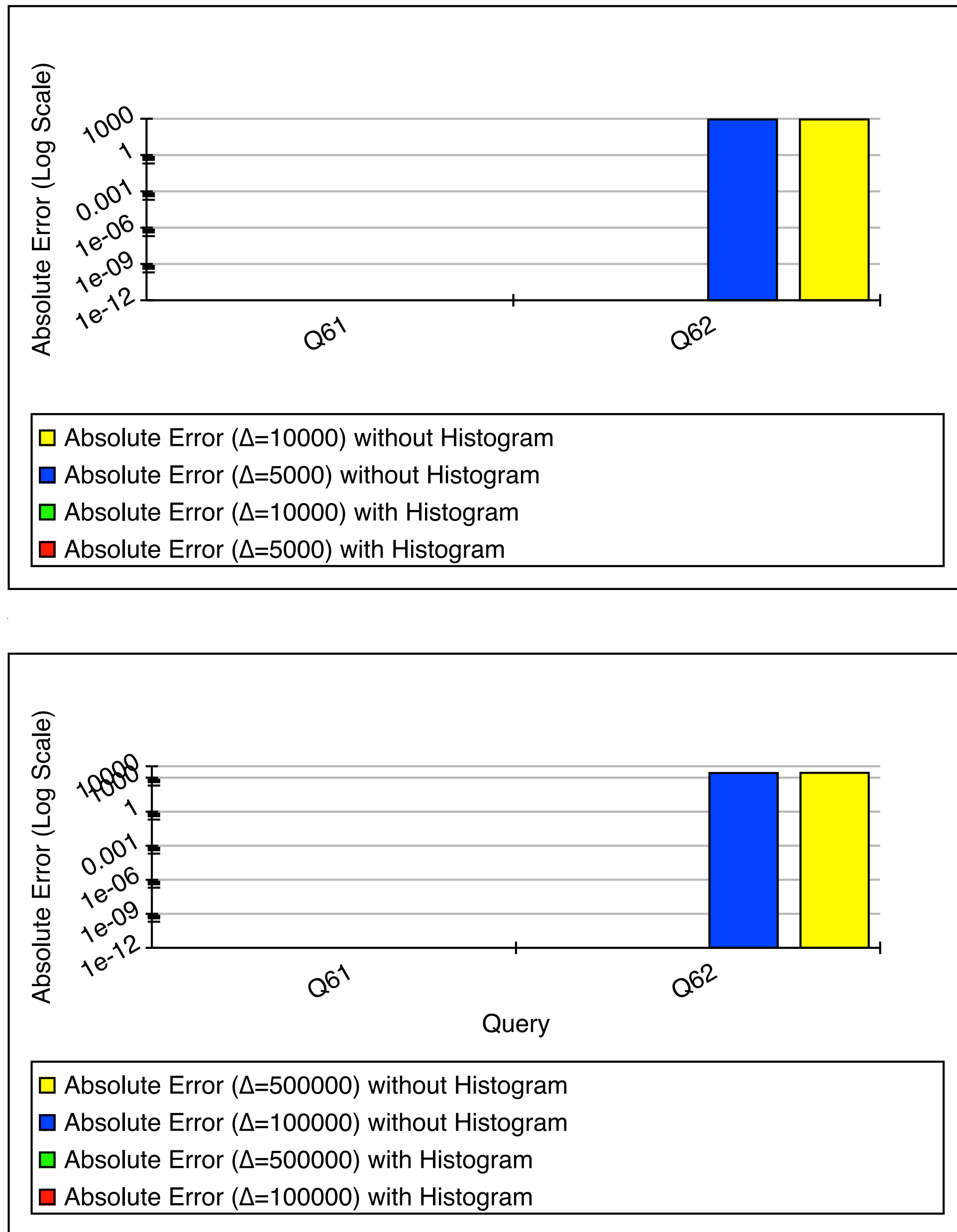


Figure 13.20: Accuracy tests for negative queries (XMark tiny and standard documents)

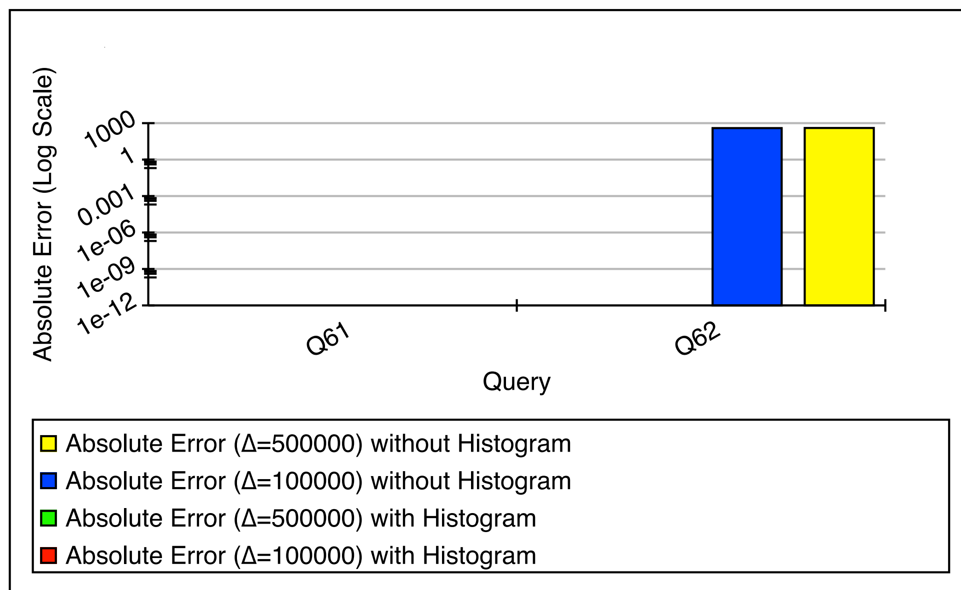


Figure 13.21: Accuracy tests for negative queries (DBLP document)

Chapter 14

Conclusions

THIS IS NOT AN EXIT
Bret Easton Ellis American Psycho

This Thesis presented a model for estimating the cardinality of XML queries. The model is based on the idea of predicting the distribution of data into query results, and then using such distribution as input for further estimation operations, or for computing an aggregate ("raw") measure of cardinality.

The estimation model was built on top of an experimental database management system, whose architecture, logical and physical query algebras, and storage model were described in the Thesis.

The definition of the estimation model started with the identification of the requirements of a prediction model for XML queries; these requirements leaded to the design of a framework that, by abstracting from specific statistical models, offers prediction services, such as twig branch correlation, group cardinality estimation, and predicate selectivity factor propagation.

The statistical model of Xtasy, then, was designed as an instance of this framework; the model proved to be accurate, at the price of relatively expensive space requirements for statistics.

Though covering the FLWR fragment of XQuery, the proposed model is still limited. In particular, the model does not deal with recursion nor with universally quantified predicates: these are significant limitations that we hope to overcome in a near future.

The estimation model described in this Thesis, together with the underlying database system, is only the starting point for a wider research. We plan to extend this work in various ways, as briefly discussed below.

Improvement of the database core The current version of Xtasy is written in Pure Java. Despite the claims about the robustness of the JVM, the virtual machine showed memory management problems, which directly affected the stability and

the scalability of Xtasy. Moreover, Java does not allow the programmer to control what is really written to secondary storage, hence preventing the use of some space optimizations.

To overcome these limitations, we plan to rewrite the core of Xtasy, i.e., the PSM and the Catalog Manager, by using C, C++, or Objective-C: this solution should lead to substantial gains in stability, scalability, and raw performance.

Development of a cost-based query optimizer The current version of Xtasy contains a *pseudo-random* query plan generator, used in place of the query optimizer; earlier efforts in the implementation of a query optimizer were frustrated by the lack of accuracy of the estimations provided by earlier versions of the size estimator, as well as by the difficulties in estimating the number of I/O operations performed by the system (this is a consequence of mediation of the JVM and the operating system).

By relying on the estimation model described in this Thesis, and on the rewriting of the core of Xtasy, we plan first to design a cost model for the physical operators of Xtasy, and then to exploit this model in a query optimizer conforming to the Volcano architecture [Gra93].

Extension of the estimation model The current prediction model is still quite limited, since it does not cover recursive functions and universally quantified predicates. In the short term, we plan to improve the estimation for predicates without proper histograms, as well as to study the problem of estimating the selectivity of universally quantified predicates; in the long term, we plan to extend the logical and physical query algebras of Xtasy with operators for recursive functions, in the line of [LMS+93] and [Moe02], as well as to design correspondent size equations.

Bibliography

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 591–600. Morgan Kaufmann, 2001.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman, 1999.

[ACV+00] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, and Fanny Wattez. Querying xml documents in xyleme. In *Proceedings of ACM SIGIR 2000 Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[AM98] G.O. Arocena and A.O. Mendelzon. Weboql: Restructuring documents, databases and webs. *Proceedings of ICDE'98, Orlando, Florida*, 1998.

[BBM+01] Denilson Barbosa, Attila Barta, Alberto O. Mendelzon, George A. Mihaila, Flavio Rizzolo, and Patricia Rodriguez-Guianolli. Tox - the toronto xml engine. In *Proceedings of the International Workshop on Information Integration on the Web, Rio de Janeiro, Brazil, April 9-11, 2001*, pages 66–73, 2001.

[BCF+03] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, May 2003. W3C Working Draft.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of 5th International Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.

[BFH+02] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Legodb: Customizing relational storage for xml documents. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 1091–1094, 2002.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE 2002, Proceedings of 18th International Conference on Data Engineering, February 26 - March 1,2002, San Jose, California*. IEEE CS, 2002.

[BM02] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Technical report, World Wide Web Consortium, May 2002. W3C Recommendation.

[BOS94] Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors. *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing. Springer, 1994.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extendible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation.

[BT99] Catriel Beeri and Yariv Tzaban. Sal: An algebra for semistructured data and xml. In *Proceedings of the ACM SIGMOD Workshop on The Web and Databases (WebDB'99), June 3-4, 1999*, Philadelphia, Pennsylvania, USA, June 1999.

[Bun97] Peter Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 117–121. ACM Press, 1997.

[CCMS98] Vassilis Christophides, Sophie Cluet, Guido Moerkotte, and Jérôme Siméon. Optimizing generalized path expressions using full text indexes. *Networking and Information Systems Journal*, 1(2):177–194, 1998.

[CCS98] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. Technical report, INRIA, Verso database group, November 1998.

[CDQT02] Agostino Cortesi, Agostino Dovier, Elisa Quintarelli, and Letizia Tanca. Operational and abstract semantics of the query language g-log. *Theoretical Computer Science*, 275(1-2):521–560, 2002.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 177–188, New York, June 1998. ACM Press.

[CG01] Luca Cardelli and Giorgio Ghelli. A query language for semistructured data based on the ambient logic. In *Proceedings of the 10th European Symposium on Programming, ESOP 2001, Held as part of the Joint Eu-*

ropean Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, pages 1–22, 2001.

[CJK+01] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 595–604. IEEE Computer Society, 2001.

[CKM02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic xml trees. In *Proceedings of PODS 2002*. ACM Press, 2002.

[CKMP97] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Jarke et al. [JCD+97], pages 286–295.

[CKN03] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey F. Naughton. On Relational Support for Xml Publishing: Beyond Sorting and Tagging. In Halevy et al. [HID03], pages 611–622.

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 646–648, Los Altos, CA 94022, USA, 2000. Morgan Kaufmann Publishers.

[Cla99] James Clark. XSL Transformations (xslt) Version 1.0. Technical report, World Wide Web Consortium, Nov 1999. W3C Recommendation.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In Beeri et al. [BOS94], pages 226–242.

[CM94] Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bases. Technical report, University of Karlsruhe, 1994.

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 2000.

[dbl] `http://dblp.uni-trier.de/xml/`.

[Dea01] S. Deach. Extensible Stylesheet Language XSL Specification. Technical report, World Wide Web Consortium, Oct 2001. W3C Recommendation.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, August 1998. Submission to the World Wide Web Consortium.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for xml. *Computer Networks*, 31(11-16):1155–1169, 1999.

[DFF+03] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, may 2003. W3C Working Draft.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 431–442, New York, June 1–3 1999. ACM Press.

[DTCÖ03] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In Halevy et al. [HID03], pages 623–634.

[FFK+97] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. STRUDEL: A Web site management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):549–560, 1997.

[FFK+98] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, and Dan Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 414–425. ACM Press, 1998.

[FHR+02] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. Statix: Making xml count. In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002, USA*. ACM Press, 2002.

[FKMS01] Paolo Ferragina, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Two-dimensional substring indexing. In *Proceedings of the Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*, 2001.

[FKS+02] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, December 2002.

[FM00] Thorsten Fiebig and Guido Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings of the third International Workshop WebDB 2000*, pages 125–136, 2000.

[FMM+03] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model. Technical report, World Wide Web Consortium, May 2003. W3C Working Draft.

[FSW01] Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 263–300. Springer, 2001.

[gal] `http://db.bell-labs.com/galax/`.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the second International Workshop WebDB '99, Pennsylvania*, June 1999.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HID03] Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[HP00] Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed xml processing language (preliminary report). In *Proceedings of the Third International Workshop on the Web and Databases, WebDB 2000, Adam's Mark Hotel, Dallas, Texas, USA, May 18-19, 2000, in conjunction with ACM PODS/SIGMOD 2000*, pages 111–116, 2000.

[ips] `http://xml.ipsi.fhg.de/xquerydemo/`.

[JCD+97] Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors. *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997.

[JLST01] H.V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for xml. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01), Frascati, Rome, September 8-10, 2001*, 2001.

[LMS+93] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The aqua data model and algebra. In Beeri et al. [BOS94], pages 157–175.

[MH01] Guido Moerkotte and Sven Helmer. Building Query Ccompiler (Draft), 2001. Manuscript Draft.

[Moe02] Guido Moerkotte. Incorporating XSL Processing into Database Engines. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.

[PHH92] H. Piranesh, J. Hellerstein, and W. Hasan. Extensible/rule-based query rewrite optimization in starburst. In *SIGMOD 1992, Proceedings of ACM SIGMOD Conference on Management of Data*, pages 39–48, 1992.

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In Jarke et al. [JCD+97], pages 486–495.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 294–305. ACM Press, 1996.

[PWM96] Y. Papakonstantinou, J. Widom, and H.G. Molina. Object exchange across heterogeneous information sources. *Proceedings of IEEE Int. Conference on Data Engineering, Birmingham, England*, 1996.

[QRSU95] D. Quass, A. Rajaraman, Y. Sagiv, and J. Ullman. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, Singapore, December 4-7, 1995*, pages 319–344, 1995.

[RDF+99] Jonathan Robie, Eduard Derksen, Peter Fankhauser, Ed Howland, Gerald Huck, Ingo Macherius, Makoto Murata, Michael Resnick, and Harald Schning. XQL (XML Query Language). August 1999.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. Technical report, World Wide Web Consortium, December 1999. W3C Recommendation.

[SA02] Carlo Sartiani and Antonio Albano. Yet Another Query Algebra For XML Data. In Mario A. Nascimento, M. Tamer Özsu, and Osmar Zaïane, editors, *Proceedings of the 6th International Database Engineering and Applications Symposium (IDEAS 2002), Edmonton, Canada, July 17-19, 2002*, 2002.

[Sah00] Arnaud Sahuguet. KWEELT, the Making-of: Mistakes Made and Lessons Learned. Technical report, Department of Computer and Information Science - University of Pennsylvania, November 2000.

[Sar03] Carlo Sartiani. A Framework for Estimating XML Query Cardinality. In *Proceedings of the Sixth International Workshop on the Web and*

Databases (WebDB 2003), San Diego, California, June 12-13, 2003, 2003.

[SGM86] ISO 8879. Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML), 1986.

[Sim99] Jérôme Siméon. *Intégration de sources de données hétérogènes.* PhD thesis, Université Paris XI, 1999.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SWK+01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML Benchmark Project. Technical report, Centrum voor Wiskunde en Informatica, April 2001.

[tam] `http://www.tamino.com`.

[TBMM02] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, May 2002. W3C Recommendation.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating xml. In *SIGMOD 2001*, 2001.

[tim] `http://www.eecs.umich.edu/db/timber/`.

[WPJ02] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating answer sizes for xml queries. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Proceedings of the 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, 2002*, volume 2287 of *Lecture Notes in Computer Science*, pages 590–608. Springer, 2002.

Appendix A

Formalizations

A.1 Formalization

A.1.1 *Env* and tuples operations

Four basic operations are defined on *Env* structures and tuples.

1. $t.A = t_j$ where $A = label_j$ (where t is a tuple) (field extraction)
2. $t.\vec{A} = \{t_{i1}, \ldots, t_{ip}\}$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$ (repeated field extraction)
3. $t \downarrow \vec{A} = [label_{i1} : t_{i1}, \ldots, label_{ip} : t_{ip}]$ where $\vec{A} = (label_{i_1}, \ldots, label_{i_p})$
4. $\bullet$, a concatenation operator between tuples (known as *tup_concat* in other algebras).

A.1.2 Support operators

1. $e[x] = \{[x : t] \quad | \quad t \in e\}$
2. $child(t) =$
   (a) if $t = v_B$, then $child(t) = \{\}$
   (b) if $t = (oid)_[t_1, \ldots, t_n]$, then $child(t) = \{t_i \quad | \quad i \in 1, \ldots, n\}$
3. $descendant(t) = child(t) \cup \bigcup_{t_i \in child(t)} descendant(t_i)$
4. $self(t) = \{t_1, \ldots, t_n \quad | \quad t = t_1, \ldots, t_n\}$
5. $self - descendant(t) = self(t) \cup descendant(t)$
6. $nav(op)(label)(t) =$
   (a) if $op = (_)$, then $nav(op)(label)(t) = \{t_i \quad | \quad t = t_1, \ldots, t_n \wedge label(t_i) = label\}$
   (b) if $op = (/)$, then $nav(op)(label)(t) = \{t'_j \quad | \quad t = t_1, \ldots, t_n \wedge \exists i \in 1, \ldots, n : t'_j \in child(t_i) \wedge label(t'_j) = label\}$
   (c) if $op = (//)$, then $nav(op)(label)(t) = \{t'_j \quad | \quad t = t_1, \ldots, t_n, \exists i \in 1, \ldots, n : t'_j \in self - descendant(t_i) \wedge label(t'_j) = label\}$

A.1.3 Basic Operators

Map $\chi_f(e) = \{f(t) \quad | \quad t \in e\}$

Join $e_1 \bowtie^f_{Pred} e_2 = \{f(t_1, t_2) \quad | \quad t_1 \in e_1 \wedge t_2 \in e_2 \wedge Pred(t_1, t_2)\}$

TupJoin $e_1 \bowtie_{Pred} e_2 = \{t_1 \bullet t_2 \quad | \quad t_1 \in e_1 \wedge t_2 \in e_2 \wedge Pred(t_1, t_2)\}$

DJoin $e_1 < e_2 >= \{y \bullet x \quad | \quad y \in e_1, x \in e_2(y)\}$

Selection $\sigma_{Pred}(e) = \{t \quad | \quad t \in e, Pred(t)\}$

Projection $\pi_{\vec{A}}(e) = \{t \downarrow \vec{A} \quad | \quad t \in e\}$

GroupBy $GroupBy_{g;\,A;\,f_1;\,f;\,\theta}(e) = \{y.A \bullet [g : G] \quad | \quad y \in e, G = f(\{x \quad | \quad x \in e, f_1(x)\theta f_1(y)\})\}$ where $A \subseteq Att(e)$ and $g \notin Att(e)$

Sort

TupSort $TupSort_{(x_1,\ldots,\$x_n)}(e) = Sort_{<_{Tup}(\$x_1,\ldots,\$x_n)}(e)$ where:

$$
\begin{aligned}
<_{Tup} (\$x_1, \ldots, \$x_n)(u, v) = \quad & (pos(u.\$x_1) < pos(v.\$x_1))\vee \\
& (pos(u.\$x_1) = pos(v.\$x_1) \wedge pos(u.\$x_2) < pos(v.\$x_2)) \vee \ldots \vee \\
& (pos(u.\$x_1) = pos(v.\$x_1) \wedge \ldots \wedge pos(u.\$x_{n-1}) = pos(v.\$x_{n-1})\wedge \\
& pos(u.\$x_n) < pos(v.\$x_n))
\end{aligned}
$$

Rename $rename_{\$X\rightarrow\$Y}(e) = \{t \downarrow \overrightarrow{Att}(e) \;\; \$X \bullet [\$y : t.\$X] \mid \; t \in e\}$

UnnestVar

1. $unnest(l[t_1, \ldots, t_2] = t_1, \ldots, t_n$
2. $unnest(\emptyset) = \emptyset$
3. $unnestvar_{\$X}(e) = \{t \downarrow \overrightarrow{Att}(e) \setminus \$X \bullet [\$X : unnest(t.\$X)] \mid \; t \in e\}$

A.1.4 Path

Input filters grammar

(1)	$F ::= F_1, \ldots, F_n \quad | \quad F_1 \vee \ldots \vee F_n \quad | \quad (op, var, binder)label[F] \quad | \quad \emptyset$
(2)	$op \in \{/, //, _\}$
(4)	$var \in label \cup \{_\}$
(5)	$binder \in \{_, in, =\}$

$path_f(t) =$

1. if $f = f_1, \ldots, f_m$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_{f_1}(t)\ TupJoin(true) \ldots TupJoin(true)\ path_{f_m}(t)$
2. if $f = f_1 \vee \ldots \vee f_m$, then $path_f(t) = path_{f_1}(t)\ OuterUnion \ldots OuterUnion\ path_{f_m}(t)$
3. if $f = (_, _, binder)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
4. if $f = (_, _, binder)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(_)(label)(t))$;
5. if $f = (op, l, in)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = nav(op)(label)(t)[l]$;
6. if $f = (op, l, =)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}$;
7. if $f = (op, l, in)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \bigcup_{t_i \in nav(op)(label)(t)} \{[l : t_i]\} TupJoin(true) path_F(t_i)$;
8. if $f = (op, l, =)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{[l : nav(op)(label)(t)]\}\ TupJoin(true)\ path_F(nav(op)(label)(t))\}$;
9. if $f = (op, _, _)label[\emptyset]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = \{\}$;
10. if $f = (op, _, _)label[F]$ and $t = t_1, \ldots, t_n$, then $path_f(t) = path_F(nav(op)(label)(t))$;
11. $path_\emptyset(t) = \{\}$;

A.1.5 Return

Output filters grammar

$$(1) OF ::= OF_1, \ldots, OF_n \quad | \quad label[OF] \quad | \quad @label[val] \quad | \quad val$$
$$(2) val ::= v_B \quad | \quad var \quad | \quad \nu var$$

$return_{fo}(e) =$

1. if $fo = v_B$, then $return_{fo}(e) = \bigcup_{i=1}^{n} v_B$ where $e = \{t_1, \ldots, t_n\}$;
2. if $fo = var$, then $return_{fo}(e) = \{t.var \quad | \quad t \in e\}$;
3. if $fo = \nu var$, then $return_{fo}(e) = \{fresh(t.var) \quad | \quad t \in e\}$;
4. if $fo = @label[val]$, then $return_{fo}(e) = \bigcup_{i=1}^{n} @label[return_{val}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$;
5. if $fo = label[fo']$, then $return_{fo}(e) = \bigcup_{i=1}^{n} label[return_{fo'}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$;
6. if $fo = fo_1, \ldots, fo_n$, then $return_{fo}(e) = \bigcup_{i=1}^{n} return_{fo_1}(\{t_i\}), \ldots, return_{fo_n}(\{t_i\})$ where $e = \{t_1, \ldots, t_n\}$.

where

1. $fresh(t_1, \ldots, t_n) = fresh(t_1), \ldots, fresh(t_n)$
2. $fresh((oid)label[t]) = (nu(oid))label[fresh(t)]$
3. $fresh((oid)@label[v_B]) = (nu(oid))@label[v_B]$
4. $fresh(v_B) = v_B$

Appendix B

Proofs

B.1 Nested Query Rewriting Rules

Proposition B.1.1 *Predicate Dependency*

$$e_1 < (path_{(_,\$var,=)}(return_{of}(\sigma_{Pred(\$X,\$Z,\$Y)}(path_{f_1}(db))))) >$$

$$\equiv$$

$$\pi_{\overrightarrow{Att(e_1)};\ \$var}(unnestvar_{\$var}(rename_{\$_res \to \$var}(e_1 \bowtie_{Pred_{Glob}(\$X,\$Z)} (path_{f'}(return_{of'}(\sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db)))))))$$

where

- $of' = _nested[_result[of], _env[z_1[\$z_1], \ldots, z_k[\$z_k]]]$
- $f' = (_, \$_n, in)_nested[(/, \$_res, =)_result[\emptyset], (/, _, in)_env[(/, _, in)z_1[(/, \$z_1, =)_[\emptyset], \ldots, (/, _, in)z_k[(/, \$z_k, =)_[\emptyset]]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$, $\$z_1, \ldots, \$z_k \notin Att(e_1)$, $\$Y \notin Att(e_1)$, $z_1, \ldots, z_k \notin symbols(of)$

Proof: Before starting the proof, we need some useful lemma.

Lemma B.1.2 $rename_{\$x \to \$y}(path_f(e)) \equiv path_{f(\$y/\$x)}(e)$

Lemma B.1.3 $\sigma_{Pred(\$Z)}(path_{f,(/,_,in)_env[(/,_,in)z_1[(/,\$z_1,=)_[\emptyset]],\ldots,(/,_,in)z_k[(/,\$z_k,=)_[\emptyset]]]}(return_{a[f_1,_env[z_1[\$z_1],\ldots,z_k[\$z_k]]]}(e)))$

$$\equiv$$

$$path_f(return_{a[f_1]}(\sigma_{Pred(\$Z)}(e)))$$

Lemma B.1.4 $unnestvar_{\$var}(path_{f[(/,\$var,=)label[\emptyset]]}(return_{of[label[of']]}(e)))$

$$\equiv$$

$$path_{f[(/,\$var,=)_[\emptyset]]}(return_{of[of']}(e))$$

Given the left-linearity property of $< \cdot >$ (as well as of the right members of the rewriting rules), it suffices to prove the thesis on tuple singleton.

Let $e = \{t\}$. We have to prove that:

$$\{t\} < (path_{(_,\$var,=)}(return_{of}(\sigma_{Pred(\$X,\$Z,\$Y)}(path_{f_1}(db))))) >$$

$$\equiv$$

$$\pi_{Att(\{\vec{t}\});\ \$var}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(\{t\} \ltimes_{Pred_{Glob}(\$X,\$Z)} (path_{f'}(return_{of'}(\sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db))))))))$$

Let $e_\sigma = \sigma_{Pred_{Loc}(\$Y,\$Z)}(path_{f_1}(db))$. Then, by using the common decomposition property of σ, we can further rewrite the thesis as follows:

$$\{t\} < (path_{(_,\$var,=)}(return_{of}(\sigma_{Pred_{Glob}(\$X,\$Z)}(\mathrm{e}_\sigma))) >$$

$$\equiv$$

$$\pi_{Att(\{\vec{t}\});\ \$var}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(\{t\} \ltimes_{Pred_{Glob}(\$X,\$Z)} (path_{f'}(return_{of'}(e)_\sigma)))))$$

Let's work on the right member of the thesis. Since the left operand of the join ($\{t\}$) is composed by only one tuple, and since *unnestvar* and *rename* do not work on t attributes, we can rewrite the right member as follows:

$$\pi_{Att(\{\vec{t}\});\ \$var}(\{t\} \ltimes_{Pred_{Glob}(t.\$X,t.\$Z)} unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}((path_{f'}(return_{of'}(e)_\sigma)))))$$

By relying on the Lemma B.1.2, we can futher rewrite this expression as follows:

$$\pi_{Att(\{\vec{t}\});\ \$var}(\{t\} \ltimes_{Pred_{Glob}(t.\$X,\$Z)} unnestvar_{\$var}((path_{f''}(return_{of'}(e)_\sigma))))$$

where $f'' = f'(\$var/\$_res)$.

We can now observe that the so-transformed predicate $Pred_{Glob}(t.\$X, \$Z)$ only works on inner query attributes, hence it can be detached from the join operation and safely pushed into the nested query. This transformation, together with the use of Lemma B.1.3, leads to the following expression.

$$\pi_{Att(\{\vec{t}\});\ \$var}(unnestvar_{\$var}(\{t\} \times path_{f'''}(return_{_nested[result[of]]}(\sigma_{Pred_{Glob}(t.\$X,t.\$Z)}(e_\sigma)))))$$

Since $\pi_{A_1;\,A_2}(e_1 \times e_2) \equiv \pi_{A_1}(e_1) \times \pi_{A_2}(e_2)$ if $A_1 \subseteq Att(e_1) \wedge A_2 \subseteq Att(e_2) \wedge Att(e_1) \cap Att(e_2) = \emptyset$, we can rewrite this expression as follows.

$$\pi_{\overrightarrow{Att(\{t\})}}(\{t\}) \times$$
$$\pi_{\overrightarrow{\$var}}(unnestvar_{\$var}(\{t\}path_{f'''}(return_{_nested[result[of]]}(\sigma_{Pred_{Glob}(t.\$X,t.\$Z)}(e_\sigma)))))$$

By observing that $\pi_{\overrightarrow{Att(\{t\})}}(\{t\}) \equiv \{t\}$, by pushing down *unnestvar*, and by applying Lemma B.1.4, we can rewrite the previous expression as follows.

$$\{t\} \times \pi_{\overrightarrow{\$var}}(path_{(_,\$_n,in)_nested[(/,\$var,=)_[\emptyset]]}(return_{_nested[of]}(\sigma_{Pred_{Glob}(t.\$X,\$Z)}(e_\sigma))))$$

Finally, we can further simplify this expression as shown below.

$$\{t\} \times \pi_{\overrightarrow{\$var}}(path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_{Pred_{Glob}(t.\$X,\$Z)}(e_\sigma))))$$

To conclude the proof, we can replace $\$X$ with $t.\$X$ in the left member of the equation, hence obtaining the following expression:

$$\{t\} \times \pi_{\overrightarrow{\$var}}(path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_{Pred_{Glob}(t.\$X,\$Z)}(e_\sigma))))$$

Proposition B.1.5 *Range dependency*

$$e_1 < (path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_P(path_{f_1,\ldots,f_k}(\$x_1,\ldots,\$x_k))))) >$$

$$\equiv$$

$$\pi_{\overrightarrow{Att(e_1);\,\$var}}(unnestvar_{\$var}(rename_{\$_res\rightarrow\$var}(e_1 \bowtie_{\$X=\$X'}$$
$$(path_{f'}(return_{of'}(\sigma_P(path_{f''}(e_1))))))))$$

where

- $f'' \equiv (_,\$_tuple,in)tuple[(/,_,in)x_1[(/,\$x_1,=)_[f_1],\ldots,(/,_,in)x_k[(/,\$x_k,=)_[f_k]]]]$
- $of' \equiv _nested[_result[of],_env[x_1[\$x_1],\ldots,x_k[\$x_k]]]$
- $f' \equiv (_,\$_n,in)_nested[(/,\$_res,in)result[\emptyset],\ (/,_,in)_env[(/,_,in)x_1[(/,\$x_1,=)_[\emptyset]],\ldots,(/,_,in)x_k[(/,\$x_k,=)_[\emptyset]]]]$

if $FV(of) \cap Att(e_1) = \emptyset$, $FV(f_1,\ldots,f_k) \cap Att(e_1) = \emptyset$, $FV(P) \cap Att(e_1) = \emptyset$, $FV(db) = \$X \subseteq Att(e_1)$, $x_1,\ldots x_k \notin symbols(of)$

Proof: We rely on the left-linearity property of $< \cdot >$, so it suffices to prove that:

$$\{t\} < (path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_P(path_{f_1,\ldots,f_k}(\$x_1,\ldots,\$x_k))))) >$$

$$\equiv$$

$$\pi_{\overrightarrow{Att(\{t\})};\ \$var}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(\{t\} \bowtie_{\$X=\$X'}$$
$$(path_{f'}(return_{of'}(\sigma_P(path_{f''}(\{t\}))))))))$$

Since t is a constant expression, we can rewrite the core of the right member of the equivalence as follows:

$$path_{f'}(return_{of'}(\sigma_P(path_{f''}(\{t\}))))$$

$$\equiv$$

$$path_{f'}(return_{of'}(\sigma_P(rename_{\$X \rightarrow \$X'}(path_{f_1,\ldots,f_k}(t.\$x_1,\ldots,t.\$x_k)))))$$

As in the previous proof, we can distribute the outer projection on the right member of the equivalence, push down the $unnestvar - rename$ pair, and split the join into a cross product and a selection, hence obtaining the following expression:

$$\pi_{\overrightarrow{Att(\{t\})}}(\{t\}) \times$$
$$\pi_{\overrightarrow{\$var}}(unnestvar_{\$var}(rename_{\$_res \rightarrow \$var}(path_{f'}(return_{of'}(\sigma_{t.\$X=\$X'}(\sigma_P(rename_{\$X \rightarrow \$X'}(path_{f_1,\ldots,f_k}(t.\$x_1,$$

Since there is only one tuple, the predicate $t.\$X = \$X'$ is trivially true, and the corresponding selection operation can be safely omitted. Moreover, $\$X'$ variables disappear before executing the cross product, so there is no need for the *rename* operation. Hence, we can rewrite the previous expression as follows:

$$\{t\} \times$$
$$\pi_{\overrightarrow{\$var}}(path_{(_,\$_n,in)_nested[(/,\$var,=)_[\emptyset]]}(return_{_nested[of]}(\sigma_P(path_{f_1,\ldots,f_k}(t.\$x_1,\ldots,t.\$x_k)))))$$

As in the previous proof, this expression can be further rewritten as follows:

$$\{t\} \times path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_P(path_{f_1,\ldots,f_k}(t.\$x_1,\ldots,t.\$x_k))))$$

The left member of the equivalence can be rewritten as follows:

$$\{t\} \times path_{(_,\$var,=)_[\emptyset]}(return_{of}(\sigma_P(path_{f_1,\ldots,f_k}(t.\$x_1,\ldots,t.\$x_k))))$$

which concludes the proof.

Proposition B.1.6 *Projection dependency*

$$e_1 < (path_{(_,\$_var,=)_[\emptyset]}(return_{of(\$X)}(\sigma_P(path_{f_1}(db))))) >$$

$$\equiv$$

$$\pi_{\overrightarrow{Att(e_1)};\ \$_var}(unnestvar_{\$_var}(rename_{\$_res \rightarrow \$_var}(e_1 \bowtie_{\$X=\$X}$$
$$path_{f'}(return_{of'}(\pi(\$X)(e_1) \bowtie_{true} (\sigma_P(path_{f_1}(db)))))))$$

where

- $of' \equiv _nested[_result[of], _env[x_1[\$x_1], \ldots, x_k[\$x_k]]]$
- $f' \equiv (_, \$_n, in)_nested[(/, \$_res, in)result[\emptyset], (/, _, in)_env[(/, _, in)x_1[(/, \$x_1, =)_[\emptyset]], \ldots, (/, _, in)x_k[(/, \$x_k, =)_[\emptyset]]]]$

if $FV(of) \cap Att(e_1) \neq \emptyset$, $FV(f_1) \cap Att(e_1) = \emptyset$, $FV(P) \cap Att(e_1) = \emptyset$, $\$X \subseteq Att(e_1)$, $x_1, \ldots x_k \notin symbols(of)$

Proof: The proof is based on the left-linearity property of $< \cdot >$.

Let $e_1 \equiv \{t\}$, and let $e_\sigma \equiv \sigma_P(path_{f_1}(db))$. To prove the thesis it suffices to show that the following equivalence holds:

$$\{t\} < (path_{(_,\$_var,=)_[\emptyset]}(return_{of(\$X)}(e_\sigma))) >$$

$$\equiv$$

$$\pi_{\overrightarrow{Att(\{t\})};\ \$_var}(unnestvar_{\$_var}(rename_{\$_res \to \$_var}(\{t\} \Join_{\$X=\$X} path_{f'}(return_{of'}(\pi(\$X)(\{t\}) \Join_{true} (e_\sigma)))))$$

The right member of the equivalence can be rewritten as follows:

$$\pi_{\overrightarrow{Att(\{t\})};\ \$_var}(unnestvar_{\$_var}(rename_{\$_res \to \$_var}(\{t\} \times \sigma_{\$X'=t.\$X}(path_{f'}(return_{of'}(rename_{\$X \to \$X'}(t.\$X \times e_\sigma)))))))$$

Now, we can push the selection $\sigma_{\$X'=t.\$X}$ down into the nested query, and remove the inner *rename* as well as all operations concerning $\$X$, hence obtaining the following expressions:

$$\pi_{\overrightarrow{Att(\{t\})};\ \$_var}(unnestvar_{\$_var}(rename_{\$_res \to \$_var}(\{t\} \times path_{f'}(return_{of'}(\sigma_{\$X=t.\$X}(t.\$X \times e_\sigma))))))$$

$$\pi_{\overrightarrow{Att(\{t\})};\ \$_var}(unnestvar_{\$_var}(rename_{\$_res \to \$_var}(\{t\} \times path_{f''}(return_{_nested[_result[of]]}(\sigma_{\$X=t.\$X}(t.\$X \times e_\sigma))))))$$

where $f'' = (_, \$_n, in)_nested[(/, \$_res, in)result[\emptyset]]$.

By observing that *unnestvar* and *rename* only works on inner query variables, we can further rewrite the previous expression as follows:

$$\pi_{\overrightarrow{Att(\{t\})};\ \$_var}(\{t\} \times unnestvar_{\$_var}(rename_{\$_res \to \$_var}(path_{f''}(return_{_nested[_result[of]]}(\sigma_{\$X=t.\$X}(t.\$X \times e_\sigma))))))$$

By applying the simplification rules for *unnestvar* and *rename*, this expression can be rewritten as follows:

$$\{t\} \times path_{(_,\$_var,=)_[\emptyset]}(return_{of}(\sigma_{\$X=t.\$X}(t.\$X \times e_{\sigma})))$$

By observing that (1) of contains references to t, and (2) $t.\$X$ and $\sigma_{\$X=t.\$X}$ are used for capturing those references, we can further rewrite the previous expression as:

$$\{t\} \times path_{(_,\$_var,=)_[\emptyset]}(return_{of}(e_{\sigma}))$$

, which concludes the proof.

Appendix C

Benchmarks

C.1 XMark Queries

C.1.1 Path queries

Q11:

```
for $n in input()/site/regions/europe/item/name
return $n
```

Q12:

```
for $n in input()/regions/*/item/name
return $n
```

Q13:

```
for $n in input()//name
return $n
```

Q14:

```
for $n in input()//*/name
return $n
```

C.1.2 Twig queries

Q21:

```
for $i in input()/site/regions/europe/item,
    $d in $i/description,
    $n in $i/name
return < item > { $n, $d } < /item >
```

Q22

```
for $i in input()//item,
    $d in $i/description,
    $n in $i/name,
    $m in $i//mail
return < item > { $n, $d,$m} < /item >
```

C.1.3 Twig queries with groups

Q31:

```
for $i in input()//item,
    $n in $i/name
let $m_list := $i/mail
return < item > $n, $m_list < /item >
```

Q32:

```
for $auc in input()//open_auction,
    $init in $auc/initial
let $bids := $auc/bidder
for $inc in $b/increase
return < open_auctions > $auc, $inc < /open_auctions >
```

C.1.4 Predicate queries

Q41:

```
for $i in input()//item,
    $n in $i/name,
    $d in $i/description
where $i/quantity > 1
return < item > $n, $d < /item >
```

Q42:

```
for $l in input()//location
where data($l) = "United States"
return $l
```

C.1.5 Nested queries

Q51:

```
for $auc in input()//closed_auction,
    $i in input()//item,
    $b in for $p in input()//person
          where $p/@id = $auc/buyer/@person
          return < buyer > {$p/name, $p/homepage } < /buyer >
where $i/@id = $auc/itemref/@item
return < closed_auction > {$i/name, $b } < /closed_auction >
```

Q52:

```
for $auc in input()//closed_auction,
    $i in input()//item
let $b := for $p in input()//person
          where $p/@id = $auc/buyer/@person
          return < buyer > {$p/name, $p/homepage } < /buyer >
```

```
let $s := for $p in input()//person
          where $p/@id = $auc/seller/@person
          return < seller > {$p/name, $p/homepage} < /seller >
where $i/@id = $auc/itemref/@item AND $auc/price > 10
return < cheap_auction > { $i/name,
                           $auc/price,
                           $b,
                           $s }
       < /cheap_auction >
```

C.1.6 Negative queries

Q61:

```
for $p in input()//person,
    $i in $p//item
where $i/name = "scarce brook"
return < result > { $p } < /result >
```

Q62:

```
for $p in input()//person
where $p/country = "Mars"
return < result > { $p } < /result >
```

C.2 DBLP Queries

C.2.1 Path queries

Q11:

```
for $t in input()/dblp/article/title
return $t
```

Q12:

```
for $t in input()/dblp/*/title
return $t
```

Q13:

```
for $a in input()//author
return $a
```

Q14:

```
for $a in input()//*/author
return $a
```

C.2.2 Twig queries

Q21:

```
for $a in input()/dblp/article,
    $au in $a/author,
    $t in $a/title
return < article > { $au, $t } < /article >
```

Q22

```
for $i in input()//*,
    $au in $a/author,
    $t in $a/title
return < article > { $au, $t } < /article >
```

C.2.3 Twig queries with groups

Q31:

```
for $a in input()/dblp/article,
    $t in $a/title
let $a_list := $a/author
return < article > { $t, $a_list } < /article >
```

Q32:

```
for $dblp in input()/dblp
let $b_list := $dblp/book,
for $a in $b_list/author,
    $t in $b_list/title
return < authortitle > { $a, $t } < /authortitle >
```

C.2.4 Predicate queries

Q41:

```
for $a in input()//article,
    $t in $a/title,
    $y in $i/year
where $y > 2001
return < recentarticle > $t, $y < /recentarticle >
```

Q42:

```
for $a in input()//article,
    $t in $a/title,
    $au in $a/author
where data($au) = "Peter Buneman"
return < bunemanarticle > $t, $au < /bunemanarticle >
```

C.2.5 Nested queries

Q51:

```
for $a in input()//article,
```

```
    $au in $a/author
let $t_list := for $p in input()//article
                  where $p/author = $au
                  return $p/title
return < authortitle > {$au, $t_list } < /authortitle >
```

Q52:

```
for $a in input()//author
let $at_list := for $p in input()//article
                   where $p/author = $a
                   return $p/title < /buyer >
let $bt_list := for $p in input()//book
                   where $p/author = $a
                   return $p/title
return < authortitle > {$a, $at_list, $bt_list } < /authortitle >
```

C.2.6 Negative queries

Q61:

```
for $au in input()//author,
    $a in $au//article
return < result > { $au, $a } < /result >
```

Q62:

```
for $a in input()//article,
    $t in $a/title,
    $y in $i/year
where $y > 2030
return < futurearticle > $t, $y < /futurearticle >
```