A General Framework for Estimating XML Query Cardinality

Carlo Sartiani

Dipartimento di Informatica - Università di Pisa
Via Buonarroti 2, Pisa, Italy
sartiani@di.unipi.it,
phone: +39 05022133140 - fax: +39 0502212726

Abstract. 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, and it may be helpful in embedded query execution.
Existing estimation models for XML queries focus on particular aspects of XML querying, such as the estimation of path and twig expression cardinality, and they do not deal with the problem of predicting the cardinality of general XQuery queries. This paper presents a framework for estimating XML query cardinality. The framework provides facilities for estimating result size of FLWR queries, hence allowing the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. The framework can also be used for extending existing models to a wider class of XML queries.

1 Introduction

The last few years have seen the rapid emerging of the eXtensible Markup Language (XML). Given its ability to represent nearly any kind of information, XML imposes itself as the standard format for representing *semistructured* data, and it is also increasingly used as data exchange format.

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 [5], and it may be helpful in embedded query execution.

As for many other common database tasks, the peculiar nature of XML makes result size estimation more difficult than in the relational context: in particular, the (very) non-uniform distribution of tags and data, together with the dependencies imposed by the tree structure of XML data, make not so reasonable the usual hypothesis used in relational size estimators.

1.1 Our Contribution

This paper describes a framework for estimating the cardinality of FLWR XQuery queries. Existing estimation models for XML queries focus on particular issues of result size estimation: in particular, some models are limited to path expression estimation only [2], while others also deal with twig queries [3] [8], but still ignoring critical issues such as iterators, binders, nested queries, etc. The proposed framework, instead, offers algorithms for estimating the cardinality of full FLWR queries, with the only exception of universally quantified predicates.

Furthermore, the algorithms in the framework estimate not only the *raw* cardinality of query results, but also their distribution, while existing models, with the only notable exception of the StatiX model [5], return only raw cardinalities (e.g., number of tuples in the result).

The framework allows the model designer to concentrate her efforts on the development of adequate and accurate, while concise, statistic summaries for XML data. The framework forms the basis of the estimation model of Xtasy [6], and it can also be used for extending existing models to a wider class of XML queries.

1.2 Paper Outline

The paper is organized as follows. Section 2 describes the main issues in XML query result size estimation. Section 3, then, introduces the framework and its estimation policy, and provides an informal description of the framework algorithms. Next, Section 4 illustrates some experimental results about the estimation model of Xtasy, which is an instance of the framework being described in this paper. Section 5, then, presents the state of the art in XML query cardinality estimation. In Section 6, finally, we draw our conclusions.

2 Issues in Result Size Estimation

Referring to the FLWR fragment of XQuery, the most problematic aspects in result size prediction concern the estimation of path and twig cardinality, the estimation of predicate selectivity, as well as the estimation of group cardinality (let binder of XQuery). While path and twig estimation is a peculiar issue of XML and semistructured query languages, predicate and group 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.

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 Fig. 1.

```
<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>
```

Fig. 1. A sample XML document

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 [3], [8], and [2]), 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,

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

[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.

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*.

3 The Framework

3.1 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 3.3. Inside ETLSs, data distribution is described by means of sequences of *match occurrences*, which are themselves illustrated in Section 3.3.

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.

3.2 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 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 [5]), or the location of the node in the originating database (*extensional* region, e.g., the grid cell in the related *position* histogram in TIMBER [8]). 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 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.

Definition 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 1. Consider the sample document of Fig. 1, obeying the schema of Fig. 2. Assume that we build an intensional partitioning scheme based on such schema. The corresponding tagged region graph $\mathcal{G}$, then, is shown in Fig. 3; statistical information related to these graph is shown in Tables 1 an 2.[2]

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 ($N_{afo}((l, r), (l', r'))$).

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

```
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]
```

Fig. 2. A schema for the sample document

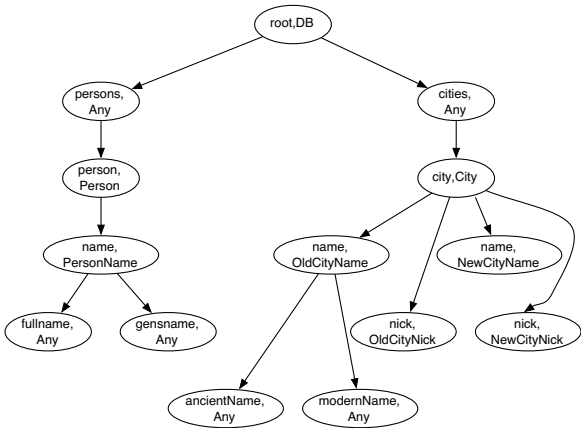

Fig. 3. Tagged region graph with /-edges

Table 1. N_{el} table (Xtasy model) for the tagged region graph of Fig. 3.

(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 2. N_{afo} table (Xtasy model) for the tagged region graph of Fig.3.

(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

3.3 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 4. *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, which are called ECLSs, and can be further organized in more complex structures called ETLSs.

Definition 5. *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 6. *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 2. Consider the following query clause:

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

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

The estimation function first estimates the result 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:

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

3.4 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 [1]; 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., [5]).

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

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

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.

3.5 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 4. 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 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).

3.6 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 2 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:[4]

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

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,
```

[4] $grouped(etls)$ 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.

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 $\mathcal{S}$, 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 5. 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 Fig. 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.

$$\{\$n\_list : \{\{\ (name, OldCityName, 1)\},\\ \{(name, OldCityName, 1)\}\}\\ \{(name, NewCityName, 1)\}\} .$$

3.7 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 6. 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 7. *Given a unary predicate P, the selectivity factor of P $psf[P]$ is a function*

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

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

This definition naturally leads to histogram-based selectivity factors, 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 7. 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 Fig. 1, the selectivity factor for the predicate $P \equiv data(\$n) = "Monticello"$ could be the following:

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

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 8. Consider again the query fragment of Example 6, and the selectivity factor of Example 7; assume that the for clause estimation returns the following ETLS:

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

By applying the selectivity factor of Example 7, the framework generates the following ETLS:

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

(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 some experimental results about the statistical model of Xtasy.

Example 9. 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.

4 Experimental Results

This Section presents some experimental results concerning the estimation model of Xtasy, which is an instance of the framework being described in this paper. The statistical model of Xtasy is based on an extensional partitioning of XML data: a region $\mathcal{R} = (h, p)$ is defined as the set of nodes at the level h in the tree, and whose position in the document order is comprised between p and $p + \delta_p$, where δ_p is a tunable parameter.

The experimental results were obtained by running a set of benchmark queries over the XMark Standard dataset [7], and by comparing predictions with the actual result size.

4.1 Benchmark Queries

The experiments performed to validate the prediction model are based on a set of benchmark queries that, unlike usual XML database benchmarks (see [7] for instance), were designed with the purpose of testing the accuracy of size predictions.

The benchmark contains six sets of queries, each of them related to different size prediction issues: **path queries**, which evaluate linear path expressions, both fully specified and with wildcards and closure operators; **twig queries**, which evaluate twig expressions, both fully specified and with closure operators, but without the grouping binder; **twig queries with groups**, which evaluate

complex twig queries with the grouping binder (`let ... :=`); **queries with predicates**, which apply unary predicates to the result of twig expressions; **nested queries**; and, **negative queries**, which are used for estimating the accuracy of size predictions on empty queries.

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 [3], is motivated by the will to test the accuracy of the model even in the worst conditions.

4.2 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.[5]

$$RE(Q) = \frac{ES(Q) - AS(Q)}{AS(Q)} \ . \tag{1}$$

$$AE(Q) = ES(Q) - AS(Q) \ . \tag{2}$$

4.3 Experimental results

We perform the benchmark queries on two distinct statistical configurations: we use $\delta_p = 100000$ and $\delta_p = 500000$. For predicate queries, we estimate the accuracy of the size model with and without proper histograms.

Graphs for the six classes of benchmark queries are shown in Fig. 4: in these graphs, relative errors are shown in the form of percentage relative errors.

Path queries 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.

Twig queries 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%).

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.

[5] $ES(Q)$ denotes the estimated size of Q, while $AS(Q)$ denotes its actual size.

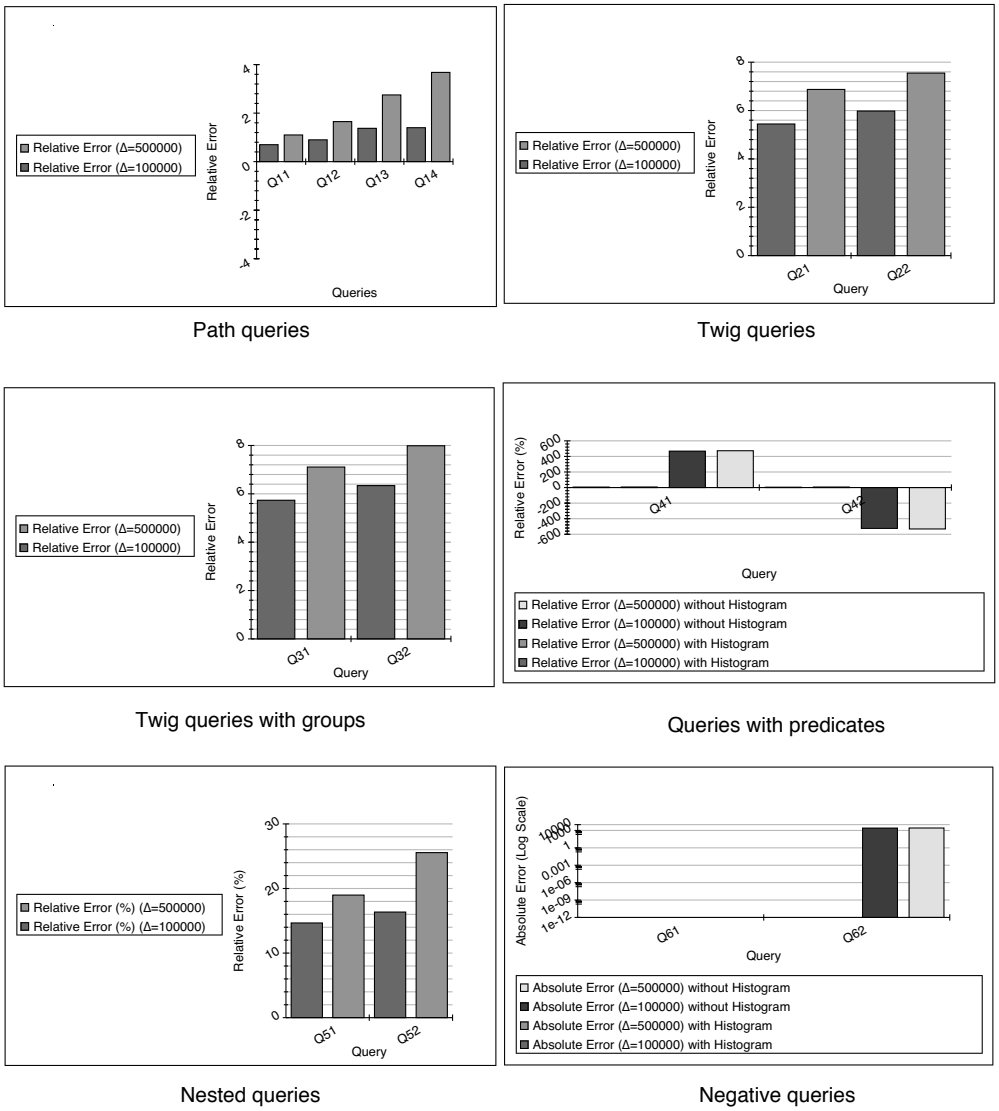

Fig. 4. Accuracy tests for benchmark queries

Queries with predicates 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 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 This class consists of a *structurally* empty query, and of a query with a false predicate. In the case of the structurally empty query, 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.

5 Related Works

Correlated Subpath Trees In [3] 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 Σ^* (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 [4] 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 subpath 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 forms 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 [2] ; 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.

TIMBER Result Size Model In [8] authors propose two models for estimating the number of matches of a twig query Q over a XML tree T. The proposed models rely on *position histograms* for predicates in a set $\mathcal{P}$. Position histograms can be used for estimating the raw cardinality of simple ancestor/descendant queries. The first model exploits position histograms only, while the second one also uses *coverage histograms*, 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.

The model based on coverage histograms is much more accurate than the model based only on position histograms; 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. Finally, the model only deals with twig matching, hence ignoring critical issues such as iterators, binders, nested queries, etc.

Niagara's Models In [2] authors present the path expression selectivity estimation models employed in Niagara. 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 first model is based on a structure called *path tree*. 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.

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. As for path trees, the size of a Markov table may exceed the total amount of available main memory, hence summarization techniques are required.

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.

StatiX Statistic Model In [5] authors describe a methodology for collecting statistics about XML documents; the proposed approach is applied in LegoDB for providing statistics about XML-to-relational storage policies, and, to a less extent, in the Galax system for predicting XML query result size.

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).

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.

6 Conclusions

This paper has described a framework for estimating the cardinality of XML queries. The proposed framework offers tools and algorithms for predicting not only the raw size of query results, but also the distribution of data inside them, hence making the prediction of the size of subsequent operations more accurate. The facilities offered by the framework range from group cardinality estimation

to twig branch correlation, and selectivity factor application; by relying on these facilities, the model designer can focus on the definition of accurate and concise statistic summaries.

7 Acknowledgments

The author would like to thank Dan Suciu for his help during the revision of the paper.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman, 1999.
2. 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.
3. 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.
4. 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.
5. 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.
6. Carlo Sartiani. Efficient Management of Semistructured XML Data, 2003. Manuscript draft.
7. 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.
8. 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.