A Framework for Estimating XML Query Cardinality

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Via Buonarroti 2, Pisa, Italy
sartiani@di.unipi.it

ABSTRACT

Tools for querying and processing XML data are increasingly available. In this context, the estimation of the cardinality of queries on XML data is becoming more and more important. The information provided by a query result estimator can be used as input to the query optimizer, and as an early feedback to user queries. 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 emerging and the wide diffusion of XML. As a consequence, tools for storing, querying, and manipulating XML data are nowadays increasingly available. In the context of XML data management system, the estimation of the cardinality of queries on XML data 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 [4].

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, makes not so reasonable the usual hypothesis used in relational size estimators.

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. Moreover, these models usually estimate the *raw* cardinality of queries (e.g., the number of nodes returned by a path, or the number of tuples generated by tuple-based execution engines), without providing any information about the distribution or the nature of the expected result, which is necessary for correctly predicting the size of further operations.

Our Contribution. This paper presents a framework for estimating the cardinality of FLWR XQuery queries. The framework provides facilities for estimating the raw size as well as the data distribution of query result. In particular, it offers algorithms for estimating the size of sets built by the let clause of XQuery, for correlating the estimation of twig branches, and for applying predicate selectivity factors according to the distribution of data.

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

2. ISSUES IN RESULT SIZE ESTIMATION

Result size estimation for XML queries requires the system to predict the cardinality of any query in the language. Referring to a fragment of XQuery [3] without recursive functions, the most problematic aspects 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,

International Workshop on the Web and Databases (WebDB).
June 12–13, 2003, San Diego, California.

```
<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 1: A sample XML document

its relevance in operations like path and predicate evaluation. For example, consider the XML document shown in Figure 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, 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. Unlike SQL, 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 a) to estimate the number of distinct groups created, b) to correlate each group to the variables on which it depends ($\$n\_list$ depends on $\$c$), and c) to 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

Dealing with the estimation problems described in the previous Section requires the prediction model to estimate not only the raw cardinality of results, but also their distribution. The following Sections will explain the estimation approach employed in the framework.

3.1 Match Occurrences

The facilities offered by the framework rely on the notion of *match occurrence*. A match occurrence o is a triple (l, r, m), where l is a node label, r is the *region* where o occurs, 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. 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.,

[1] We are aware of proposals for extending XQuery with explicit `groupby` clauses. Due to time and space constraints, we cannot discuss such proposals in this paper.

```
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 2: A schema for the sample document

the node type and its bucket in the corresponding *structural* histogram in StatiX [4]), or the location of the node in the originating database (*extensional* region, e.g., the grid cell in the related *position* histogram in TIMBER [6]). Regions, hence, can express both intensional and extensional concepts, and are the main tool for describing the distribution of data. Regions are also the basic tools for correlating match occurrences coming from different branches of a twig; as shown in Section 3.4, correlation by means of regions is one of the most interesting and useful facilities offered by the framework. The following example illustrates the notion of match occurrence.

Example 3.1 Consider the XML document shown in Figure 1, and assume that the path expression `input()//name` is being evaluated on it. Assuming an extensional partitioning of the tree based on height, the result distribution is the following:

$$\{(name, 4, 3)\}$$

This sequence denotes the occurrence of three `name` elements at level 4.

Assuming an intensional partitioning of data based on the schema of Figure 3.1, the result would be the following:

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

■

3.2 ECLSs and ETLSs

Given the notion of match occurrence, the main idea behind the framework is to use estimation functions that manipulate sequences of match occurrences called Extended Context Label Sequences. An ECLS, hence, is a sequence where match occurrences for a given path are accumulated. The following example briefly illustrates this point.

Example 3.2 Consider the query fragment of the previous example, and the intensional partitioning hypothesis. Then, the ECLS generated for the path expression would be the following.

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

■

ECLSs are bound to variable symbols to form a data structure called ETLS (Extended Tuple Label Sequence), which collects 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 a sample ETLS.

Example 3.3 Consider the following query clause:

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

The estimation of the cardinality for this clause (type regions) would generate the following ETLS.

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

■

3.3 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., [4]).

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 $\| \cdot \|$.

This notion of cardinality, while widely used, provide no information about the shape and structure of XML data returned by a query, hence models relying on this raw notion only are exposed to (many) potential estimation errors.

3.4 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.[2]

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

[2]The framework makes the implicit assumption that twig cardinality is not directly stored in statistics, but, instead, computed from path cardinality. This assumption is common in statistical models for XML queries.

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 3.4 Consider the following query fragment:

```
for $c in input()//city,
    $y 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 (intentional partitioning):

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

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 estimates 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).

3.5 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:

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

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,
```

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 step: the algorithm, first, collects all match occurrences of the let path expression in 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 3.5 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 Figure 1, the framework generates the following ETLS:

$$\{\$c :< \{(city, City, 2)\} >\}$$

The list of match occurrences collected for the let path expression (intensional partitioning) is the following:

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

The framework creates 2 groups, and distributes match occurrences as shown below.

$$\begin{array}{ll} \{\$n\_list :< \{ & (name, OldCityName, 1)\}, \\ & \{(name, NewCityName, 1)\} >\} \end{array}$$

■

The group creation algorithm has $O(n^2)$ worst case time complexity, where n is the number of match occurrences. This moderate time complexity is the price payed for obtaining a very accurate size estimation.

3.6 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 tag 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 3.6 Consider the following query fragment:

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

The predicate `data($n) = "Monticello"` applies to ancientname and currentname elements, and its selectivity factor depends on the tag of the subject node, since values can have different distributions in ancientname and currentname 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 3.7 *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 3.8 Consider the query fragment of the previous example, where variable $\$n$ is bound to ancientname as well as currentname elements. Assuming that statistics are gathered from a bigger document than that of Figure 1, the selectivity factor for the predicate $P \equiv data(\$n) = "Monticello"$ could be the following:

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

■

The definition of predicate selectivity factors extends from unary predicates to binary and n-ary predicates. For binary and n-ary predicates, selectivity factors are built on a single variable, whose choice is left to the specific model.

The second main issue about predicate selectivity estimation concerns the way these factors are applied to ETLSs in the framework. For instance, given the well-known 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 3.9 Consider again the query fragment of Example 3.6, and the selectivity factor of Example 3.8; assume that the for clause estimation returns the following ETLS:

$$\begin{array}{ll} \{\$c :< \{ & (city, r_1, 2), (city, r_3, 1), (city, r_4, 1)\} >, \\ \$n :< \{ & (ancientname, r_5, 2), (currentname, r_6, 1), \\ & (currentname, r_9, 1)\} >\} \end{array}$$

By applying the selectivity factor of Example 3.8, the framework generates the following ETLS:

$$\begin{array}{ll} \{\$c :< \{ & (city, r_1, .14), (city, r_3, .75), (city, r_4, .67)\} >, \\ \$n :< \{ & (ancientname, r_5, .14), (currentname, r_6, .75), \\ & (currentname, r_9, .67)\} >\} \end{array}$$

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

4. COMPLEXITY ISSUES

In this Section we will briefly expose some complexity results about the most important algorithms of the framework, namely the correlation function, the selectivity factor propagation algorithm, the group cardinality estimation algorithm, and the cardinality computing function; due to lack of space these algorithms are reported in [5].

The correlation function determines whether two match occurrences o_1 and o_2 are correlated by a common ancestor in the sequence of ECLSs 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) should be modified to be used in this context: in particular, by endowing each pair (l, r), during statistics collection, with its reachability sets in $\mathcal{G}$ (both ancestors and descendants), this problem can be solved in time $O(n)$, where n is the number of match occurrences bound to the variable $\$root$. These sets are used for lowering time complexity of other algorithms too.

In a similar way, the selectivity factor propagation problem can be reduced to the exploration of the (l, r) graph; hence, by exploiting the reachability set of (l, r) pairs and supplementary data structures, the propagation can be performed in $O(m + n)$ time, where n is the number of match occurrences and m is the number of edges in the graph. Due to the wide space requirements of the auxiliary data structures, in [5] we describe a more space-conscious algorithm with time complexity $O(n^2)$.

The group cardinality estimation algorithm is heavily based on the correlation function, since match occurrence distribution is performed according to the correlation relation. The algorithm just scans, for any match occurrence in the let path expression, the list of the root match occurrences, in order to find the target groups; as a consequence, the algorithm has $O(n^2)$ worst case time complexity, where n is the number of root match occurrences.

The cardinality computing function identifies trees of matching occurrences in the ETLS, and then computes their cardinality. This is performed by computing the cardinality of

each twig in the query Q, and by combining them in the cardinality of the whole query; the calculation relies again on the correlation function, hence leading to a $O(n^3)$ worst case time complexity.

5. RELATED WORKS

TIMBER result size model. In [6] authors propose two models for estimating the number of matches of twig queries. 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 statistics model. In [4] 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 Dario Colazzo for his support during and after the writing of this paper.

8. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman, 1999.

[2] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. 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] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, April 2002. W3C Working Draft.

[4] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. 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.

[5] C. Sartiani. A Framework for Estimating XML Query Cardinality - Moderately Extended Version, 2003. Available at `http://www.di.unipi.it/ sartiani/papers/wedb03ext.pdf`.

[6] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for xml queries. In C. S. Jensen, K. G. Jeffery, J. Pokorný, S. Saltenis, E. Bertino, K. Böhm, and M. 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.