StatiX: Making XML Count

Juliana Freire[1] Jayant R. Haritsa[2] Maya Ramanath[2] Prasan Roy[1] Jérôme Siméon[1]
[1]Bell Labs [2]Indian Institute of Science
{juliana,prasan,simeon}@research.bell-labs.com {haritsa,maya}@dsl.serc.iisc.ernet.in

ABSTRACT

The availability of summary data for XML documents has many applications, from providing users with quick feedback about their queries, to cost-based storage design and query optimization. *StatiX* is a novel XML Schema-aware statistics framework that exploits the structure derived by regular expressions (which define elements in an XML Schema) to pinpoint places in the schema that are likely sources of *structural skew*. As we discuss below, this information can be used to build concise, yet accurate, statistical summaries for XML data. StatiX leverages standard XML technology for gathering statistics, notably XML Schema validators, and it uses histograms to summarize both the structure and values in an XML document. In this paper we describe the StatiX system. We develop algorithms that decompose schemas to obtain statistics at different granularities and discuss how statistics can be gathered as documents are validated. We also present an experimental evaluation which demonstrates the accuracy and scalability of our approach and show an application of these statistics to cost-based XML storage design.

1. INTRODUCTION

XML has become an important medium for data representation and the volume of XML-based data processing is increasing at a rapid pace. Accordingly, there is a growing need for designing systems that efficiently store and query XML data. A critical component in such systems is the *result estimator*, which estimates the *cardinalities* of the results of user queries. Its importance arises from the fact that estimated cardinalities serve as inputs in many aspects of XML data management systems: from cost-based storage design and query optimization, to providing users with early feedback about the expected outcome of their queries and the associated computational effort.

In this paper, we consider the problem of developing efficient and accurate XML query result estimators. Specifically, we present a system called *StatiX* that addresses this issue in the context of documents described with XML Schema [23], and user queries presented in XQuery [5].

Design Issues. A large body of literature is available for result estimators in traditional database systems (*e.g.*, [21]). These estimators are typically based on collecting summary statistics about the data elements, such as *minimum value*, *maximum value*, *number of unique values*, etc., and computing estimates based on these statistics. In the newly-developing XML world, however, the design of result estimators becomes more complex due to the following reasons:

- The flexibility allowed by the use of regular expressions to define elements in the XML data model typically results in data that has *highly skewed structure*—therefore, assumptions of uniform distribution that are sometimes resorted to in traditional DBMS may result in unacceptably high errors when applied to XML data.
- XML documents are often *broad and tall*—that is, the equivalent document tree features both large fanouts and deep nesting in the element structures. (A good example of such a structure is XMark [26], a recently announced XML benchmark). Queries on such documents typically feature *path expressions* that cover sub-trees of the document. Evaluating these queries requires a large number of "joins", much larger than those handled in typical database applications. This increases the scope for inaccurate estimates since it is known that errors propagate rapidly through multi-way join queries [12].

Proposals for XML result estimators have appeared in the recent literature [14, 6, 1, 17]. Because these approaches provide statistical information in the context of *schemaless* semistructured data and need to process and summarize the whole document structure, they can be expensive both in terms of space and memory consumption. Secondly, they support only limited classes of queries. For example, [1] is restricted to simple (non-branching) path expressions in the document tree and cannot handle *equality predicates* or queries that involve reference chasing. Similarly, [6] cannot efficiently handle *value range selections*, *e.g.*, a query such as $1991 \leq YaYe \leq 2000$. Finally, the proposals involve either usage of specialized data structures, or expensive processes for system initialization, or costly maintenance for document updates. (See Section 6 for a more detailed discussion on related work.)

The StatiX System. Our new system, StatiX, attempts to address the above-mentioned shortcomings of the state-of-the-art in XML result estimators. In particular, its design is founded on the following core principles:

- **XML Schema-based statistics collection**: In a growing number of data-oriented applications, the use of schemas is becoming commmonplace. (See [15] for descriptions of several applications and standardized XML Schemas.) We capitalize

on this industry trend by employing XML Schema types as the basis for statistics gathering, and this enables StatiX to produce concise, yet accurate, summaries of XML data.

- **Histogram-based statistics maintenance**: A large variety of mechanisms are available for representing statistical summaries. We have specifically selected histograms for this purpose—the use of histograms enables StatiX to maintain scalable and symmetric summaries of both the *structures* of the types as well as the *values* in the data.

Basing statistics on XML Schema types facilitates the re-use of standard XML technology, namely, *validating parsers*, for statistics gathering. Another advantage of type-based statistics is that the granularity of statistics gathering can be easily tuned through schema transformations (*e.g., delete, add, merge* types) that change the types but retain the tag structure (*i.e.,* the same documents can be validated by the transformed schemas).

Using histograms to store structural summaries elegantly captures the data skew prevalent in XML documents. Histograms are attractive because they are simple to implement, have been well-studied, and proven to be effective for selectivity estimation [18, 19]. In addition, they provide a flexible means to adjust to memory requirements (more or fewer buckets can be used depending on the available memory). Moreover, because histograms are already widely used in relational database engines, our framework can be easily integrated with these systems—in fact, we can re-use standard histogram-based estimation algorithms available in relational optimizers.

Finally, StatiX is able to handle a large class of XQuery queries, including tree-pattern queries that involve reference chasing, selections over value ranges, and ancestor-descendent paths.

Contributions. Our main contributions in this paper are:

- a novel statistics framework for XML, based on histograms and XML Schema, that provides concise summaries and accurate estimates for a significant subset of XQuery;
- a description of how to modify an XML Schema validating parser to perform statistics gathering;
- exploitation of XML Schema transformations in order to obtain statistics at different granularities;
- demonstration of the effectiveness of StatiX in a real application, namely, cost-based relational storage design for XML data, which cannot be handled by previous approaches to size estimation;
- experimental results indicating that gathering statistics adds acceptable overhead and that StatiX summaries lead to accurate estimates.

Organization. The remainder of the paper is organized as follows: An overview of the StatiX system architecture is provided in Section 2, and the statistics gathering algorithm is presented in Section 3. In Section 4, we explain how StatiX can be utilized in conjunction with cost-based approaches to XML storage design. Then, in Section 5, we present experimental results that demonstrate the effectiveness of our approach. Related work is reviewed in Section 6. We conclude in Section 7 with a summary of our results and directions for future research.

```
type Show =
 show [ title[ String ],
        Show_year,
        Aka*,
        Review*,
        ( box_office [ Integer ] |
          (seasons[ Integer ],
          Episode* )
        ) ]

type Show_year =
 year [ Integer ]

type Review =
 review [ String ]

type Aka =
 aka [ String ]

type Episode =
 episode [ Aka{0,2} ],
           guest_dir [ String ]* ]
```

Figure 1: XML Schema for Movie Data

2. THE STATIX FRAMEWORK

In this section, we use an example to illustrate the importance of detailed statistical information in order to accurately estimate sizes of XML queries. We then describe the StatiX model and the architecture of the system.

2.1 Motivating Example

The XML schema in Figure 1, inspired from the Internet Movie Database [11], describes information about shows.[1] All shows have a title, a release year, zero or more alternative titles (aka's), and zero or more reviews. If the show is a movie, it also has the box office proceeds, or if it is a TV show, it has the number of seasons during which the show was broadcast, and information about zero or more episodes. Sample data reflective of real-world information that conforms with this schema is shown in Figure 2 (the IDs in the figure are not part of the document and are only included for illustrative purposes, as discussed later in this section).

Estimating Cardinalities of XML Queries. Note that, because XML allows deeply nested structures, evaluating path expression queries may require a large number of joins. This increases the risk for inaccurate estimates since it is known that errors propagate rapidly through multi-way join queries [12]. In addition, the structure of XML Schema defines dependencies among elements that are crucial for cardinality estimation. In our example, the schema specifies that episodes should have at most 2 `aka` elements, and that all shows with an episode must have a `seasons` element. This kind of structural dependency yields non-uniform distributions that are likely to increase the risk for inaccurate estimates.

Take for example the following query, which lists the titles, akas, and reviews of shows that were broadcast after 1991:

```
FOR $s in document(``myshows.xml'')/show,
    $e in $s/episode, $a in $e/aka, $r in $s/review
WHERE $s/year > 1991
RETURN $s/title, $a, $r
```

Applying the access relation rewriting strategy [13] to transform the navigation in the query into join operations over the types defined in the schema, the original XQuery can be represented by the following relational algebra expression:

$$\pi_{title,aka,review}\{\sigma_{year>1991}\{\text{Show} \bowtie \text{Episode} \bowtie \text{Aka} \bowtie \text{Review}\}\}$$

[1] In this paper, for simplicity, we use the XML schema notation for types from the XML Query Algebra [8].

```
<imdb>
<show> <!--ID=1-->
  <title> Fugitive, The </title>
  <year> 1993 </year>
  <aka> Auf der Flucht </aka> <!--ID=12-->
  <review> best action movie of the decade...
         </review> <!--ID=30-->
  <review> Ford and Jones at their best...
         </review> <!--ID=31-->
  <review> top notch action thriller...
         </review> <!--ID=32-->
  <review> Solid action and great suspense...
         </review> <!--ID=33-->
  <box_office> 183752965 </box_office>
</show>

<show> <!--ID=2-->
  <title> X Files, The </title>
  <year> 1994 </year>
  <seasons> 8 </seasons>
  <review> spooky ... </review> <!--ID=34-->
  <episode>
     <aka> Ghost in the Machine </aka>  <!--ID=13-->
  </episode>
</show>

<show> <!--ID=3-->
  <title> Seinfeld </title>
  <year> 1990 </year>
  <seasons> 9 </seasons>
  <review> The best comedy series ever!
    </review> <!--ID=35-->
  <episode>
     <aka> Soup Nazi, The </aka>  <!--ID=14-->
     <aka> Soup, The </aka> <!--ID=15-->
  </episode>
  <episode>
     <aka> Chinese Woman, The </aka> <!--ID=16-->
  </episode>
  <episode>
     <aka> Good Samaritan, The </aka> <!--ID=17-->
     <guest_director> Alexander, Jason </guest_director>
  </episode>
  <episode>
     <aka> Gum, The </aka> <!--ID=18-->
  </episode>
  <episode>
     <aka> Airport, The </aka>  <!--ID=19-->
  </episode>
</show>

<show>  <!--ID=4-->
  <title> Dalmatians </title>
  <year> 1998 </year>
  <review> Cute and entertaining
    </review> <!--ID=36-->
  <review> please, no 103 dalmatians!
    </review> <!--ID=37-->
  <box_office> 50000000 </box_office>
</show>

<show> <!--ID=5-->
  <title> N Sync: Live </title>
  <year> 2000 </year>
  <seasons> 0 </seasons>
</show>
</imdb>
```

Figure 2: Sample Movie Data

From the data of Figure 2, we can infer that the actual size of the result is 1. Had we used a simplistic *uniform distribution* assumption, the estimate would have been a highly inaccurate 17. This simple example shows that a naive approach to XML statistics does not work, and that more detailed information about the structure must be captured in a statistical model in order to derive accurate estimations.

2.2 The StatiX System

StatiX leverages information available in an XML Schema to generate concise summaries that capture both the structural skew as well as information about the values in an XML document. As a result, StatiX is able to provide accurate estimates for a large class of XQuery queries. The diagram in Figure 3 depicts the two main components of StatiX: the **XML Schema Validator**, that simultaneously validates the document against the schema and gathers the associated statistics; and the **XML Schema Transformer**, which enables statistics collection at different levels of granularity. In what follows, we give an overview of these components and describe which and how statistics are gathered in StatiX.

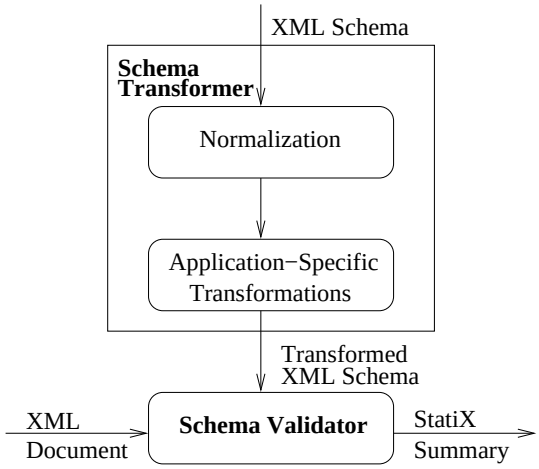


Figure 3: Statistics Collection in StatiX

2.2.1 XML Schema Validator

While in practice the XML Schema Validator comes into play only after the XML Schema Transformer has completed its rewritings, for ease of exposition we will describe the validator first. As mentioned earlier, the validator has two functions: validation of the source XML document against its XML Schema description, and the simultaneous collection of statistics for this schema. The validator gathers statistics on a *per-type* basis. As illustrated in Figure 2, globally unique identifiers (IDs) are assigned to all instances of the types defined in the schema[2] (a similar ID assignment scheme is also implemented in [6], however, they assign an ID to each element in the data). Together with these ID assignments, *structural histograms* are constructed which use the IDs to summarize information about how elements are connected—the histograms support the estimation of the cardinality of each edge in the XML Schema type graph.

For the example presented above, a sample StatiX summary is given in Figure 4. Note that one histogram is created for each type in the schema. For example, for the type Review, StatiX stores the range of IDs for the instances of Review in the document, as well as the ranges of Show IDs that join with reviews in PARENT HISTOGRAM Show (under type Review). This histogram indicates that 4 reviews join with shows with IDs in the range [1,2), 1 review joins with shows with IDs in the range [2,3), and the remaining 3 reviews join with shows with IDs in the range [3,5). Note that all ranges (*i.e.*, ID domains, value domains, and bucket ranges) in StatiX summaries represent intervals closed on the left and open on the right.

While StatiX uses histograms in a novel way to summarize structural information, histograms are also used in a more traditional sense: *value histograms* can be built for types that are defined in terms of base types (*e.g.*, *Integer*). In our example schema, information is stored about the distribution of years within shows: 3

[2]In the schema of Figure 1, type names are shown in sans serif .

```
DEFINE STAT Show {
  CARDINALITY { 5 }
  ID_DOMAIN { 1 TO 6 } }

DEFINE STAT Show_year {
  VALUE_DOMAIN { 1990 TO 2001 }
  BUCKET NUMBER { 2 }
  BUCKETS {
      FROM 1990 TO 1995 COUNT 3,
      FROM 1995 TO 2001 COUNT 2 } }

DEFINE STAT Review {
  CARDINALITY { 8 }
  ID_DOMAIN { 30 TO 38 }
  PARENT HISTOGRAM Show {
    BUCKET NUMBER { 3 }
    BUCKETS {
      FROM 1 TO 2 COUNT 4,
      FROM 2 TO 3 COUNT 1,
      FROM 3 TO 5 COUNT 3} } }

DEFINE STAT Aka {
  CARDINALITY { 8 }
  ID_DOMAIN { 12 TO 20 }
  PARENT HISTOGRAM Show_Episode {
    BUCKET NUMBER { 3 }
    BUCKETS {
      FROM 1 TO 2 COUNT 1,
      FROM 2 TO 6 COUNT 0,
      FROM 6 TO 12 COUNT 7} } }

DEFINE STAT Episode {
  CARDINALITY { 6 }
  ID_DOMAIN { 6 TO 12 }
  PARENT HISTOGRAM Show {
    BUCKET NUMBER { 2 }
    BUCKETS {
      FROM 2 TO 3 COUNT 1,
      FROM 3 TO 4 COUNT 5} } }
```

Figure 4: Statistics Summary for Movie Data

shows were released in [1990,1995), and 2 during [1995,2001).[3]

The combination of structural and value-based information allows StatiX to provide accurate estimates for a large class of XQuery queries. For example, using standard histogram multiplication over the StatiX summary of Figure 4, the estimated size of the query in our example is 4, which is significantly closer to the correct answer (*i.e.*, 1) than the size estimated under the uniform distribution assumption (*i.e.*, 17).

An explanation regarding how cardinalities are computed in the above example is in order: for simplicity and uniformity, we take it to be the cardinality of the number of records that appear in the *flattened* version of the output of the XML query. A similar approach to counting results is taken in the OODBMS literature [13]. Another detail is that for queries that mention tags which do not have an associated type, we check whether the tag has a value histogram: if the histogram is available, it is used in the computation, otherwise, by default, a uniform distribution is assumed.

Handling Shared Types. A crucial feature of the structural histograms is that they naturally distinguish *different* occurrences of the same type in the schema (*i.e.*, types with *multiple* parents). In our example, the type Aka occurs under both Show and Episode. The corresponding histogram (in Figure 4, PARENT HISTOGRAM Show_Episode), stores this information by maintaining three buckets for the parent IDs of Aka: the first bucket contains the range of show IDs; the last contains the range of episode IDs; and the middle accounts for the gap between the values of the show and episode IDs.[4]

[3] In the current system, users may indicate for which tags value histograms must be built by creating new types for these tags.

[4] Note that empty buckets need not be explicitly stored—they are shown here only for clarity of presentation.

Two important points are worthy of note. First, *different* granularities can be used for the different sections in the histogram, *e.g.*, if there are many more shows than episodes, more buckets can be used to represent shows than to represent episodes. This finer-grained information would be useful when there is a selection predicate preceding a join that restricts the join to a subset of the identifiers in the parent type. Second, there is no mandatory requirement that IDs for a given type be contiguous, but when they are not contiguous, more buckets may be needed to capture their distribution. In the remainder of this paper, for simplicity, we will assume that all ID ranges are contiguous and disjoint across types.

In Section 3, we describe our implementation of the Validator, how statistics are gathered and how a standard XML validating parser is extended to generate both structural and value histograms.

2.3 XML Schema Transformer

The ability to specify *types* is an important feature of XML Schema which is not present in DTDs. In XML Schema, there is a distinction between *elements* (*e.g.*, `title [String]` indicates a title element whose content is a string value) and *types* (*e.g.*, Show)—the latter do not appear in the document, but are used during validation as annotations. A variety of transformations, described in more detail below, can be applied which change the way that annotations are performed but do not alter the set of valid documents. We utilize this feature, by applying the appropriate set of transformations in the XML Schema Transformer module, to *tune* the granularity of the statistics gathering process so as to suit the requirements of individual applications.

In the remainder of this section, we illustrate a few representative transformations and explain how they impact the statistics gathering process. (A concrete example on how these transformations can be adapted to a given application are given in Section 4, where we describe the use of StatiX in cost-based storage design for XML.)

2.3.1 Transformations

Consider once again the XML schema of Figure 1. A possible transformation that can be applied to this schema is to *distribute the union* in the definition of Show creating two new types as follows (see Figure 5(a)): Show1 corresponds to movies (which have box office proceeds) and Show2 corresponds to TV shows (which have seasons and episodes). When statistics are gathered under this new schema, separate ranges of IDs are assigned to the different kinds of shows, making it possible to distinguish the occurrences of the children of Show. That is, under the transformed schema, it is possible to know exactly how many reviews are associated with movies and how many reviews are associated with TV shows. This is illustrated in the StatiX summary fragment shown in Figure 5(b).

Another useful, yet simple, transformation is to associate type names with tags. For example, a new type can be created for the tag `guest_dir`: type GuestDirector = `guest_dir [String]`. The presence of this new type implies that statistics are gathered about how these values are distributed among episodes. Given this additional information, precise estimates can be given for queries such as *"Find the guest directors of (TV) shows later than 1991 that were broadcast for more than 10 seasons"*.

Apart from the above transformations, other possible transformations include [2]: (1) merging duplicate definitions; (2) deleting a type by inlining its definition within the definition of its parent; (3) splitting shared types; and (4) unrolling repetitions (*e.g.*, a* can be rewritten as a?,a*).[5]. By combining these transformations, many different schemas, and consequently, many different summaries can be derived that are appropriate for different applications. As we

[5] See [2] for detailed definitions of these transformations

```
type Show =  Show1 | Show2

type Show1 =
  show [ title[ String ],
         Year,
         Aka*,
         Review*,
         box_office [ Integer ] ]

type Show2 =
  show [ title[ String ],
         Year,
         Aka*,
         Review*,
         (seasons [ Integer ],
         Episode* ) ]
```

(a) Transformed schema

```
DEFINE STAT Show1 {
  CARDINALITY { 2 }
  ID_DOMAIN { 1 TO 3 } }
DEFINE STAT Show2 {
  CARDINALITY { 3 }
  ID_DOMAIN { 3 TO 6 } }
DEFINE STAT Review {
  CARDINALITY { 8 }
  ID_DOMAIN { 30 TO 38 }
  PARENT HISTOGRAM Show1_Show2 {
    BUCKET NUMBER { 2 }
    BUCKETS {
      FROM 1 TO 3 COUNT 6,
      FROM 3 TO 6 COUNT 2 } } }
```

(b) StatiX summary for transformed schema

Figure 5: Transformed Schema and Associated Statistics

discuss in Section 4, in cost-based XML storage design, transformations can be applied to further decompose specifically the subset of the schema that is *relevant* to the input workload.

2.3.2 Memory Issues

Transformations that add types clearly lead to more detailed statistics, but they also increase the amount of memory required for the StatiX summary since histograms are kept for each type. If a budget is set for memory, a ceiling in the number of types can be easily set and transformations are applied only until that ceiling is reached. That is, in contrast to the expensive *reductive* process followed by the previous literature, wherein large disk-based representations are whittled down to memory-resident structures (see Section 6 for details), a *constructive* approach is taken in StatiX. Note, however, that establishing the *optimal* set of types and histograms for a given memory budget is clearly a computationally intensive optimization problem.

Most transformations have no real impact on the *algorithmic complexity* of the statistics gathering process, although they might alter the overheads of statistics collection. This is because adding or deleting a type only means that statistics will or will not be gathered for the particular pattern defined by the type. However, this is not true for transformations that add *ambiguity* and lead to schemas that are not *proper* XML Schemas—in XML Schema, types are defined so that there is no ambiguity to ensure that validation can be done with one look-ahead token and in linear time.

Union distribution, which we had discussed earlier in Figure 5, is a good example of an ambiguity-causing transformation. This is because when we distribute the union in Show and create the new types Show1 and Show2, both these new types have the same tag, `show`. The presence of ambiguity in the transformed schemas requires non-trivial extensions to standard validating parsers, including the incorporation of sophisticated machinery such as *tree automata* [20] and may significantly increase the complexity of statistics gathering. In our future work, we plan to investigate how to minimize the impact of ambiguity on the validation process.

2.3.3 Schema Normalization

An important detail that was not mentioned in the above discussion is that, in practice, as a prelude to applying the transformations, the user-supplied XML Schema is first *normalized*. Schema normalization, as defined in [2], is composed of the following set of schema transformations:

1. New types are introduced for all tags that are part of repetitions or that are optional, *e.g.*,

   ```
   type Episode =
     episode [ Aka{0,2} ],
               guest_dir [String]* ]
   ```

   is transformed to

   ```
   type Episode =
     episode [ Aka{0,2} ],
               GuestDirector* ]
   type GuestDirector =
     guest_dir [ String ]
   ```

2. New types are introduced for all complex structures that occur within unions and repetitions, *e.g.*,

   ```
   type Show =
     show [ ...,
            ( box_office [ Integer ] |
              (seasons[ Integer ],
              Episode* )
            ) ]
   ```

   is transformed to

   ```
   type Show =
     show [ ...,
            ( Movie | TVShow )
   type  Movie = box_office [ Integer ]
   type  TVShow =
             seasons[ Integer ],
             Episode*
   ```

   These complex structures (*i.e.*, optional elements, union, repetition) are a major source of structural skew in the data. By ensuring types are defined for these complex structures, normalization provides a controlled way to gather detailed statistics which capture this skew. Having these detailed statistics is important because not all transformations (*e.g.*, union distribution and repetition unrolling) *preserve* the accuracy of the statistics. Therefore, if we initially generate statistics at the *finest* type granularity, it is possible to retain the accuracy in spite of subsequent transformations. This issue is illustrated in Section 4.

3. STATISTICS GATHERING

StatiX exploits XML Schema validation in order to collect statistics. In this section, we first give a brief overview of how schema validation works, along with a few complexity results. We then explain how StatiX modifies the validation process in order to collect statistics. Experimental results obtained with our implementation of the StatiX validator are given in Section 5.

3.1 StatiX and XML Schema Validation

Schema validation [23] is the process by which an XML document is checked against a given XML schema. By exploiting the necessary schema validation process, StatiX is able to amortize the cost of statistics collection. Another benefit of this approach is that StatiX can extend existing implementations of XML parsers and XML Schema validators [25, 9]. Our initial prototype of the statistics collector was built on top of Galax [9], a freely available schema validator.

DTDs and XML Schemas impose certain restrictions on the regular expressions they allow in order to achieve determinism, and techniques to generate deterministic tree automatas directly from DTDs or XML schema have been proposed [3, 4]. Checking whether a given tree belongs to the language defined by this automata, *i.e.,* performing validation, can then can be executed in linear time in the size of the data. Other kinds of tree automatas are more expressive but can result in exponential computations [20]. The validation algorithm we use leverages these restrictions in order to perform validation by directly using a deterministic top-down tree automata.

When the validation process is successful, it results in a type assignment for nodes or sequences in the XML document. This type assignment is the basis for statistics generation in StatiX. Intuitively, we just proceed with schema validation and count the number of occurrences of each type. It is important to note that because of the determinism, there is always a unique way to perform this type assignment, and this ensures the consistency of the statistics gathering process.

3.2 Identifier Assignment

In order to gather information about how different elements relate to each other, StatiX assigns unique IDs to type instances in the document. Ideally, the assignment should be such that for each type, the ranges of the IDs covered by its instances are (a) internally contiguous, and (b) disjoint from that of other types. These goals are approximated as follows: Offline, each type in the XML Schema is assigned a unique *type ID*. Then, for each type, we maintain the following structure during validation: (a) a counter for the next available type ID; (b) the set of all parent IDs that have been processed for that type. (This is possible because the validation process is performed top down and a parent is always processed before its children.) Every time a new instance of a type is encountered, it is assigned a *local ID* using the counter associated with the type, and its parent ID is added to the parent set for the corresponding type. The *global ID* of the element is obtained by concatenating the associated type ID and its local ID. Note that even though types in the schema can form a graph, XML documents are always trees—as a result, the top-down validation approach naturally deals with shared types (for instance, the shared type Aka in Figure 4).

Currently, we fix the size of counters to be two bytes, thereby permitting up to 65536 elements per type. In the event a counter attains its maximum, we can safely bail out of this problem by assigning a new unique ID to the type and resetting the local counter—subsequent instances of the type in the document are assigned IDs based on these new values. Note that this process may result in non-contiguous ID ranges for a type.

3.3 Histogram Construction

Once validation is completed and information has been gathered about the parent IDs for each type, we proceed to construct a more concise representation using histograms. A variety of histogram constructions have been described in the literature [19, 18]—the most common are *equi-width* histograms, wherein the domain range covered by each histogram bucket is the same, and *equi-depth* histograms, wherein the frequency assigned to each bucket is the same. Since it has been shown in [16] that equi-depth histograms result in significantly less estimation error as compared to equi-width histograms, we have implemented the former in StatiX (for details of the construction process, please refer the afore-mentioned literature).

4. APPLICATION TO COST-BASED XML-TO-RELATIONAL MAPPING

The LegoDB *cost-based* XML-to-relational storage mapping engine was recently described in [2], representing a departure from prior approaches that were largely based on heuristics [22, 7]. The LegoDB system generates relational configurations that are *efficient* for a given application. Given inputs consisting of (a) an XML Schema, (b) XML data statistics, and (c) an XML query workload, it examines a space of possible ways to *decompose* the XML document into relations. LegoDB exploits the type structure of XML Schemas to generate a space of alternative relational configurations. A fixed mapping is defined that maps XML Schema types into tables. By repeatedly applying XML-specific schema rewritings that alter the type structure of the schema (but retain the document structure) followed by this fixed mapping into relations, LegoDB generates a space of possible relational configurations. Each configuration is composed of:

- *a relational schema*, derived from the transformed schema using the fixed type-to-relation mapping;
- *a statistical summary for each relation in the schema*, derived from the given XML data statistics;
- *an SQL workload*, derived from the given XQuery workload and the relational schema.

LegoDB uses a traditional relational optimizer (*e.g.,* [10]) to estimate the cost of the derived relational configurations and selects the configuration with the lowest cost. The cost of a configuration is computed as the cost of processing its SQL workload on its relational schema on the basis of its statistical summary. Note that the optimizer is used as a *black box* in this process and is completely unaware of the XML ancestry of its inputs.

Clearly, the accuracy of the optimizer's cost estimates, and hence the efficacy of LegoDB, crucially depends on the accuracy of the statistics. If the statistics cause the optimizer to over- or underestimate the cost of alternative configurations, desirable choices may be discarded and substituted with poor configurations. As such, for LegoDB, we need a result estimator that is able to compute accurate statistics for different relational configurations. In principle, LegoDB could derive these statistics for each new configuration by loading the XML source data into relations and then analyzing the contents of these relations. However, this is not a feasible alternative since the solution does not scale—a large number of distinct relational configurations are considered, and processing each configuration in the described manner can become prohibitively expensive, especially for large documents.

A more scalable approach is the following: First compute the statistical summary for the initial XML Schema. Then, as and when a new alternative XML Schema is generated by the LegoDB mapping engine

- *incrementally derive* the statistical summary for this XML Schema, and
- *translate* this derived statistical summary to statistics for the associated relational schema.

The previous sections have explained how StatiX can be used to build an accurate fine-grained statistical summary for the initial XML Schema. In the remainder of this section, we describe how StatiX is used in LegoDB.

Gathering XML Statistics An important requirement for LegoDB is that the XML statistical summary should contain enough information that allows *precise* relational statistics to be derived for each

```
type Show = Show1 | Show2

type Show1 =
  show [ title[ String ],
         Year,
         Review*,
         Movie ]

type Show2 =
  show [ title[ String ],
         Year,
         Review*,
         TVShow ]

type Review =
  review [ String ]
...
```

(a) Normalized schema

```
DEFINE STAT Show1 {
  CARDINALITY { 2 }
  ID_DOMAIN { 1 TO 3 } }
DEFINE STAT Show2 {
  CARDINALITY { 3 }
  ID_DOMAIN { 3 TO 6 } }
DEFINE STAT Review {
  CARDINALITY { 8 }
  ID_DOMAIN { 30 TO 38 }
  PARENT HISTOGRAM Show1_Show2 {
    BUCKET NUMBER { 2 }
    BUCKETS {
      FROM 1 TO 3 COUNT 5,
      FROM 3 TO 6 COUNT 3 } } }
```

(b) StatiX summary for normalized schema

```
type Show =
  show [ title[ String ],
         Year,
         Review*,
         box_office [ Integer ]
         (seasons [ Integer ],
         Episode* ) ]
         ]
type Review =
  review [ String ]
```

(c) Merged schema

```
DEFINE STAT Show {
  CARDINALITY { 5 }
  ID_DOMAIN { 1 TO 6 } }
DEFINE STAT Review {
  CARDINALITY { 8 }
  ID_DOMAIN { 30 TO 38 }
  PARENT HISTOGRAM Show {
    BUCKET NUMBER { 1 }
    BUCKETS {
      FROM 1 TO 6 COUNT 8 } } }
```

(d) StatiX summary for merged schema

Figure 6: Schema Transformation and Statistics Derivation

configuration examined by the system. The *schema normalization* procedure of StatiX (described in Section *2.3.3*) delivers the XML Schema at the *finest type granularity*, and collecting the statistics with respect to this schema allows StatiX to derive the accurate statistics for any configuration generated by the LegoDB search engine *without looking at the data*.

An example of this process is presented in Figure 6. The initial normalized XML Schema and the associated StatiX summary are shown in Figures 6(a) and 6(b), respectively. During the search process LegoDB applies a series of transformations, that may include, for example, merging the types Show1 and Show2, and inlining the definitions of Movie and TVShow, which would lead to the schema of Figure 6(c). For this new schema, it is possible to construct an accurate structural histogram for the merged type Show, as shown in Figure 6(d), using only the information in the summary for the initial schema (Figure 6(b)). This is achieved by simply (a) adding up the cardinalities and combining the domains of Show1 and Show2, and (b) combining the buckets in the parent histograms of Review.

Note that, in contrast, had the statistics been collected for the XML Schema with the merged type Show, and a subsequent rewriting had split it into Show1 and Show2, it would have been impossible to construct the accurate histograms for the new edges Show1-Review and Show2-Review. This is because, although we are given the histogram for Show-Review, the *partition* of the Show ids into Show1 and Show2 is not well defined.

Deriving Relational Statistics Since StatiX provides type-based statistics for a given XML Schema, mapping this statistical summary to relational statistics follows the fixed type-to-relation mapping defined in LegoDB [2]. The relational statistics generated contain:

- a trivial (single bucket) histogram covering the domain of the type for the ID column (the primary key);[6]
- a histogram translated from the value histogram for each column that represents a base-type sub-element; and
- for the foreign key column, a histogram that is constructed by merging the PARENT HISTOGRAMs of the types which correspond to the foreign key.

5. EXPERIMENTS

We have evaluated the performance of StatiX on a variety of XML documents covering a range of sizes, document characteristics and application domains. We report results here for two representative datasets: the XMark benchmark [26], which creates deeply nested documents that model an Internet auction site, and the Internet Movie Database (IMDB) [11], which has a mostly flat structure and contains information about movies.[7]

The performance was evaluated with regard to the following metrics:

- The overhead imposed by statistics collection on the validation process,
- The space occupied by the statistical summaries, and
- The accuracy of the cardinality estimates.

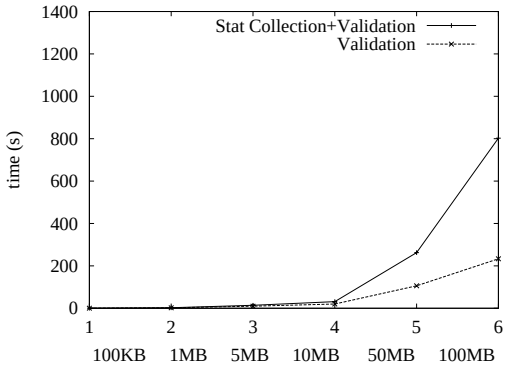
The above metrics were evaluated for a variety of schemas including the original XML Schema, its normalized version, as well as the schemas obtained after various transformations. Our experiments were conducted on a PIII-750 MHz machine, running Redhat Linux 6.2, with 2 GB main memory and 18 GB local SCSI disk.

5.1 Statistics Collection

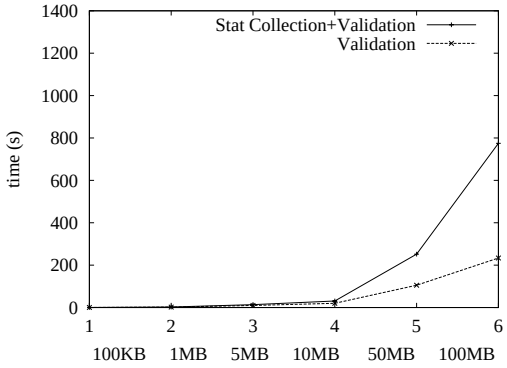
For a range of XMark document sizes going from 100KB to 100MB, Figure 7 shows the added overhead of statistics collection for three different versions of the XMark schema: *normalized* (consisting of 125 types); *split-item* wherein the type corresponding to `item` is split to eliminate its shared occurrences (resulting in 132 types); and *split-all* wherein all shared types in the schema are split (resulting in 1887 types). The algorithmic complexity of validation is linear in the size of data and thus, it is not impacted

[6] Multiple buckets may be used if the IDs are not contiguous.

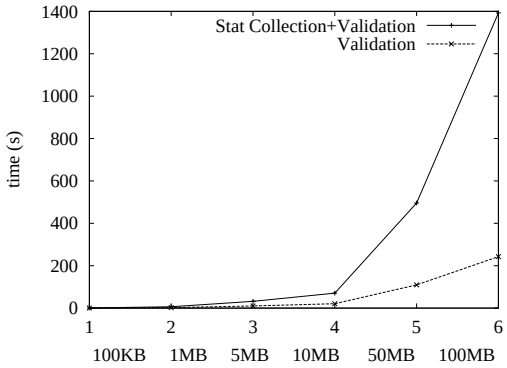
[7] We created the XML Schemas for both XMark, for which only a DTD is available, and IMDB, for which no DTD is provided.



(a) Normalized



(b) Split Item



(c) Split All

Figure 7: Overheads for Statistics Collection

by the number of types in the schema. Statistics gathering, on the other hand, does depend on the number of types, as Figure 7 shows. A major source of overhead in our prototype are the structures that are built (and maintained) by Galax [9] to construct the histograms; these structures are not optimized for large documents. However, with careful tuning or by using sampling-based techniques, this can be improved.

For comparative purposes, we also evaluated the "Twigs" [6] approach (described in more detail in Section 6) with respect to the time taken to build the summary structures. The Twigs software (which was provided to us by its authors) excutes in two phases: First, a preprocessing phase in which the XML document is parsed and stored into a format suitable to generate the summary structure; and, second, a building phase, where the summary structure is actually constructed. We ran Twigs on a 10MB document as well as a 50MB XMark document. The preprocessing phase took a considerable amount of time—around 10 minutes for the 10MB document and over an hour for the 50MB document. The building phase, on the other hand, took about 90 seconds for the 10MB document and 450 seconds for the 50MB document. The summary contained 1000 nodes with a signature size of 10.

In contrast, the worst case performance of StatiX, corresponding to the "Split All" schema (Figure 7(c)), takes approximately 70 seconds for processing the 10MB document and 500 seconds for the 50MB document—note that these figures include the *entire* set of operations: parsing, validation and statistics collection.

5.2 Summary Sizes

The sizes of the StatiX summaries are directly proportional to the number of incoming edges into complex types ($ctypes$), number of base types for which statistics are gathered ($btypes$), and the size of the histograms which is determined by the number ($nbuckets$) and size ($bsize$) of buckets:

$$summary\_size = \begin{array}{l} \sum_i (ctypes_i * nbuckets_i * bsize_i) + \\ \sum_j (btypes_j * nbuckets_j * bsize_j) \end{array}$$

where i ranges over the IDs of complex types and j over the IDs for the base types. For example, with the 100MB XMark document, a normalized schema, and 30 buckets per histogram (in our implementation, each bucket consumes 8 bytes), the total summary size was only 25KB, whereas for the split-all schema it was around 200KB. This clearly shows that StatiX summaries are extremely concise as compared to the documents whose statistics they are representing.

It is important to note here that in previous approaches, the space space overhead is proportional to the size of the *data*. With StatiX, however, the overhead is proportional to the size of the *schema* since it is based on the number of types. As schemas are typically much smaller than the data, major savings can be realized with our approach. For comparative purposes, we evaluated the memory consumption of Twigs on the 10MB XMark document and found that even for this relatively small-sized document it took about *double* the space required by StatiX (as we discuss below, StatiX is able to derive estimates that are more accurate even with a significantly smaller summary). We can expect that this difference would only grow larger with increase in document size.

5.3 Estimation Accuracy

To assess the estimation accuracy of StatiX, we ran a variety of queries against the 100MB version of XMark (using the normalized schema) and against a 44MB version of the IMDB dataset. We describe these results, obtained using 30 buckets per histogram, below. As an indicator of the amount of skew encountered in the data we also evaluated the estimates that would have been obtained from a uniform distribution modeling approach.

XMark Dataset. The list of XMark queries used in our experiments is given in Table 1 (please refer to [26] for details of the XMark document structure). Some of these queries are derived from the query set supplied with the benchmark, while others were

Q	Description
X1	all buyers of European items
X2	all bidders for European items
X3	names of sellers who sold an item
X4	names of people and their emails
X5	list of item ids and buyer's ids in closed auctions
X6	names of people with homepages
X7	names and descriptions of australian items

Table 1: XMark Benchmark Query List

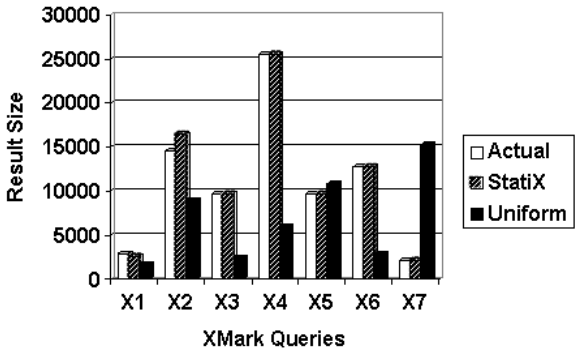


Figure 8: Accuracy of StatiX vs Uniform for XMark queries

constructed by us. Note that queries X1, X2 and X3 feature *value-based joins*, while X4, X5, X6 and X7 only involve *tree patterns* (the joins are *structure-based*). The number of joins in each of the queries is listed below:

- X4, X6: 2 joins involving 3 types
- X7: 3 joins involving 4 types
- X5: 4 joins involving 5 types
- X3: 5 joins involving 6 types
- X1, X2: 8 joins involving 9 types.

Figure 8 shows the cardinality estimates for the above queries using StatiX and uniform distribution, as well as the actual query sizes. It is gratifying to note here that the StatiX estimates are all within reasonable error margins and, in fact, for most of the queries, the estimate contains virtually no error. These results become even more attractive when taken in conjunction with the fact that the StatiX summaries are very small in size. For example, with queries X6 and X7, where StatiX has virtually no error, Twigs, which utilized double the storage (see above), had 3% and 53% error, respectively.

We also note in Figure 8 that using a uniform distribution assumption results in large errors. An important factor that contributes to these errors is the structural skew that arises as a consequence of *shared types*. For example, query X5 requires the following join: *Closed_auction* $\bowtie$ *Itemref* $\bowtie$ *Buyer*. But since both closed and open auctions have *Itemref*s, the uniform estimator assumes that 50% of *Itemref*s would join with *Closed_auction*—as the 11.5% error indicates, this assumption does not hold for the XMark dataset. These errors are magnified when (a) the queries involve types with greater number of parents, *e.g.*, queries X4 and X6 involve the type *Name* which is shared among 3 types, leading to errors greater than 70%; and (b) the queries involve multiple

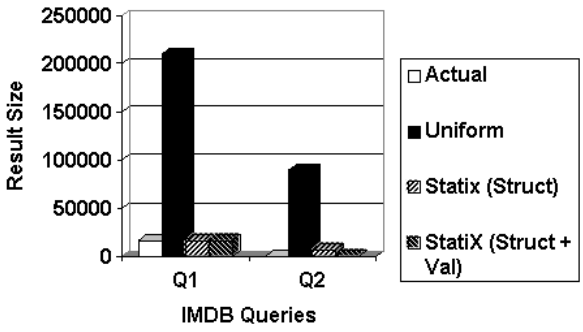


Figure 10: StatiX Accuracy for the IMDB dataset

shared types, *e.g.*, query X7 includes the types *Item* (shared among 6 types) and *Name* (shared among 3 types), leading to an estimate that is 7 times larger than the actual query size.

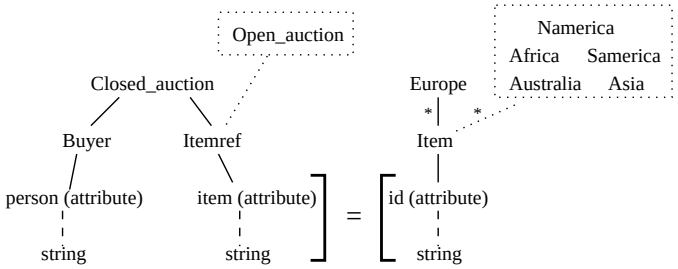
Let us now examine the queries X1 and X3, whose pictorial representation is shown in Figure 9. Note that these two queries are quite similar, but whereas the StatiX estimate for X3 is precise, the estimate for X1 is off by 10.4%. This can be explained as follows: In query X1, the type *Item* (and consequently *id*) is shared among 6 parents (*i.e.*, Europe, North America, etc.). When the join is performed between *Itemref* and *Item*, StatiX estimates the cardinality correctly. However, as it does not keep track of the *correlation* between the *id* values and *Item* values, when the selection to filter out European items of closed auctions is subsequently applied, error is introduced in the estimate. In contrast, for query X3, because the types of the join attributes are not shared, StatiX derives precise estimates.

IMDB Dataset. We now move on to evaluating the accuracy of StatiX on the IMDB dataset. For this dataset, we focus on the subset of the schema that covers movies and alternative titles (aka's)—a movie can have from zero to an unlimited number of aka's. This subset has a total of 113,269 movie elements and 47,756 aka elements. The skewness of this data is indicated by the fact that of the 113,269 movies, only 30,506 have at least one aka, while 9 movies have 26 aka's.

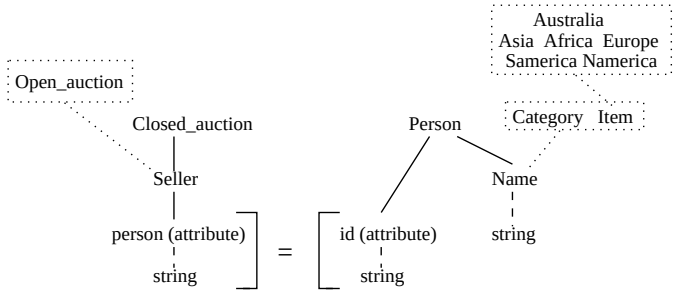
In Figure 10, we show cardinality estimates for the following queries over the IMDB data: Q1:*Find title, year, and pairs of akas for all shows*; and Q2:*Find title, year, and pairs of akas for shows released later than 1990*. These are tree-pattern queries and require a 3-way join (*show* $\bowtie$ *aka* $\bowtie$ *aka*). Notice that under uniform distribution assumption, the cardinality over-estimates are unacceptably high: 12 times for Q1 and 147 times for Q2. In marked contrast, using the structural histograms in the StatiX summary, the estimates for Q1 are far better—within 1% of the actual size. For Q2, however, which includes the conditional check for $year > 1990$, structural histograms alone are not sufficient to generate a precise estimate: in the absence of information about the distribution of year values across shows, uniform distribution is assumed for these values, resulting in the large error (12 times over-estimate). However, when a value histogram (with 5 buckets) is added for the year element, the error becomes significantly reduced.

6. RELATED WORK

A variety of proposals for XML result estimators have appeared in the recent literature [14, 6, 1, 24, 17]. McHugh and Widom



(a) Query X1



(b) Query X3

Figure 9: Pictoral Representation of XMark Queries X1 and X3

[14] proposed a scheme wherein statistics about *all* subpaths p of length $\leq k$ that occur in the XML document are explicitly stored (k is a tunable parameter). These statistics are sufficient to give accurate answers for queries that involve path expressions of length $\leq k+1$. For queries with longer paths, the statistics of the k-length subpaths occurring in the query are combined to estimate the result cardinalities.

In [6], Chen et al propose a scheme that captures *correlations between paths*, resulting in accurate estimation of the so-called *twig* or tree queries. Their strategy consists of gathering counts for frequently occurring twiglets in the data, and then assigning each twiglet a "set hash" signature that captures the correlations between the subpaths in the data tree. Query selectivity is then estimated by combining the hash signatures of the twiglets occurring in the query. While this work is restricted to tree-structured data, the more general problem of handling queries on *graph-structured* XML documents, involving chasing through references (*i.e.*, idrefs), is addressed in [17].

Aboulnaga et al [1] propose and evaluate two memory-efficient data structures for estimating sizes of *simple* path expressions (as opposed to twigs): a *summarized path tree*, which collapses the tree corresponding to the original XML document by deleting or coalescing low-frequency nodes; and a *summarized path table*, which stores the counts of all frequently occurring paths of length $\leq k$.

While the above proposals represent novel and ingenious approaches to the estimator problem, they have some limitations. Notably, *they focus on schemaless semistructured data—i.e.*, the XML source document comes in an "as is" condition. However, in a growing number of data-oriented applications, the use of schemas is commonplace. (See [15] for descriptions of several applications and standardized XML Schemas.) As we have described, this schema information can be used to improve the quality of the statistics as well as reduce their storage overheads. Wu et al [24] proposed a method that does take schema information into account. They defined *position histograms* that capture the ancestor-descendent relationships among nodes in the document. Even though these histograms can be used to efficiently answer path queries containing the descendent construct, they are not able to capture skew that is derived by other important structural constraints (*e.g.*, union, repetition) present in typical XML Schemas.

Another limitation of previous works is that they *support a limited class of XML queries*. For example, [1] is restricted to simple (non-branching) path expressions in the document tree and cannot handle *equality predicates* or queries that involve reference chasing. [6] cannot efficiently handle *value range selections, e.g.,* a query such as $1991 \leq Year \leq 2000$, cannot be handled unless converted to separate sub-queries for each of 1991, 1992, . . . , 2000. Note that this conversion requires domain knowledge about the legal data values in the range. Further, the translation into subqueries and the subsequent estimation by integration of the individual sub-query results is likely to be expensive. [24] constructs a separate position histogram for each distinct predicate—this approach has limited scalability and may be infeasible for ad-hoc query workloads.

Except for [17], existing approaches for XML estimators perform summarization by *reduction* of fully-specified structures. For example, in [1], the complete path-tree for the entire document is built on disk and then whittled down to an in-memory structure. This may entail considerable disk activity during the reduction process. Instead, the alternative approach of *construction* adopted by StatiX, wherein the statistics are built up until the memory budget is exhausted, is more scalable and less resource-intensive.

In some cases, specialized data-structures such as suffix-trees [6, 1] and data-guides [14] have been employed to host the statistical information—these structures are relatively complicated to implement. Our choice, on the other hand, of histograms as the statistics-maintaining structure results in a simple and scalable implementation.

Lastly, in some of the schemes, the addition of new nodes in the document graph may have *global side-effects*—that is, necessitate changes in several locations in the statistics-maintaining structure. For example, in [6], the addition of a node may require the updating of all signatures occurring in the path leading to the node. Such side-effects do not occur with StatiX, however, since the histograms are schema-based, not document-based; therefore, it is only the histograms associated with the types of the added nodes whose values may have to be updated.

7. CONCLUSIONS

In this paper, we have defined an XML statistics model and presented StatiX, a system that implements this model. StatiX leverages the XML Schema data model, schema rewriting transformations, and histograms to provide simple, concise, flexible, and scalable data summaries for query selectivity estimation, a critical input in database configuration and usage.

The advantages of our approach over the prior art include handling a wider spectrum of queries, localized impact of document updates, and a constructive approach to memory utilization. Further, from an implementation perspective, statistics gathering is integrated with a standard XML validation module, making it attractive for hosting in current systems.

Experiments with large data sets and complex queries, including those derived from XMark, against a relational backend indicate that StatiX provides highly accurate query selectivity estimates, especially as compared to simplistic uniform distribution assumptions. The computational overhead due to statistics collection is also quite reasonable, considering that this collection is a one-time operation—our normalization process ensures that statistics for all rewritten schemas can always be derived from the initial statistics. Further, our overheads are considerably smaller in magnitude than those incurred by previous approaches.

Currently, StatiX only supports XQuery queries that can be translated into SPJs. In future work, we plan to add support for both aggregates and recursion. Other issues that we intend to explore include usage of sampling to reduce the overhead of statistics collection, and the efficient handling of the ambiguity caused by transformations such as union distribution.

Acknowledgements

We thank Zhiyuan Chen and Divesh Srivastava for generously providing the Twigs software and helping us with its installation and use.

8. REFERENCES

[1] A. Aboulnaga, A.R. Alameldeen, and J.Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proceedings of VLDB*, pages 591–600, 2001.

[2] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of ICDE*, pages 64–75, 2002.

[3] A. Brüggemann-Klein. Regular expressions into finite automata. *TCS*, 120(2):197–213, 1993.

[4] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[5] D. Chambelin, J. Clark, D. Florescu, Jonathan Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C Working Draft, June 2001.

[6] Z. Chen, H.V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R.T. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of ICDE*, pages 595–604, 2001.

[7] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proceedings of SIGMOD*, pages 431–442, 1999.

[8] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra, February 2001. http://www.w3.org/TR/2001/WD-query-algebra-20010215.

[9] Galax system, October 2001. http://db.bell-labs.com/galax/.

[10] G. Graefe and W. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of ICDE*, pages 209–218, 1993.

[11] Internet Movie Database. http://www.imdb.com.

[12] Y. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM TODS*, 18(4):709–748, 1993.

[13] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proceedings of VLDB*, pages 290–301, 1990.

[14] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, 1999.

[15] XML query language (xql). http://www.oasis-open.org, 2001.

[16] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of SIGMOD*, pages 256–276, 1984.

[17] N. Polyzotis and M. Garofalakis. Statistical synopses for graph structured XML databases. In *Proceedings of SIGMOD*, 2002.

[18] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of VLDB*, pages 486–495, 1997.

[19] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of SIGMOD*, pages 294–305, 1996.

[20] G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages*, volume 3. Springer Verlag, 1997.

[21] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of SIGMOD*, pages 23–34, 1979.

[22] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of VLDB*, pages 302–314, 1999.

[23] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Working Draft, February 2000.

[24] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for xml queries. In *Proceedings of EDBT*, 2002.

[25] Xerces java parser 1.4.3. http://xml.apache.org/xerces-j/.

[26] Xmark. http://monetdb.cwi.nl/xml.