INCREMENTAL MAINTENANCE OF SCHEMA-BASED XML STATISTICS

Maya Ramanath Lingzhi Zhang[1] Juliana Freire[2] Jayant R. Haritsa

Technical Report
TR-2004-05

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

Portions of the work presented in this report have appeared in the Proceedings of the *International Conference of Data Engineering*, Tokyo, Japan, April 2005.

[1]OGI/OHSU
[2]OGI/OHSU. Current address: School of Computing, University of Utah, Salt Lake City, UT 84112; juliana@cs.utah.edu.

Abstract

Current approaches for estimating the cardinality of XML queries are applicable to a static scenario wherein the underlying XML data does not change subsequent to the collection of statistics on the repository. However, in practice, many XML-based applications are dynamic and involve frequent updates to the data. In this paper, we investigate efficient strategies for incrementally maintaining statistical summaries as and when updates are applied to the data. Specifically, we propose algorithms that handle both the addition of new documents as well as random insertions in the existing document trees. We also show, through a detailed performance evaluation, that our incremental techniques are significantly faster than the naive recomputation approach; and that estimation accuracy can be maintained even with a fixed memory budget.

1 Introduction

The database community has invested substantial research in recent years on developing systems for the efficient storage and querying of XML data. A component that is essential to the success of such systems is the *result estimator*, which estimates the *cardinalities* of the results of user queries [9]. These 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.

Several techniques for estimating XML query cardinalities have appeared in the recent literature, including [1, 5, 13, 14, 20, 21, 26]. These proposals differ in many aspects, from the summary data structures used to the class of supported queries. An especially important distinction is in terms of the *meta-data* associated with the documents – while most of the proposals focus on *schemaless* semistructured data, the StatiX system [9] leverages the schema information to improve the quality of the statistics as well as reduce the storage overheads.

Statistics Maintenance. A common lacuna of the above-mentioned proposals is that they primarily address *statistics production*, but not the equally important issue of *statistics maintenance*. This is a critical shortcoming since many XML applications are dynamic and frequently *update* the underlying XML data. In the absence of statistics maintenance, the cardinality estimates could go completely haywire due to the lack of correspondence between the original statistics and the current database contents. Further, while updates to XML documents may be typically expected to be "appends", as in the case of a data warehouse, it is also possible to have applications that insert, modify, or delete at random locations *within* the existing document. For example, an XML workflow application that keeps track of customer purchase orders may dynamically update book-keeping information about the status of the order as it navigates through the order-processing cycle.

Periodically recomputing the statistics from scratch on the updated documents is an obvious choice to cater to the XML update problem. But since recomputation requires *the whole document to be parsed*, it can be prohibitively expensive [17] if recomputations occur frequently, especially for large documents. This is especially problematic for statistics collection techniques that make multiple passes over the data (*e.g.,* [20]). Further, if recomputations are not adequately timed, stale statistical summaries may lead to unacceptable estimation errors. We argue the case

here that it is preferable to incrementally update the XML statistics and to use recomputation only as a comparatively infrequent backup mechanism.

Technical Challenges. In this paper, we present new techniques to incrementally update XML statistical summaries in parallel with the receipt of document updates. We assume that a detailed accurate summary of the data, created at the document loading time, is initially made available, and then, as and when updates are received, this summary is also correspondingly updated. Specifically, given an initial document D and its summary S, and a stream of updates $U = U_1, U_2, \ldots, U_m$ comprising of inserts, deletes or modifications, the goal is to efficiently and incrementally create summaries, $S^1, S^2, \ldots, S^m$, such that the accuracy of these summaries are comparable to those obtained with a recomputed-from-scratch summary $S_R^1, S_R^2, \ldots, S_R^m$. Moreover, this should be achieved within a *fixed memory budget* (that is, the incremental approach has the same resource constraints as recomputation).

Incremental maintenance of data statistics per se is not a new issue to the database community, having been previously addressed in the context of relational database systems (see *e.g.,* [11]). However, what is novel in the XML context is that statistics about both *structure* and *value* have to be maintained. That is, while in an RDBMS, there is *no* difference, as far as the statistics go, between the insertion of a tuple in the middle of a relation or the appending of the same tuple at the end, the *location of the update* is always an issue in XML. Secondly, the *size of the update* in an RDBMS can only be either a single tuple or a set of tuples. But, in an XML environment, the update could be an arbitrarily complex XML fragment, or sets of fragments. For example, the update could require inserting sub-trees at various locations in the original document. Thus, maintaining accurate statistics for XML databases poses a fresh set of problems as compared to those tackled in prior systems.

The IMAX Technique. Our solution to the XML statistics maintenance problem is an algorithm called **IMAX** (Incremental MAintenance of XML statistics), which we present in detail in the remainder of this paper. IMAX is built around the recently-proposed StatiX framework [9], which not only produces concise and accurate summaries for XML documents, but also has several features that make it attractive in a dynamic scenario. For example, StatiX captures order information among the elements in a document through the document schema and its numbering scheme (see Section 2 for details). This information makes it possible to estimate the location of updates — a key step in IMAX. In addition, its use of histograms to uniformly capture structural and value skew simplifies adjusting summaries to fixed memory budgets, and also permits the re-use of well-known techniques for incremental histogram maintenance.

An important extension that we make to the StatiX framework is the use of *two-dimensional* value histograms (instead of the originally proposed 1D histograms) to capture the correspondence between the node ids and their values. Not only does the use of 2D histograms improve cardinality estimation in StatiX, but is also a key factor in the effectiveness of IMAX. An empirical evaluation of IMAX (with both 1D as well as 2D histograms) over a variety of XML documents and update streams demonstrates that IMAX provides, at a marginal run-time cost, accuracy comparable to the brute-force recomputation approach, even with a fixed memory budget.

2 Overview of StatiX

In this section, we provide background material on the StatiX framework. StatiX uses the *document schema* to specify which statistics to gather – specifically, statistics are gathered for the *types* defined in a given schema. This has several benefits, notably: a standard validating parser (*e.g.,* Xerces [27]) can be used to gather the statistics as the document is parsed, amortizing the cost of statistics collection; and the granularity of statistics can be easily tuned for a given application by adding type definitions for relevant elements and by applying schema transformations (see [9] for details).

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="IMDB" type="Imdb"/>
<xsd:complexType name="Imdb">
   <xsd:sequence>
       <xsd:element name="SHOW" type="Show"
            minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Show">
   <xsd:sequence>
      <xsd:element name="TITLE" type="xsd:string"/>
      <xsd:element name="YEAR" type="Year"/>
      <xsd:choice>
         <xsd:element name="MOVIE">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="BOXOFFICE" type="xsd:string"/>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="TV" type="Tv"/>
      </xsd:choice>
      <xsd:element name="AKA" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="REVIEW" type="Review"
           minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:simpleType name="Year">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1900"/>
      <xsd:maxInclusive value="2000"/>
   </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="Tv">
   <xsd:sequence>
      <xsd:element name="SEASONS" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Review">
   <xsd:sequence>
      <xsd:element name="RATING" type="xsd:string"/>
      <xsd:element name="COMMENT" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

(a)

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

DEFINE STAT Review {
  CARDINALITY { 16 }
  ID_DOMAIN { 1 TO 17 }
  PARENT HISTOGRAM Show {
    BUCKET NUMBER { 2 }
    BUCKETS {
      FROM 1 TO 4 COUNT 10,
      FROM 4 TO 6 COUNT 6 } } }

DEFINE STAT Tv {
  CARDINALITY { 2 }
  ID_DOMAIN { 1 TO 6 }
  PARENT HISTOGRAM Show {
    BUCKET NUMBER { 1 }
    BUCKETS {
      FROM 1 TO 6 COUNT 2 } } }

DEFINE STAT Year {
  VALUE_DOMAIN { 1990 TO 2001 }
  NUM_DISTINCT {5}
  BUCKET NUMBER { 2 }
  BUCKETS {
    FROM 1990 TO 1994 COUNT 3,
    FROM 1994 TO 2001 COUNT 2 } }
%
```

(b)

Figure 1: IMDB schema and the corresponding StatiX summary

The successful validation of an XML document against a given schema results in the assignment of types (defined in the schema) to the nodes in the document [23]. StatiX leverages this information to build the statistical summaries. Intuitively, as the document is validated, StatiX

keeps track of the number of occurrences of each type, and how the instances of a given type are distributed over the instances of its parent type(s).

Statistics gathering proceeds as follows. Each type defined in the schema is associated with a unique *type id*. As a document is parsed and occurrences of a given type are encountered, a new sequential *node id* is assigned to each occurrence. The concatenation of type id and node id uniquely identifies a given node in the document tree. Note that the order of occurrence of the type in the document determines the order in which node ids are assigned. For each type defined in the schema, StatiX has an associated *parent set*. Since validation is performed in a top-down fashion, and a parent is always processed before its children, for each type instance encountered, the id of the parent node is incrementally added to the parent set of the corresponding child type. This information is later summarized in a *structural histogram*, which supports cardinality estimation for each edge in the XML Schema type graph.

Assigning *contiguous* ids to a given type is critical to building accurate and concise histograms—the use of non-contiguous ids may result in large gaps within buckets as well as betweeen buckets. Since equi-depth histograms result in significantly smaller estimation errors as compared to equi-width histograms [19], we have implemented the former in StatiX. Besides structural information, StatiX also captures value distributions at the leaf-node level using *value histograms*. While structural histograms are unique to the XML context, value histograms are commonly used in traditional relational storage systems.

An example of an XML schema and a possible StatiX summary corresponding to this schema is shown in Figure 1. The schema describes a database which contains information about shows. A show can be either a movie or a TV show; has a title and year of release; and may contain zero or more reviews, and zero or more alternative titles (*i.e.,* `AKA`). The summary contains statistical information about all types defined in the schema. For each complex type, it records the type cardinality, *i.e.,* the number of occurrences of that type in the document; its id range; and its parent histogram. For example, the type Review has cardinality 16; ids ranging from 1 to 16[3]; and a parent histogram corresponding to Show, which indicates that there are 10 instances of `REVIEW` under `SHOW`s with ids from 1 to 3 and 6 instances under `SHOW`s with ids from 4 to 5. Simple types, that correspond to elements with atomic content, are associated with value histograms. For example, the type Year has a value histogram indicating that there are 3 occurrences of Year with values between 1990 and 1993, and 2 occurrences with values between 1994 and 2000.

Estimating Query Cardinality in StatiX. StatiX estimates the cardinality of XML queries using histogram multiplication. Since path queries are expressed in terms of element (tag) names, and StatiX collects statistics for types, the tags in the query are first mapped to the corresponding types; and then the structural and value histograms corresponding to the tags in the path are multiplied. If a structural histogram is not available for a given tag, uniform-distribution is assumed for that tag. Note that value-based *joins* across different paths are also supported. As an example, consider the following query asking for all Reviews of Shows made before 1992, on data corresponding to the schema in Figure 1:

```
Query 1:    //SHOW[YEAR < "1992"]/REVIEW
```

Here, the mapping of element names to types is straightforward, and in order to compute the

[3]In StatiX summaries, intervals are left-closed and right-open.

$Card_{Year} = \sigma_{<1992}$ (Year)	1.5
Key_{Show} = distribute $Card_{Year}$ into id range of Show	[1-6: 1.5)
$Card_{Review}$ = freq ($parentHist$(Review) $\bowtie$ Key_{Show})	≈ 5

Table 1: Cardinality Computation in StatiX

query cardinality, we perform the computations outlined in Table 1. From the third row, we conclude that the cardinality of the query (that is, the number of Reviews) is approximately 5.

Tuning the Accuracy of StatiX Summaries. The accuracy of the StatiX summaries can be tuned by: (i) increasing/decreasing the number of buckets in the histograms; and/or by (ii) adjusting the *granularity* of the statistics collection. The latter option is feasible due to the type-based statistics gathering of StatiX. Although the types defined in an XML Schema do not appear in the document, they can be used during validation as annotations to document nodes. Thus, by modifying the type-structure of an XML Schema without altering the tag-structure, it is possible to generate many equivalent schemas that validate the same set of documents [3]. Armed with these transformations, we can gather different granularity statistics (coarser or finer) as required by a given application.

3 Issues in Updating StatiX Summaries

The previous section highlighted the main components of a StatiX summary, namely, (i) the schema, and (ii) structural and value histograms. We now discuss the steps required to incrementally update StatiX summaries.

Given an update query, it is important to know both *how many* updates will be applied and also *where* they will be applied. The importance of knowing the location of the insertions stems from the fact that structural histograms capture the relative distribution of children with respect to their parents. Hence, if the correct ids of the inserted components can be computed, the appropriate buckets of the histogram can be updated. In the case of XML updates there is always an implicit *location* component to the update. For example, consider the following insertion (using the syntax of [12]):

```
update
insert <REVIEW>
         <RATING>Top drawer stuff!</RATING>
       </REVIEW>
into  //SHOW[TITLE="The sixth sense"]
```

Here, the path expression: `//SHOW[TITLE="The sixth sense"]` describes the particular Show at which the update applies. Inherently, there is an ordinal associated with this `SHOW`, which is critical in updating the summary. Moreover, the ordinal of `SHOW` determines the ordinal of the other elements in the update fragment. For example, for the above update query, in the parent histogram of Review, the count of the bucket which contains the Show id of "The sixth sense" needs to be incremented; and based on where the review is added, the parent histogram of Rating also needs to be updated. Note that if titles are unique, there is a *single* location in the document which is updated with the given `REVIEW` fragment. However, an update can also be applied to a

set of locations. For example, the following query inserts a new `AGE` sub-element into all movies and TV shows made prior to 1930:

```
update
insert <AGE> Golden Oldie ! </AGE>
into  //SHOW[YEAR < "1930"]
```

Location and cardinality estimation. It is possible to rely on the actual update operation to determine the number and location of updates—the database can provide this information to the estimator module. Recall, however, that the accuracy of estimation and the conciseness of summaries achieved by StatiX are largely due to *contiguous* node ids which both capture the order among elements and are effectively summarized by histograms. While such a numbering scheme is effective for StatiX, it may not be suitable for the backend database—using a contiguous node id scheme at the backend could lead to unacceptable update performance, since it may require a large number of elements to be renumbered [6, 22, 25]. Therefore, instead of relying on a translation mechanism between the contiguous node id scheme required by StatiX and the many possible id schemes at the backend, we make update maintenance self-sufficient by estimating both the *cardinality* and *location* of the updates.

Updates to structure and value histograms. Another important difference to note in the case of updating StatiX summaries is the nature of the histograms being updated. Previously proposed techniques for histogram maintenance (*e.g.,* [11]) were designed for *value histograms*, not structural histograms. There are important differences between a structural histogram and a value histogram. First, there is no sanctity to the values in a structural histogram—structural histograms are based on node ids, but the specific value of the node id is not relevant as long as the histogram correctly captures the parent-child distribution. For example, it does not make a difference whether a sequence of Shows is numbered from 1 to 10 or from 100 to 110, as long as the parent histograms of its children use the same values. Second, the term "insertion" in the case of value histograms and structural histograms take on different meanings. In the case of insertion into a value histogram, the count of the corresponding value is updated. However, in the case of structural histograms, a "new" value is inserted and the subsequent values renumbered. For example, if a new `SHOW` is inserted between `SHOW` 2 and `SHOW` 3, the id of the new `SHOW` is set to 3, and the ids of the subsequent shows are incremented. Thus, the *domain of the values* in a structural histogram continuously changes, and this change in ordinals affects the bucket boundaries of all the parent histograms for the children of type Show.

4 The IMAX Algorithm

In this section we introduce our techniques for maintaining statistics in an XML document in the presence of insertions and deletions of tree fragments. We restrict our attention to the class of updates where the location of the update is determined through branching path expressions in the query. The general format of such a branching path expression is $/t_1[b_1]/t_2[b_2]/.../t_n[b_n]$, where t_i is the tag and b_i is a path expression which may contain value and structural predicates. In the sequel, we use T_i to denote the *type* corresponding to the tag t_i.

A high-level description of IMAX is provided in Algorithm 1. It consists of three main steps: location estimation; id estimation; and summary update. These steps are described in detail in the remainder of this section.

Algorithm 1 IMAX Algorithm

1: **Input:** *Summary* $\mathcal{S}$, *Update* $\mathcal{U} = (c, u)$
   {$\mathcal{S}$ is the initial summary; $\mathcal{U}$ is divided into condition c, and u, the update fragment}
2: **Estimate the location of update using** c **and** $\mathcal{S}$
3: **Estimate the ids of update fragment** u **using** $\mathcal{S}$
4: **Update** $\mathcal{S}$

4.1 Estimating the Location of the Update

Given the branching path predicate for the update location, IMAX needs to estimate the cardinality of these updates, as well as the ids of the nodes where the update takes place. Estimating the location of the update is closely tied to the cardinality estimation. Initially, each type can be thought of as having a trivial one-bucket *key histogram* whose end points are the range of ids of the type, and whose frequency is the cardinality of the type. As we explain below, we utilize this key histogram and the parent histogram associated with each type to perform cardinality and location estimates. A high-level description of the procedure is shown in Algorithm 2.

Algorithm 2 Location and Cardinality Estimation for the Update

1: **Input:** c, $\mathcal{H}$
   {c is the path expression identifying the location}
   {$\mathcal{H}$ is the set of histograms (value and structure) for all types corresponding to the elements in c}
2: let $c = /t_1[b_1]/t_2[b_2]/t_3[b_3]/.../t_n[b_n]$
   {t_i is the tag (correspondingly, its type is T_i)}
3: **for all** $i \in 1$ to n **do**
4: B_i = result distribution of b_i
5: $J_i = B_i \bowtie keyHist(T_i)$
6: $keyHist(T_i)$ = key distribution of T_i based on J_i
7: $parentHist(T_i)$ = compute distribution based on $keyHist(T_i)$
8: **end for**
9: **for all** $i \in 1$ to $n-1$ **do**
10: $J_i = keyHist(T_i) \bowtie parentHist(T_{i+1})$
11: $keyHist(T_{i+1})$ = distribute $freq(J_i)$ into $keyHist(T_{i+1})$
12: **end for**
   {Cardinality of the update}
13: $card$ = frequency (J_n)
   {We now compute the location ids}
14: *locations* = randomly choose *card* ids from the buckets of $keyHist(T_n)$ in proportion to their frequency

Algorithm 3 Estimating Ids

1: **Input:** $parentHist_{child}$, id_{parent}
2: **Output:** id_{child}
{The parent histogram of the child element and the id of the parent are the inputs}
3: $id_{child} = 0$
4: $B_k \in parentHist_{child}$ such that $id_{parent} \in B_k$
5: **for all** $i \in 1$ to $k-1$ **do**
6: $id_{child} +=$ frequency of B_i
7: **end for**
8: $id_{child} += \lfloor freq(B_k)/range(B_k) * (id_{parent} - lowerbound(B_k)) \rfloor$

This procedure operates in three stages: (i) compute the key distribution and parent-key distribution for each of the t_is in the presence of predicates *individually* (lines 3 through 8); (ii) use these individual distributions to compute the overall key distribution of the complete query (lines 9 through 12); and finally (iii) estimate the cardinality and the location of the updates (lines 13,14).

There are three basic operations – histogram multiplication (lines 5 and 10), finding the key distribution (line 6), and finding the parent key distribution (line 7). Histogram multiplication is a well-known operation to find the join estimate given two histograms. Below, we describe the other two operations in more detail.

Key distribution. Note that when two histograms are multiplied, one of the histograms is the key histogram having values which occur exactly once. However, the join distribution gives the total number of tuples in the result – that is, the values in the key histogram may occur multiple times in the result. From this join histogram, we need to determine which distribution of keys occurs in the join (line 6). The fact that keys are unique can be used to compute this distribution as follows: (i) initially, construct a key distribution histogram K by dividing the key histogram into the same number of buckets as the join histogram and in which the frequency of each bucket is the same as its range, (ii) for corresponding buckets j_i in the join histogram and k_i in the key distribution histogram, if frequency of j_i is less than the frequency of k_i, change the frequency of k_i to that of j_i. The resulting histogram is the statistically determined distribution of keys in the join. This histogram is used to compute the parent key distribution described next.

Parent key distribution. An important observation in the case of structural histograms is that the node ids (keys) and parent ids have a strong correspondence with each other – that is, if $nodeid_1 > nodeid_2$, then $parentid(nodeid_1) >= parentid(nodeid_2)$. The parent histogram is a summarization of this correspondence, as illustrated in Figure 2.

As another example, consider the case where the parent histogram of Review (with respect to Show) is `[1-4: 10; 4-6: 6]`. The multi-bucket key histogram of Review would then be `[1-11: 10; 11-17: 6]`. Conversely, suppose Review has now been "filtered" through a value predicate (say, Reviews with Rating > 6) leading to the following key histogram for Review: `[1-11: 5; 11-17: 3]`. The corresponding distribution in the parent histogram of Review is now: `[1-4: 5; 4-6: 3]`. The new parent histogram is then used to compute the cardinality and join distribution of the result (lines 9 to 12).

Choosing the ids. By performing the steps in Algorithm 2, we get the key distribution of the

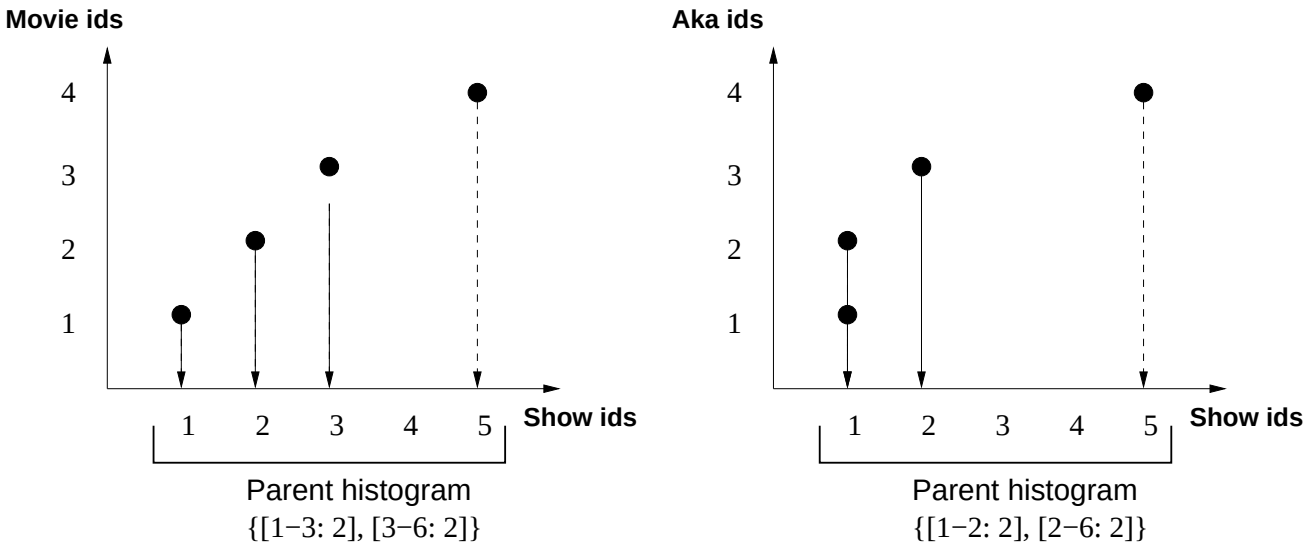

Figure 2: Node and Parent ids have a Correspondence

result of the query (that is, t_n, shown in line 2). Computing the actual location ids is now a matter of choosing the ids from this key histogram. The ids are chosen from the buckets of the key histogram *in proportion to their counts*. For example, suppose the final key distribution of Review is: `[1-11: 5; 11-17: 3]`. We randomly choose 5 Review ids from 1 to 11 and 3 ids from 11 to 17—these choices comprise the statistically determined Review ids where the updates take place.

4.2 Estimating the Ids of the Update Fragment

Once the location of the update is determined, we next need to estimate the ids of the elements in the update fragment. In the case of insertions, the update fragment is explicitly given in the query. The *number* of elements being inserted is known, while the *ids* of these elements have to be estimated. But, in the case of deletions, only the root of the subtree to be deleted is given, so the number as well as the ids of the deleted elements in the subtree need to be estimated.

In order to estimate the ids of the update fragment, we use the parent histogram which summarizes the correspondence between parent and child ids (Figure 2).

Estimating ids for insertions. Algorithm 3 describes how the parent histogram is used to estimate the id of a child fragment. The algorithm outputs a single child id. If there are multiple children in the update with the same tag, then a set of *contiguous ids* are assigned beginning from the estimated id of the first child (as determined by Algorithm 3).

Estimating ids for deletions. In the case of deletions, only the root node of the subtree to be deleted is given. The elements in this subtree have to be first determined from the schema. Since the id of the root node of the deletion is known, Algorithm 3 can be used to estimate the id of the child. In addition, the frequency of the child can be estimated from B_k (line 4 in Algorithm 3) through dividing the frequency of B_k by the range of B_k.

4.3 Updating the Summary

The relevant parent histograms in the summary need to be updated by either inserting new ids or deleting them. This includes not only the parent histograms of the types in the update fragment, but also the children of these types which may not be present in the update fragment. However, a large number of insertions or deletions to the histogram may make it inaccurate. For example, if new documents are appended continuously, then clearly, only the last bucket of a histogram is updated each time with new ids. Therefore, while the last bucket keeps accumulating counts eventually making it inaccurate, the remaining buckets retain their original counts. One strategy to approximately maintain the equi-depth histogram is to periodically *redistribute* the bucket counts by splitting a bucket once its count reaches a threshold occupancy T into two new buckets and then simultaneously merging a pair of buckets whose combined count is less than T [11]. If more than one such pair exists, then the pair whose combined frequency is the least is chosen. If such a pair of *split-merge* operations cannot be performed, then either the histogram is recomputed from the base data, or, only the split is performed and the number of buckets is increased. We utilize the former approach since our aim is to work within the originally allocated memory budget.

Algorithm 4 highlights the main steps in inserting a new value into a parent histogram. The input to the algorithm is the pair (id, f). Note that the id in this case is the id of the *parent*, while the histogram being updated is the parent histogram of the *child*. The pair (id, f) indicates the number of times, f, the given child occurs under the given parent with id id. Steps 4 to 8 perform a *shift* operation to indicate the insertion of a new id – this is equivalent to renumbering the previous ordinals of the elements due to the insertion of a new one. Steps 9 to 16 determine whether only a *reorganization* will suffice or whether a complete *recomputation* of the inaccurate histogram needs to be computed from the base data.

For deletions, instead of ids being "inserted", the ids need to be deleted. Similar issues also arise for deletions – that is, a single bucket may have a very small count compared to the others, affecting its equi-depthness. The strategies outlined for insertions can be easily modified to handle deletions as well.

4.4 Improved Location Estimation

A potential limitation in the current location estimation process (Section 4.1), is the use of single dimensional histograms for values. The problem stems from the fact that *no correspondence* between the occurrence of a value and the id of the node at which it occurs is stored as in the case of structural histograms. Consequently, we have to make the independence assumption when computing the distribution of the nodes containing particular values – that is, distribute the estimated cardinality into the parent histogram in proportion to the bucket counts.

In order to overcome this limitation, we propose the use of 2D histograms to explicitly capture the correspondence between values and the corresponding node ids. Note that since 2D histograms require more space, the budget for value histograms must be increased to improve the accuracy. However, as we show in Section 5, the advantages of using 2D histograms are substantial. Moreover, the use of these histograms benefits StatiX as well: since it removes the independence assumption, higher accuracy can be obtained for queries that involve value-based

Algorithm 4 Insertion of a new id into a parent histogram

1: **Input:** $Histogram : H$, $Update : (id, f)$, $Threshold : T$
   {H is the histogram to be updated}
   {(id, f) is the update consisting of new (id, frequency) pair}
   {T is threshold occupancy at which a bucket is split}
2: $B_k \in H$ such that $id \in B_k$
   {Update the frequency of the bucket}
3: $B_k.frequency += f$
   {Update bucket's upper limit to reflect *insertion* of new id}
4: $B_k.hi = B_k.hi + 1$
   {n is the number of buckets in H}
   {Update the boundaries of remaining buckets}
5: **for all** $i \in k+1$ to n **do**
6: $B_i.lo = B_i.lo + 1$
7: $B_i.hi = B_i.hi + 1$
8: **end for**
9: **if** $B_k.frequency >= T$ **then**
10: found = find B_i, B_{i+1} in H such that $B_i.frequency + B_{i+1}.frequency < T$
11: **if** found **then**
12: **REORGANIZE** H merging B_i, B_{i+1} and splitting B_k
13: **else**
14: **RECOMPUTE** H from base data
15: **end if**
16: **end if**

predicates.

We use the algorithm proposed in [15] to build equi-depth 2D histograms – choosing one axis at a time to build the histogram. We choose the *key dimension* as the first dimension – the key dimension is *contiguous* and hence will lead to histogram buckets which are well packed in that dimension. A split-merge strategy with a threshold T is used to maintain the 2D histograms as well. Splits are restricted to occur only along the values axis and merge pairs are chosen along the same axis – that is, whenever new buckets are created by splitting or merging, their boundaries along the key axis always match. Figure 3 (a) shows a 2D histogram with the key axis being chosen first during construction. Two buckets in the figure are selected for merging (darkened rectangles) – both buckets have the same boundaries on the key axis. Figure 3(b) shows the histogram after the buckets have been merged.

5 Experimental Evaluation

We have carried out a detailed evaluation of the IMAX approach on synthetically generated IMDB data and also on a subset of DBLP data available from [7]. All experiments were performed on a Compaq ES45 dual-processor machine with 1.25 GHz and 16 GB memory. For ease of presentation, we classify the types of insertions into: (i) Append only, and (ii) Random insertions.

Memory Budget. The memory budget for the summary depends on the number of types in

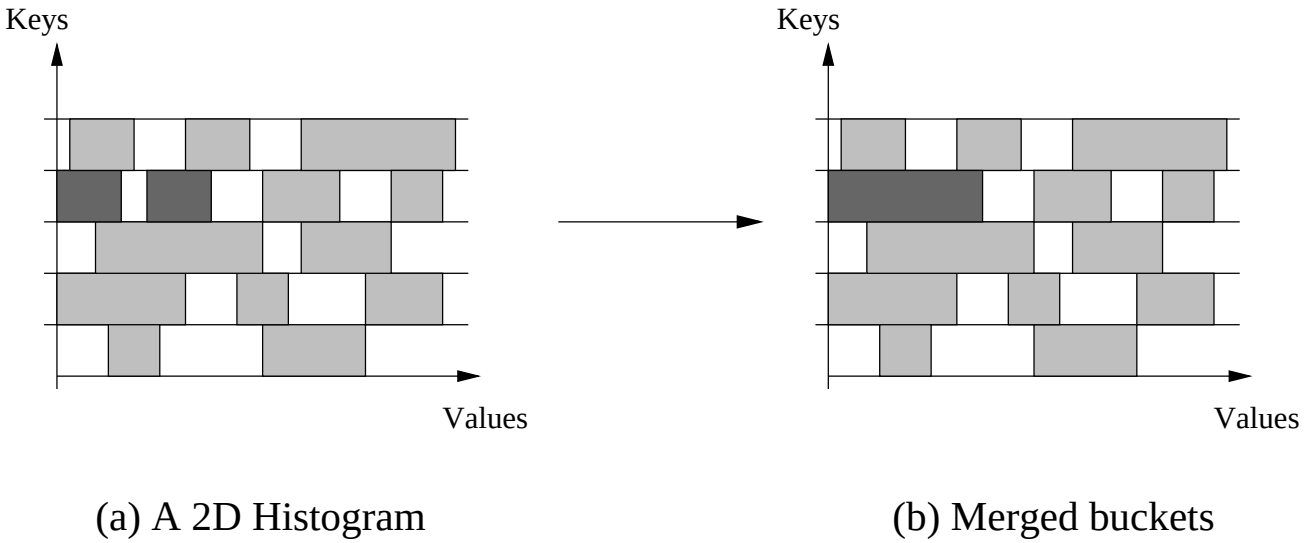

Figure 3: 2D Histograms - Construction and Merge

the schema and the number of buckets allocated for structural histograms and value histograms. All experiments in this section were performed with a minimum of 5 buckets for each structural histogram and 100 buckets for each value histogram – translating to about 5KB of memory, and a maximum of 30 structural histogram buckets and 500 value histogram buckets – translating to about 23 KB of memory.

Threshold Factor. The reorganization threshold of histogram H_i is set as $T_i = t * f_i$ where f_i is the equi-depth bucket frequency of histogram H_i, and t is a user-specified *threshold factor*. In our experiments, the threshold factor was set to 2.5.

Metrics. Our primary performance metric is to compare how close the IMAX incrementally-generated summary is with respect to the computed-from-scratch StatiX summary. For each affected histogram, this is quantitatively captured by μ_{mse} defined as follows:

$$\mu_{mse}(IMAX) = \frac{\sum_{i=1}^{N}(Est_{StatiX} - Est_{IMAX})^2}{totalCardinality}$$

where $i = 1$ to N covers the total range of values in the histogram, Est_{StatiX} is the estimate of value i from the histogram computed by StatiX, and Est_{IMAX} is the estimate computed from IMAX. The *totalCardinality* refers to the overall occupancy of the histogram.

To quantitatively establish that there is indeed a significant difference between the updated document and the original document, we also compute μ_{mse} between the currently computed-from-scratch summary and the original summary (*i.e.,* before any updates were received), as shown below:

$$\mu_{mse}(ORIGINAL) = \frac{\sum_{i=1}^{N}(Est_{StatiX} - Est_{ORIGINAL})^2}{totalCardinality}$$

While the above metrics measure the *accuracy* of IMAX in the face of significant updates, our next metric aims to measure its *efficiency*. This is done by tracking the number of recomputations-from-scratch incurred by IMAX during its maintenance process. This metric,

called $RECOMP$, is defined as the number of recomputations divided by the total number of insertions into the histograms, that is, $RECOMP = \frac{r}{I}$ where r is the number of recomputations and I is the total number of histogram insertions. $RECOMP$ can be calculated on a per-type basis or over all types in the insertions.

In addition, we also present the *cardinality estimation accuracy* for both IMAX and StatiX by computing the *relative error* for a query workload. We tabulate the average time per update for IMAX and the recompute-from-scratch StatiX and discuss these results in Section 5.3.

5.1 Append-Only Updates

Append-only updates occur in warehouse scenarios, where new documents are continuously being added. The main complexity in append-only updates is in the reorganization of the histograms since appends occur at the root of the document. We performed experiments for both DBLP and IMDB. For DBLP, several ARTICLEs were appended to the existing document, while in the case of IMDB, new Shows were appended.

Results. The μ_{mse} values for two types: Review and Aka are shown in Figure 4 for the IMDB database. Note that the histograms correspond to the *parent histograms* of these types[4]. In this graph, the number associated with each algorithm in the legend (10 in Review(10,IMAX)) refers to the number of structural histogram buckets. Note that the number of value histogram buckets is not an issue here, since the location condition does not involve a value predicate.

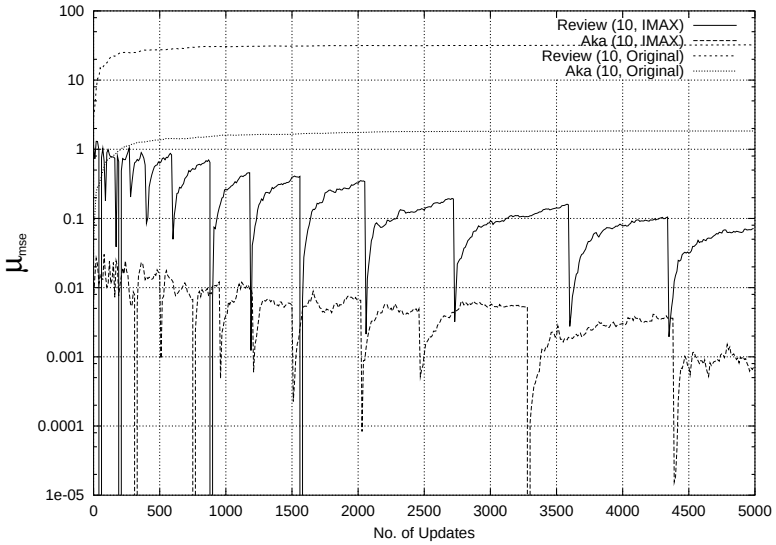

Figure 4: **IMDB**: μ_{mse} values for types Review and Aka with $t = 2.5$

The first point to note in Figure 4 is that the μ_{mse} values (which are shown on a *log-scale*) for IMAX are very low, especially when compared with the μ_{mse} values for the original histogram—in fact, there is close to *two orders of magnitude difference* in their quality. This clearly indicates that (a) there is a substantial change between the original document and the updated document, and (b) IMAX is able to track these changes rather well.

[4]We use the phrase "histogram of type x" to mean the parent histogram of that type in the rest of the section.

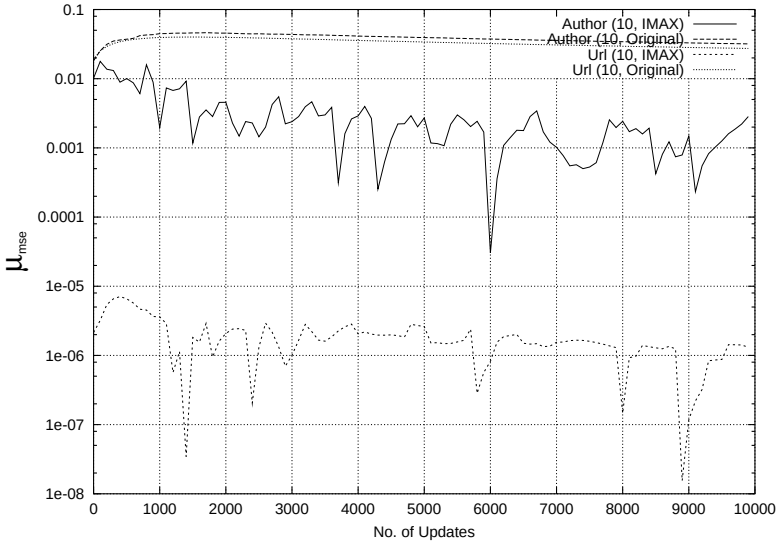

Figure 5: **DBLP**: μ_{mse} values for types Author and Url with $t = 2.5$

Next, the efficiency aspect is captured in the RECOMP numbers shown in Table 2, which show that only a very small fraction of recomputations are required to support the IMAX incremental maintenance strategy.

Type	**No. of Insertions**	RECOMP, t = 2.5
Show	5000	0
Review	170123	0.008%
Aka	9798	0.12%
Tv	2461	0.28%
Movie	2539	0.27%
Year	5000	0.02%
TOTAL	189921	0.01%

Table 2: **IMDB**: RECOMP with Appends

Similar results for the DBLP dataset are shown in Figure 5 and Table 3 for the μ_{mse} and $RECOMP$ metrics, respectively. Note that in Table 3, only a subset of types updated have been enumerated, while the last line totals up all updated types.

5.2 Random Insertions

Turning our attention to random insertions, the most important component here is the location estimation. If a single update query results in updates in multiple locations, then the cardinality estimation also comes into play. We divided insertions into two categories: (i) Unique insertions, where a single update query results in insertion at a unique location in the document, and (ii) Multiple insertions, where a single update query results in insertions at multiple locations in the document.

For IMDB, we generated an Actor database consisting of information about actors. Each `ACTOR`

Type	No. of Insertions	RECOMP, t = 2.5
ARTICLE	10000	0
AUTHOR	16174	0.04%
URL	9989	0.08%
TOTAL	109359	0.06%

Table 3: **DBLP**: RECOMP with Appends

subtree consists of a `NAME` sub-element, and multiple `PLAYED` sub-elements. Each `PLAYED` element may contain multiple `EPISODE` sub-elements. We chose updates which reflected the addition of new information regarding the actor's acting history where new shows in which he/she had acted in were inserted into the database. The insertions were of the form:

```
update insert
  <PLAYED>
     <EPISODE>...</>
     <EPISODE>...</>
     ....
  </>
where /ACTOR[NAME="x"]
```

The number of Actors in the database was 1000 – that is 1000 unique values for the value predicate involving Name. Note that this query has *multiple* levels of insertions where the estimated id of Actor (from Algorithm 2) is used not only to update the parent histogram of Played, but also to estimate the id of Played (from Algorithm 3). This id in turn is used to determine the ids of the multiple Episodes.

For the DBLP dataset, we chose a set of journal articles from 134 different journals. Each journal had articles published in that journal in a separate subtree. The insertions we chose reflects the addition of new articles into a database segregated on the basis of journal names. Each ARTICLE had multiple AUTHOR elements along with several other relevant information such as URL, PUBLISHER, YEAR, etc.

Additional Measures. Apart from the μ_{mse} and $RECOMP$ metrics defined earlier, we utilize two additional supporting measures here to help explain the results:

Location Estimation Accuracy: This metric measures the effectiveness of the location estimation technique. It compares the *estimated* location against the *actual* location. The location estimation is deemed to be correct if both the estimated as well as the actual location both fall into the same histogram bucket. The location estimation accuracy is defined to be: $LEA = \frac{L_{correct}}{L_{total}}$ where $L_{correct}$ is the number of correctly estimated locations and L_{total} is the total number of locations.

μ_{count}: μ_{count} considers each histogram bucket and computes the *deviation* of the frequency of the bucket from the actual frequency normalized to the average bucket count. This metric helps in highlighting where the incorrect location estimations are being distributed.

The metric [11] is defined as:

$$\mu_{count} = \frac{\beta}{N}\sqrt{\frac{1}{\beta}\sum_{i=1}^{\beta}(f_{B_i} - B_i.count)^2}$$

where N denotes the number of values, β denotes the number of buckets, f_{B_i} denotes the actual count of bucket B_i and $B_i.count$ denotes the current count of bucket B_i.

Results. The location estimation accuracy for the IMDB and the DBLP datasets under random insertions are shown in Figures 6 and 7, respectively, as a function of the number of value histogram buckets. Each graph shows the location estimation accuracy in two cases: (i) when the structural histogram contains only 5 buckets and, (ii) when it contains contains 30 buckets. Further, both the 1D and 2D versions of IMAX are presented in the graphs and we see that using 2D histograms clearly gives superior estimation accuracy as compared to using 1D histograms. Note that in order to compare *only* the location estimations, 2D histograms were used for cardinality estimation in both cases. The equivalent cardinality estimation for the 1D case would contain only the square root of the number of buckets in the value histogram. This is the tradeoff between the space utilized and the accuracy.

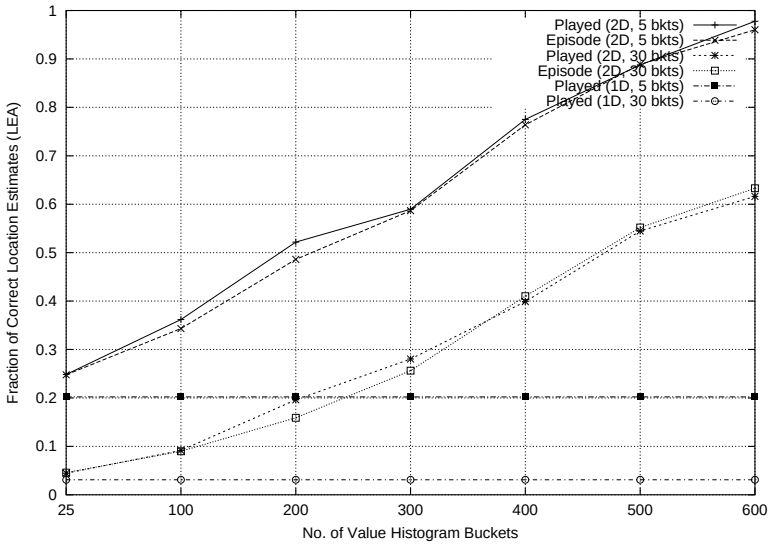

Figure 6: **IMDB**: *LEA* for Random Insertions with 1D and 2D Value Histograms

The μ_{mse} metric for the type Played is shown in Figure 8 for both the original summary, as well as with the 1D and 2D versions of IMAX. Note that, again, there is over *two orders of magnitude* difference in accuracy between the original summary and both versions of IMAX.

An interesting observation in Figure 8 is that the 2D version of IMAX provides only marginal accuracy gains over the 1D version. This is in spite of the fact that the 2D version is far superior in terms of location estimation as compared to the 1D version as shown in Figure 6. The reason is that the insertions are approximately *uniformly distributed* over the whole document. So, what may not be the correctly estimated location for one insert may very well turn out to be the correct location for some other insert, effectively *canceling out* the effect of several wrong estimations. This is clear from Figure 9 which plots the μ_{count} metric for Played. The μ_{count} values of both the 1D and 2D cases are close together here.

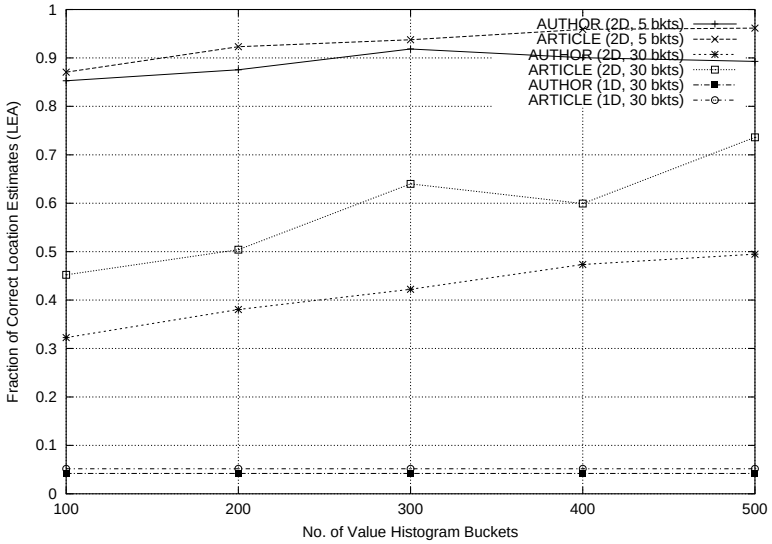

Figure 7: **DBLP**: *LEA* for Random Insertions with 1D and 2D Value Histograms

However, if we consider insertions where the locations of the insertions are *skewed*, the benefits of using 2D histograms become immediately apparent. Such insertions are possible when, say, more recently added actors need to be updated more frequently than others. Figure 10 shows the μ_{count} for such skewed insertions, and we observe nearly an order of magnitude difference in the μ_{count} values of the 1D and 2D versions. This demonstrates the benefits of using 2D histograms. The μ_{mse} metric for the 2D version showed significant improvement over the 1D version as shown in Figure 11. Similar behaviour was observed in the DBLP data as well.

Moving on to the efficiency aspect of IMAX under random insertions, the number of recomputations for both DBLP and IMDB, with and without skewed insertions, are shown in Tables 4 and 5, respectively. (The tables provide the specific measures for only a subset of the types, but the totals in the last line are across all types.)

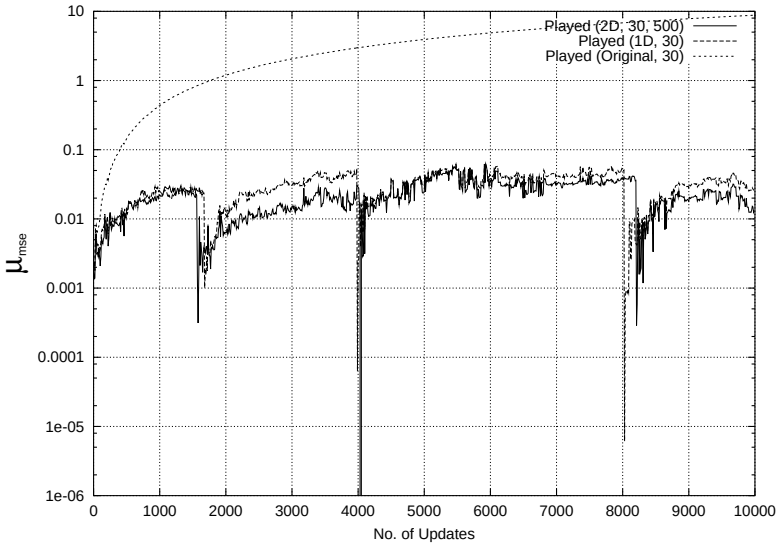

Figure 8: **IMDB**: μ_{mse} values for type Played for Random Insertions

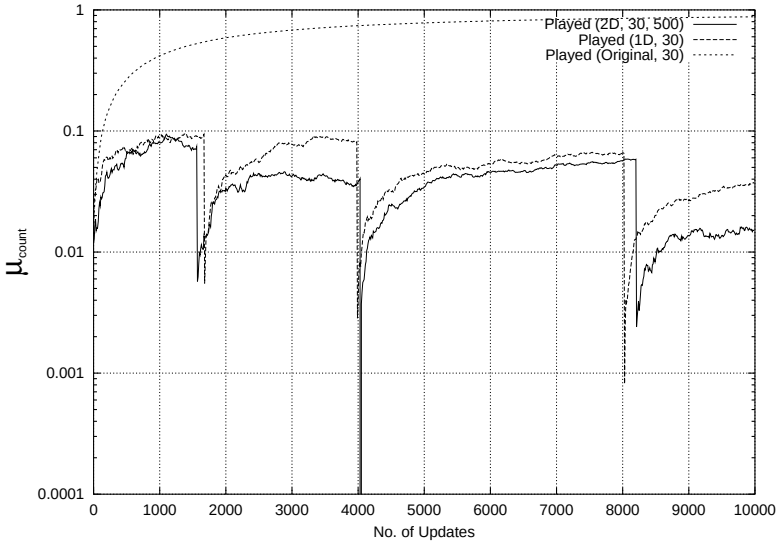

Figure 9: **IMDB**: μ_{count} values for type Played for Random Insertions

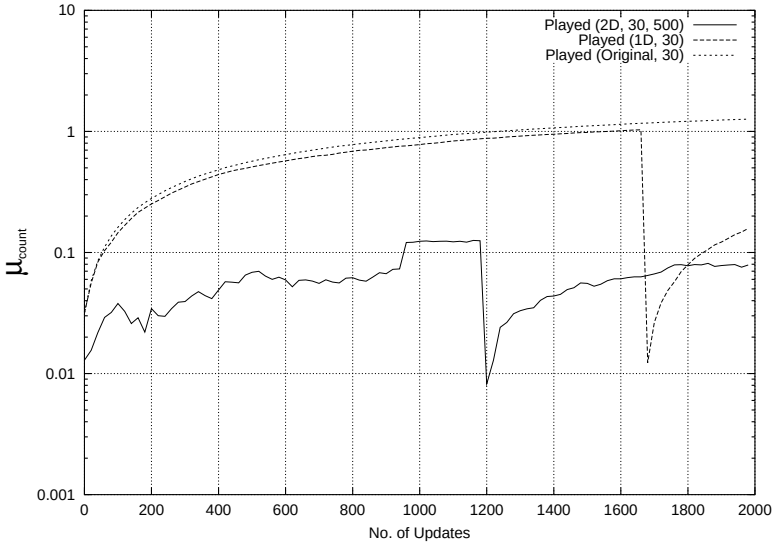

Figure 10: **IMDB**: μ_{count} values for type Played for Skewed Insertions

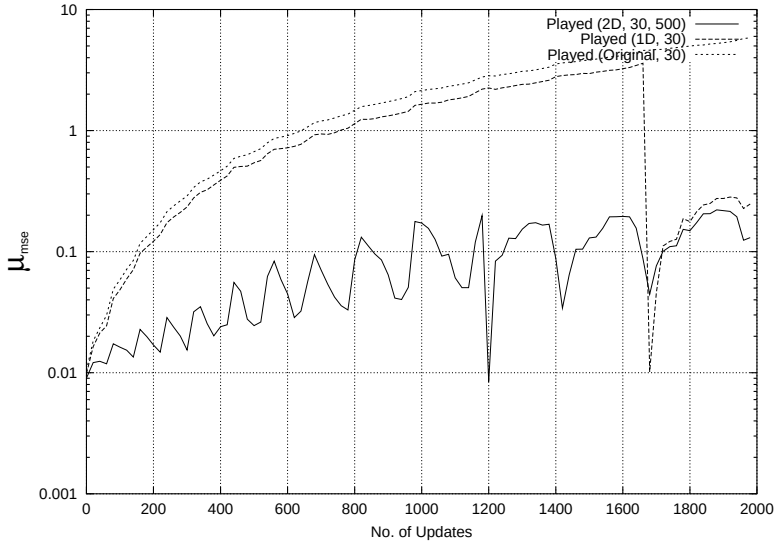

Figure 11: **IMDB**: μ_{mse} values for type Played for Skewed Insertions

Clearly, the number of recomputations required is a very small fraction of the total number of insertions made in the document. Note that the number of recomputations can be further reduced by increasing the reorganization threshold—trading off on the accuracy of the histograms.

Type	**No. of Insertions (Random)**	RECOMP, t = 2.5 **(Random)**	**No. of Insertions (Skewed)**	RECOMP, t = 2.5 **(Skewed)**
Played	10000	0.03%	2000	0.05%
Episode	104569	0.006%	20937	0.02%
TOTAL	124569	0.01%	24937	0.02%

Table 4: **IMDB**: RECOMP with Random and Skewed Insertions

Type	**No. of Insertions (Random)**	RECOMP, t = 2.5 **(Random)**	**No. of Insertions (Skewed)**	RECOMP, t = 2.5 **(Skewed)**
ARTICLE	8000	0.02%	2000	0.05%
AUTHOR	14624	0.1%	3843	0.28%
TOTAL	88414	0.14%	22606	0.36%

Table 5: **DBLP**: RECOMP with Random and Skewed Insertions

Multiple Insertions. We consider here single update queries which spawn multiple insertions. For example, adding a comment "Arnold Rocks" for all films starring Arnold Schwarzenegger, or adding information templates for all shows satisfying certain criteria. We experimented with multiple-insertion updates on the article database of DBLP. The update involved adding a LINK for a given author denoting his/her URL. Such an update would require multiple insertions of the tag `link` depending on the number of articles authored by the author since the tag should

be added to each such occurrence. The DBLP document contained a total of 1165 authors, each with at least 10 articles spread over more than 17000 articles. We performed insertions of the following form:

```
update insert  <link> .. </> into
  /dblp/article[author="x"]
```

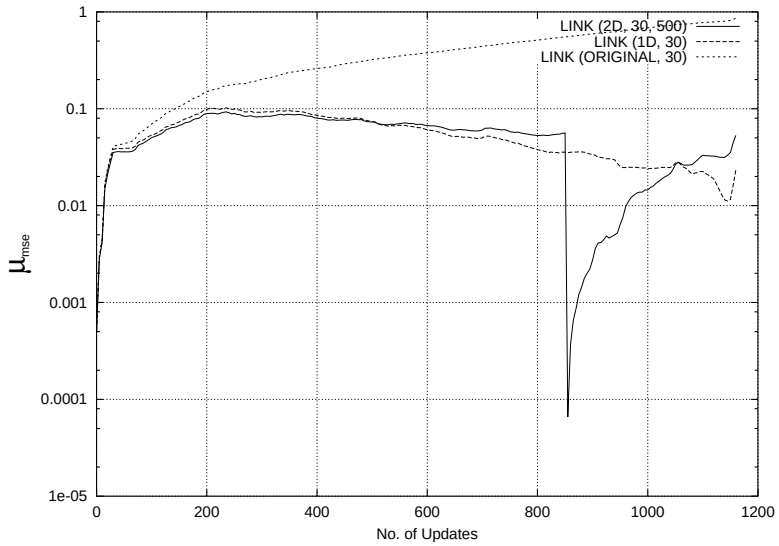

Figure 12: **DBLP**: μ_{mse} values for type LINK for Randomly Distributed Multiple Insertions

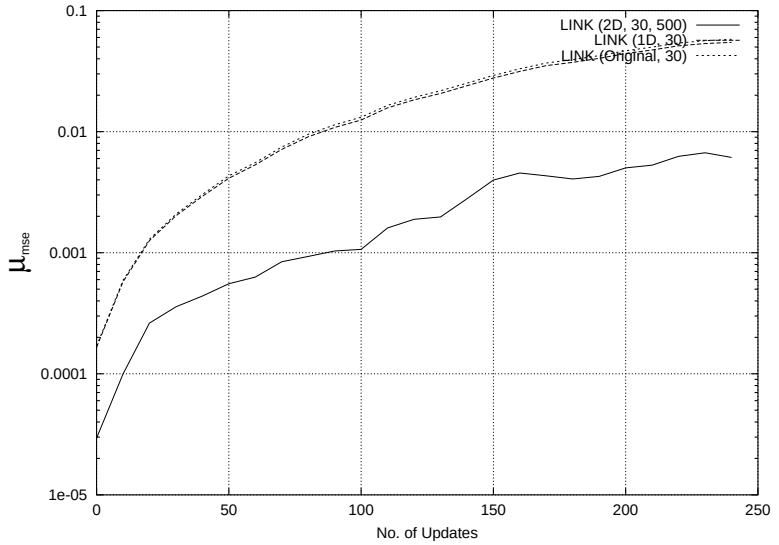

Figure 13: **DBLP**: μ_{mse} values for type LINK for Skewed Multiple Insertions

As with the unique insertions, two sets of insertions were performed: a set of skewed insertions with around 20% of authors; and another set of insertions involving all authors. We present the μ_{mse} metric for both cases in Figures 12 and 13, respectively. The utility of 2D histograms

is limited in the case of uniformly distributed insertions, but provides considerable advantage when the insertions are skewed. No recomputations were required in the case of skewed insertions, while a single recomputation was performed when 2D histograms were used in the case of random insertions.

5.3 Estimation Accuracy and Timing

The previous sub-sections dealt with the histogram accuracy (for a subset of histograms) and the number of recomputations required for various datasets. The results indicated that IMAX is very accurate when it comes to tracking the updates with a very small number of recomputations. In order to get a "global" picture of the accuracy and efficiency of IMAX, we briefly present numbers on the estimation accuracy and timing.

	Append (IMDB)	**Insert (IMDB)**	**Multiple Inserts (DBLP)**
IMAX	97	77	190
StatiX	8167	1437	5403

Table 6: Average Time per Update (in ms)

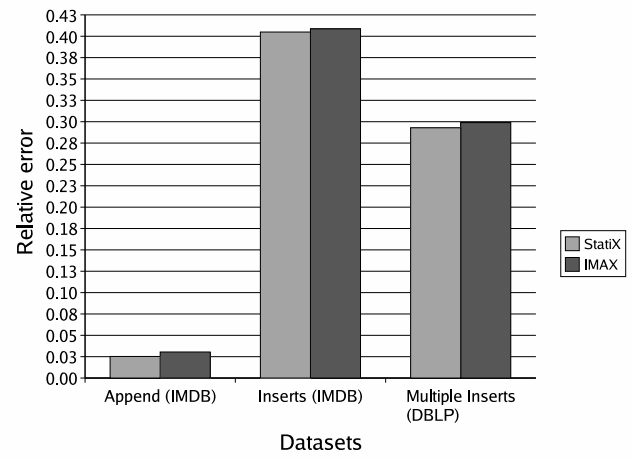

Figure 14: Relative Error for IMDB and DBLP Datasets

Table 6 tabulates the *average time* per update for the different datasets. We see here that IMAX is at least an *order of magnitude faster* than the recompute-from-scratch StatiX even when the occasional recomputations required are taken into account.

We then generated a query workload of around 300 queries with both branching path expressions without value predicates (around 15% of the workload) as well as path expressions with at least one and a maximum of two value predicates for each of the datasets. For each query workload, we computed the relative error in estimation using the IMAX summary as well as the recomputed-from-scratch StatiX summary. These results are shown in Figure 14, and indicate that the quality of the IMAX summary is almost as good as that of the StatiX summary.

6 Related Work

The problem of updating XML documents has only recently started to attract attention. Proposals for update languages have appeared in the literature [12, 22] as well as in implementations of XQuery engines (*e.g.,* Galax [10]). Further, the problem of incrementally validating updated documents has been studied in [2, 4, 18]. There has also been work on efficient labeling techniques for XML document nodes which are subjected to updates [6, 25].

Several approaches have been proposed in the literature for summarizing XML documents and estimating query cardinality [1, 5, 9, 13, 20, 21, 26]. They differ in many aspects, notably: whether or not they use schema information; and which queries they support. Whereas [1, 5, 13, 20, 21, 26] deal with schemaless data, [9] uses schema information to both improve the quality as well as reduce the size of summaries. There is a wide variation in the classes of supported queries, *e.g.,* [1] only handles (non-branching) path expressions in the document tree; [5] handles twig queries; and [9] supports a significant subset of XQuery. Common to all these proposals is the lack of support for efficiently (and incrementally) updating the statistical summaries.

In [24], a new data structure – the "bloom histogram" – is proposed to maintain *simple path expressions* in the presence of updates. Our work differs from [24] in two ways: i) we use the schema in order to build and maintain the summary, and ii) our summary can handle the estimation of branching path expressions, whereas [24] is limited to simple path expresssions.

Incremental maintenance of statistics has been addressed in the context of relational database systems (for example, [8, 16, 11], etc.). The novelty in the XML context is that statistics must be maintained for both *structure* and *values*.

7 Conclusions

We introduced IMAX, a system which extends the schema-based statistics framework of the StatiX approach to incrementally handle updates to XML repositories. The novel challenges in the design of IMAX included developing techniques for accurately estimating both the locations and the sizes of updates, as well as for the maintenance of structural histograms. To accurately estimate the location of updates, we extended the StatiX model with 2D histograms that capture the correspondence between the value of an element and its id. Although these histograms require more space, they are only needed to capture value-key correlations, *i.e.,* they replace the 1D value histograms used previously in StatiX.

Our experiments to evaluate the utility of IMAX covered a variety of updates and datasets, and indicate that the accuracy of estimation from the updated statistics is very close to that obtained from the expensive brute-force option of re-computing the statistics from scratch. Further, these benefits can be obtained quite efficiently, requiring infrequent recomputations of the summaries from the base data.

In closing, IMAX makes sustained and efficient query processing feasible even in real-world XML environments whose contents are dynamically changing, which may become the norm in the coming years.

Acknowledgements. This research was supported in part by a Swarnajayanti Fellowship of Dept. of Science & Technology, Govt. of India.

References

[1] A. Aboulnaga, A. Alameldeen, and J.Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proc. of VLDB*, 2001.

[2] D. Barbosa, A. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *Proc. of ICDE*, 2004.

[3] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of ICDE*, 2002.

[4] B. Bouchou and M. Halfeld-Ferrari-Alvez. Updates and Incremental Validation of XML Documents. In *Proc. of DBPL*, 2003.

[5] Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proc. of ICDE*, 2001.

[6] E. Cohen, H. Kaplan, and T.Milo. Labeling dynamic XML trees. In *Proc. of PODS*, 2002.

[7] DBLP. http://dblp.uni-trier.de/xml.

[8] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proc. of ICDE*, 2000.

[9] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of SIGMOD*, 2002.

[10] Galax. `http://www.galaxquery.org`.

[11] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM TODS*, 27(3), 2002.

[12] P. Lehti. Design and implementation of a data manipulation processor for an XML query language. Master's thesis, Universität Darmstadt, 2001.

[13] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: An on-line self-tuning markov histogream for XML path selectivity estimation. In *Proc. of VLDB*, 2002.

[14] J. McHugh and J. Widom. Query optimization for XML. In *Proc. of VLDB*, 1999.

[15] M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of SIGMOD*, 1988.

[16] S. Muthukrishnan and M. Strauss. Maintenance of multidimensional histograms. In *Proc. of FSTTCS*, 2003.

[17] M. Nicola and J. John. XML parsing: a threat to database performance. In *Proc. of CIKM*, 2003.

[18] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proc. of ICDT*, 2003.

[19] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of SIGMOD*, 1984.

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

[21] N. Polyzotis and M. Garofalakis. Structure and value synopses for XML data graphs. In *Proc. of VLDB*, 2002.

[22] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. of SIGMOD*, 2001.

[23] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation, October 2004.

[24] W. Wang, H. Jiang, H. Lu, and J. Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *Proc. of VLDB*, 2004.
[25] X. Wu, M.-L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proc. of ICDE*, 2004.
[26] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Proc. of EDBT*, 2002.
[27] Xerces java parser 2.5.0. http://xml.apache.org/xerces-j/.