Sketch-based Summarization of Ordered XML Streams

UCSC-SOE-08-14

Veronica Mayorga [#1], Neoklis Polyzotis [#2]

[#] *University of California at Santa Cruz*

[1] vmayorga@soe.ucsc.edu

[2] alkis@soe.ucsc.edu

ABSTRACT

XML streams, such as RSS feeds or complex event streams, are becoming increasingly pervasive as they provide the foundation for a wide range of emerging applications. An important problem in this context is the realization of continuous queries that can support on-line monitoring and analysis of the streaming XML data. The evaluation of exact results, however, can be prohibitively expensive for the resource-restricted environment of a streaming application. This leads naturally to the use of approximation techniques that can provide an on-demand estimate for the result of a continuous XML query.

In this paper, we introduce a new technique for approximately answering a complex aggregate query over an XML stream using limited memory. The main novelty of the proposed technique is that it supports XML queries with any combination of the common XPath axes, namely, ancestor, descendant, parent, child, following, preceding, following-sibling, and preceding-sibling. At the heart of our method lies an efficient transform that reduces a continuous XML query to an equi-join query over relational streams. We detail the transform and discuss its integration with randomized sketches as a basic mechanism to estimate the result of the XML query. We further enhance this mechanism with structural sieving, a technique that takes advantage of the XML data and query characteristics in order to improve the accuracy of the sketch-based approximation. We present an extensive experimental study on real-life and synthetic data sets that validates the effectiveness of our approach and demonstrates its advantages over existing techniques.

1. INTRODUCTION

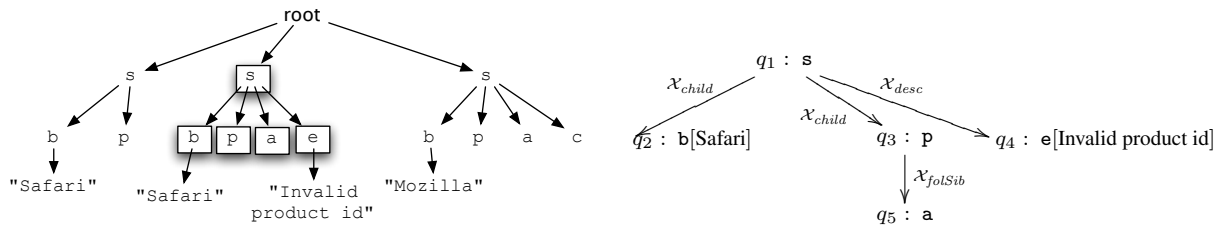
In recent years, the topic of processing streaming XML data has gained significant importance. Emerging applications include the management of complex event streams, such as those generated by the modules of a running workflow, monitoring the messages exchanged by web-services, and publish/subscribe services for RSS feeds. XML is obviously an attractive data model for this type of data, as it allows the application to encode complex stream objects without committing to a fixed schema.

The ability to perform data analytics over streaming XML data can have significant value for large-scale systems. One type of queries that arises naturally in this scenario is the continuous computation of some aggregate (e.g., COUNT(*)) over the appearances of a pattern in the XML stream. This type of queries can help detect anomalies in the content of the stream, or test hypotheses as part of a data mining task. As a concrete example, consider a stream that represents the running workflow of a web-based retailer. Assume that the stream contains information on the sessions of users, where each session is modeled as an XML element. (Hence, the stream acquires more session elements as users visit the online store.) A session element, in turn, records in its children elements the web browser of the user, the products that the user has looked at, the additions of said products to the shopping cart, and requests to checkout. A session may also include an exception object, if an error occurred. Figure 1(a) shows the representation of a sample stream of three sessions as an XML tree. Note that the ordering of elements in the tree is important and reflects the order in which the corresponding events occur in the stream. Now, a system analyst may pose the following query over the stream: "Count the times that a session ends in an error when the browser is Safari and the actions of the user involve a view of a product page followed by an addition to the shopping cart." Essentially, this query tests the hypothesis that the use of a specific browser causes a bug in the handling of the shopping cart when a specific sequence of actions occurs. By examining this running count over the stream, the analyst may be able to verify the likelihood of this correlation.

The potentially large volume of data, the need for scalability, and the desire for short response times impose severe restrictions on the CPU and memory resources that are available for evaluating such continuous queries. This leads naturally to the adoption of approximate query answers as an alternative to the computation of exact results. Under this scheme, the query processor maintains a concise stream synopsis that enables the on-demand approximation of the current query result with reasonable accuracy. We note that this solution fits naturally with the exploratory nature of on-line analytics, where the typical intention is to identify interesting trends and accuracy to the last decimal is not required.

Several recent studies [7, 19, 20, 22, 14] have investigated the problem of XML summarization that underlies the generation of approximate answers. The majority of the proposed techniques, however, require multiple passes over the complete data set. Hence, they cannot be used in a streaming environment, where the synopsis must be constructed in a single pass over the data. Moreover, most summarization studies focus on the unordered XML model, and thus cannot handle continuous queries that impose constraints on the ordering of XML elements. This type of queries can occur frequently in practice, as ordering is a natural property for several types of XML streams, e.g., event streams, or workflow traces.

Our Contributions. In this paper, we introduce a new technique for the problem of approximate query answering over XML streams. Our technique expands significantly the scope of existing studies on XML summarization, as it supports the ordered XML model and tree pattern queries with the major XPath [3] axes, namely, parent, ancestor, child, descendant, following, preceding, following-sibling, and preceding-sibling. The key technical contributions of



(a) XML tree corresponding to S (b) Query Q

Figure 1: Part (a) depicts the tree representation of a stream S corresponding to the simplified execution trace of a web application. The stream contains `session` elements that describe the interactions of users with an online store. A session contains elements that describe the `browser` of the user, the `product`s that the user has viewed, `addition`s to the shopping cart, and `check-out` actions. An `exception` element signals the occurrence of an error, whose type is described in the value content of the element. The corresponding ranked node-labeled tree is shown in Part (b).
Part (b) depicts an example tree-pattern over the execution trace. We use the notation $q : l$ to denote that variable q binds to elements of label l. We extend the notation to $q : l[constant]$ to add an equality condition on element content. The tree-pattern searches for sessions that employ a specific browser and where an addition to the shopping cart causes a specific exception to be raised. For the current contents of S, we can verify that $Q(S)$ contains exactly one binding tuple, formed by the elements enclosed in squares.

our work can be summarized as follows:

• $\mathcal{X}2R$ **Transform.** Our techniques are based on a new transform, termed $\mathcal{X}2R$, that rewrites an aggregate XML query as a join query over a relational database. The key property of $\mathcal{X}2R$ is that the relational query belongs in the well studied class of tree-join queries with equi-join predicates, which means that a rich set of existing relational techniques becomes immediately available for the processing and manipulation of XML queries. We provide a formal definition of the $\mathcal{X}2R$ transform and its properties and discuss its implementation in a streaming XML environment.

• **Sketch-based Summarization for XML Streams.** We couple the aforementioned transform with randomized sketching techniques and propose the first synopsis structure that enables approximate answers with provable guarantees for aggregate tree-pattern queries. We further enhance the basic approximation scheme with two techniques that improve significantly the quality of approximation: (a) structural filtering, that reduces the volume of sketched data by filtering the stream on-the-fly against the constraints of the query, and (b) structural partitioning, that partitions the stream in sequential sub-streams which can be sketched more effectively. We detail these mechanisms and present efficient algorithms for their realization in a streaming environment.

• **Experimental Validation.** We evaluate the proposed technique with an extensive experimental study over real-life and synthetic data sets. The results validate the effectiveness of our technique and demonstrate its advantages over other approaches.

To the best of our knowledge, this is the first work that tackles the challenging problem of approximate query answering for ordered XML streams and continuous aggregate tree-pattern queries that combine different XPath axes.

2. RELATED WORK

Several recent studies [7, 19, 20, 22, 14] have focused on the problem of off-line XML summarization, where the complete data set is available and can be accessed repeatedly. It is interesting to note that certain off-line techniques create one-pass synopses in the process of constructing an XML summary, e.g., the kernel of XSeed [22] or the count-stable graph of TreeSketch [14]. These synopses, however, are either too inaccurate (e.g., the kernel) or too large (e.g., the count-stable graph), and thus it is necessary to perform more passes over the data in order to improve their accuracy or compress them. An alternative is to treat the XML stream as a series of insertions on an initially empty document, and subsequently employ incremental techniques, such as IMax [15] or Bloom Histogram [19], that maintain a synopsis as the data is updated. The problem with these techniques is that they are designed for environments with infrequent batch updates, and this can make them prohibitively expensive when the stream arrival rate is high. Finally, previous studies have investigated summarization techniques that rely on query feedback and thus do not access the data at all [12]. This approach is not relevant in our context due to the lack of query feedback information.

SketchTree [16] directly supports single-pass synopses and is thus the most relevant summarization technique for our setting. SketchTree employs randomized sketching techniques in order to summarize the frequency distribution of XML sub-trees up to a specific size. Given a query pattern, SketchTree first maps it to a set of matching sub-trees and then probes the synopsis for the respective occurrence counts. For queries that contain recursive structural constraints, e.g., the ancestor/descendant XPath axis, the first step implies the use of schema information so that the recursion can be "unfolded" in concrete paths. Unfortunately, a schema is not always available for the data, and inferring it from the stream is not a trivial task. This makes SketchTree suitable mainly for queries without recursion. Support for query recursion is crucial, however, as it allows queries to reference the XML stream without requiring precise knowledge of the underlying schema.

Several studies have investigated the problem of query evaluation over streaming XML data [2, 5, 10, 11]. These works target a complementary problem to approximate query answering, as they assume that the query processor has the required resources to generate precise query results. In fact, a system can evaluate queries for which there are enough computational resources, and fall back on approximate answers for more expensive queries.

The topic of approximate query answering over relational streams has gained significant traction in the recent literature [6, 9, 17]. The proposed techniques, however, are specific to the flat relational model and the associated query operators. An interesting question is whether they can be extended to handle the rich semantics of semi-structured data, where the use of recursion is prevalent in the

data and query model.

3. PRELIMINARIES

In this section, we discuss formally the data and query model for streaming XML data, and define the problem of approximate query answering over XML streams.

3.1 Data and Query Model

Streaming Model for XML Data. At an abstract level, an XML stream S represents the serialization of a (possibly infinite) XML document. In our work, we model the stream as a sequence of `open`, `value`, and `close` "parsing" events defined as follows: `open`(L) signals the opening tag of an element with label L, `close`(L) signals the closing tag of an element with label L, and `value`(V) specifies the value content V under the currently open element. (We do not model attributes separately, but treat them as leaf elements with value content.) We consider XML streams where the value content appears under leaf elements only. We note that the particular streaming model is readily implementable in practice, as it follows closely the Simple API for XML specification (a de-facto standard for the efficient parsing of XML documents).

Motivated by the context of on-line data analytics, we target applications that process the stream one event a time and perform a limited amount of computation per parsing event. In what follows, we use S to denote the event sequence that has been observed up to the current point in time.

It is straightforward to map S to a node-labeled, ranked tree that represents the underlying XML structure. More concretely, each `open`(L) event in S is mapped to a node with label L, and the nesting between two opening tags is represented as an edge between the corresponding nodes. We henceforth refer to a node e in this tree as an element observed in S. Given two elements e and e', we define the following common relations: e is a *child* of e' if the edge (e', e) exists; e is a *sibling* of e' if they are children of the same element; and, e *follows* e' in document order if e' is encountered first in a pre-order traversal of the tree.

Query Model. We focus on queries of the form "*Compute aggregate $Aggr$ on the binding tuples of a tree-pattern Q.*" This class of queries is natural within the scenario that we consider in this paper, namely, on-line trend analysis over streaming XML content. In what follows, we define the class of tree-pattern queries and the types of aggregates that we consider.

A tree-pattern query Q selects a subset of elements from an XML tree according to certain structural constraints. We model Q as a tree of labeled nodes $q_1, \ldots, q_n$, and refer to a node in Q as a query variable. Each variable q_j is assigned a label from the same alphabet as element labels, and can also be associated with an optional predicate that specifies a condition on element content. Each edge (q_k, q_j) is assigned an *axis specification* that describes the structural relationship between the two variables. In our work, we consider the following axes that are also defined in the XPath standard [3]: ancestor, descendant, parent, child, following, preceding, following-sibling, and preceding-sibling. Table 1 summarizes the semantics of these structural relationships. We often refer to the ancestor/descendant and following/preceding axes as recursive constraints, since they essentially allow the query to skip over an arbitrarily large portion of the XML structure. We also note that following/preceding and following-/preceding-sibling apply constraints on the ordering of elements in the XML tree. In what follows, we use $\mathcal{X}_a(q, q')$ to denote that the edge (q, q') is assigned axis α, and overload $\mathcal{X}_\alpha(e, e')$ to denote that two elements e and e' satisfy the structural constraint of the specific axis.

Let $\mathbf{e} = (e_1, \ldots, e_n)$ be a vector of elements. The vector is called a *binding tuple* for Q if element e_i matches the label and value predicate of variable q_i, and for each edge (q_k, q_j), $1 \leq k \neq j \leq n$, the corresponding elements match the structural constraint of the edge, i.e., $\mathcal{X}_\alpha(q_k, q_j) \Rightarrow \mathcal{X}_\alpha(e_k, e_j)$. The result of Q over S, denoted as $Q(S)$, is defined as the set of binding tuples that involve elements observed in S. Given that S grows continuously, it becomes clear that $Q(S)$ may be updated with more results as more of the stream is accessed. Figure 1(b) depicts an example tree-pattern query and its set of binding tuples over the stream.

As mentioned earlier, we are interested in the computation of an aggregate $Aggr$ over $Q(S)$. In this paper, we assume that $Aggr$ can be one of the following: *Count*, *Sum*, or *Average*. (The last two operate over the values of elements that are found in a specific slot of the binding tuples.)

Structural element labeling. Let e be an element observed in S such that the corresponding closing tag also appears in S. The structural identifier of e is defined as the triplet $(start, end, pstart)$, where $start$ and end are the positions in S of the `open` and `close` events corresponding to e, and $pstart$ is the position of the `open` event corresponding to the parent of e. The definition implies that the $start$ and end values are unique across elements, and also that $start < end$ for each structural identifier. We note that structural identifiers can be generated on-the-fly by maintaining a counter for the current length of the stream and a stack with the $start$ positions of the open elements.

Structural identifiers are commonly used in XML query engines to evaluate structural constraints, including the constraints specified in our query model [18, 21]. As an example, $\mathcal{X}_{prec}(e, e')$ implies that the `close` of e' appears before the `open` of e, and hence $1 < end' < start$. As another example, the constraint $\mathcal{X}_{precSib}(e, e')$ implies that e and e' are opened after their common parent element and that e' is closed before e. This can be expressed as $pstart < end' < start \wedge\ pstart = pstart'$. Table 1 summarizes the correspondence between structural constraints and conditions on structural identifiers.

In the remainder of the paper, we assume the existence of a conceptual relation $StructId(Start, End, PStart)$ that stores the structural identifiers of elements in S. Given that start positions are unique, we often use $start$ as a key in this relation and as an identifier for elements.

3.2 Problem Definition

In this paper, we tackle the problem of approximate query answering over XML streams defined as follows:

Given an XML stream S, a tree-pattern query Q, and an aggregate $Aggr$ that is either Count, Sum, or Average, maintain a bounded-size synopsis of S that can approximate the value of $Aggr$ over $Q(S)$.

The sequential access model of S implies that the approximation algorithm must maintain the stream synopsis incrementally as new events are accessed from the stream. Moreover, the algorithm can examine each new event exactly once, and can perform only limited processing in order to keep up with the potentially high arrival rate of S. (Note that buffering the stream and backtracking on it is not a viable option given our scenario of limited computational resources.) Finally, following the paradigm of previous works on relational stream processing [6, 9, 17], the formulation of the problem focuses on a stream synopsis that is specific to the query Q. This single-query/single-stream variant is still relevant for real-world applications, and, as we show in our work, it involves significant technical challenges. Therefore, it forms a good

Axis	Semantics	Condition
$\mathcal{X}_{child}(e,e') \Leftrightarrow \mathcal{X}_{par}(e',e)$	e' is a child of e	$pstart' = start$
$\mathcal{X}_{desc}(e,e') \Leftrightarrow \mathcal{X}_{anc}(e',e)$	$\mathcal{X}_{child}(e,e') \vee \exists e'' : \mathcal{X}_{child}(e'',e') \wedge \mathcal{X}_{desc}(e,e'')$	$start < start' < end$
$\mathcal{X}_{prec}(e,e') \Leftrightarrow \mathcal{X}_{fol}(e',e)$	(e follows e') $\wedge \neg\mathcal{X}_{desc}(e',e)$	$1 < end' < start$
$\mathcal{X}_{precSib}(e,e') \Leftrightarrow \mathcal{X}_{folSib}(e',e)$	(e' is a sibling of e) $\wedge \mathcal{X}_{prec}(e,e')$	$pstart < end' < start \wedge\ pstart = pstart'$

Table 1: Semantics and evaluation of structural relationships.

first step in the study of approximate query answering over XML streams. Extending the problem to multiple queries and multiple streams is an interesting direction for future work.

At this point, it is interesting to examine the aforementioned problem assuming that the input query Q can use only the descendant and child axes. This restricted query model has been studied extensively in previous works on XML summarization [7, 19, 20, 22, 14] and thus presents an interesting case for the approximation problem that we defined earlier. We show that the approximation problem becomes very easy for this type of queries, as there is a straightforward dynamic-programming algorithm that computes the precise value of $Aggr$ over $Q(S)$ using limited memory and processing resources. The main properties of the algorithm can be summarized as follows:

PROPOSITION 3.1. *Let S be an XML stream where the maximum depth is $h_{\max}$. Let Q be a tree-pattern query with n variables. Let $Aggr$ be an aggregate that is either Count, Sum, or Average. There exists an algorithm that computes the value of $Aggr$ on $Q(S)$ by performing $O(h_{max}n)$ computation for each event in S, and using $\mathcal{O}(h_{max}n)$ memory in total.*

Procedure DP-ALGORITHM.$processEvent(ev)$
Input: An event ev in the document stream.
Global Variables: Level h of current element in S;
tree query Q with nodes $q_1 \ldots q_n$;
2-d int array $descCount$; 2-d int array $childCount$.
Initialization:
for $i = 1 \ldots n$ **do** $childCount[0][i] \leftarrow 0$ **done**
for $i = 1 \ldots n$ **do** $descCount[0][i] \leftarrow 0$ **done**
$h \leftarrow 1$
begin
1.**if** $ev = \texttt{open}(L)$ **then**
2. **for** $i = 1 \ldots n$ **do** $childCount[h][i] \leftarrow 0$ **done**
3. **for** $i = 1 \ldots n$ **do** $descCount[h][i] \leftarrow 0$ **done**
4. $h \leftarrow h + 1$
5.**else if** $ev = \texttt{close}(L)$
6. $h \leftarrow h - 1$
7. **for** $i = 1 \ldots n$ **do** $count[i] \leftarrow 0$ **done**
8. **for each** q_i that matches L **do**
9. $count[i] \leftarrow 1$
10. **for each** constraint $\mathcal{X}_{child}(q_i, q_j)$ **do**
11. $count[i] \leftarrow count[i] * childCount[h][j]$
12. **done**
13. **for each** constraint $\mathcal{X}_{desc}(q_i, q_j)$ **do**
14. $count[i] \leftarrow count[i] * descCount[h][j]$
15. **done**
16. **done**
17. **for each** $q_i \in Q$ **do**
18. $childCount[h-1][i] \leftarrow childCount[h-1][i] + count[i]$
19. $descCount[h-1][i] \leftarrow descCount[h-1][i] + count[i]$
20. $descCount[h-1][i] \leftarrow descCount[h-1][i] + descCount[h][i]$
21. **done**
22.**fi**
end

Figure 2: DP algorithm pseudo-code.

Figure 2 shows the pseudo-code for the algorithm. At the time that the given element e at depth h is opened, the algorithm instantiates the counters $descendantCount[h][i]$ and $childrenCount[h][i]$ to 0 for each node q_i in Q. These counters will be updated by the children of e. Let Q_{q_i} be the subtree of the tree query Q rooted at q_i and let S_e be the subtree of the XML Stream S rooted at the element e. At the time that the element e closes, the algorithm calculates the count of the number of binding tuples of $Q_{q_i}(S_e)$ in $count[i]$ for each query node q_i that matches the currently open element e. Before e closes, the algorithm aggregates $count[i]$ to the counters $descendantCount[h-1][i]$ and $childrenCount[h-1][i]$ of its parent node for $i = 1 \ldots n$. Since the descendants of e are also the $descendants$ of the parent of e, the $descendantCount[h][i]$ is also added to $descendantCount[h-1][i]$. The algorithm returns the count accumulated in $descendantCount[0][r]$, where r is the index of the root node of Q.

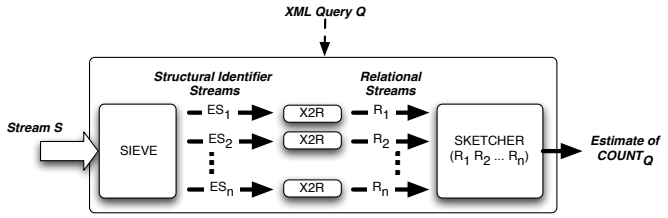


Figure 3: Conceptual overview of proposed solution.

Given that h_{max} and n are expected to be small in practice, it becomes clear that the approximation problem can be solved very efficiently for the restricted query model. The restricted query model, however, supports a limited set of tree patterns and, most importantly, it ignores element ordering. The latter is especially important in the context of stream monitoring, as ordering is a natural property in several application domains (e.g., event streams, or streams that capture execution traces). Overall, these observations provide strong motivation for the study of approximate query answering under the extended query model presented in Section 3.1.

To simplify presentation, we henceforth assume that $Aggr$ is *Count* and we use $\mathrm{COUNT}_Q(S)$ to denote the value of the aggregate over $Q(S)$, or simply COUNT_Q when S is clear from the context. The extension to *Sum* and *Average* is straightforward and is discussed in a later section.

4. SKETCH-BASED SUMMARIZATION OF XML STREAMS

In this section, we introduce a sketch-based XML stream synopsis for the approximate query answering problem defined in Section 3.2. In what follows, we first present an overview of our technique, and then discuss the details in the coming subsections.

Figure 3 provides a conceptual illustration of the proposed approach. The XML stream S is initially processed by the SIEVE operator that generates n output streams $ES_1, \ldots, ES_n$. The output stream ES_j corresponds to variable q_j in the query, $1 \leq j \leq n$, and it contains the structural identifiers of the elements in S that

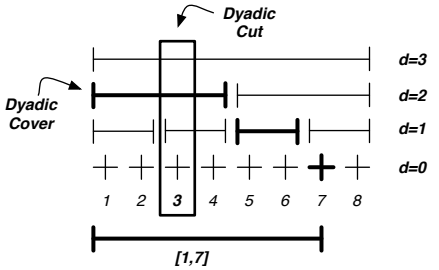


Figure 4: Dyadic decomposition for $[1, 2^3]$ and examples of a dyadic cover and a dyadic cut.

match q_j. More concretely, the SIEVE accesses S one event at a time, monitoring the appearance of `close` events. When such an event is encountered, the structural identifier of the corresponding element e is appended to stream ES_j if e matches the label and value predicate of q_j, $1 \leq j \leq n$. (The generation of structural identifiers is described in Section 3.1.) This process implies that the ordering of elements in each output stream is consistent with the ordering of the corresponding `close` events in S. We henceforth refer to this simple SIEVE implementation as basic sieving.

Each stream of structural identifiers ES_j is subsequently transformed to a relational stream R_j through a transform termed $\mathcal{X}2R$. The transform also defines a join query R_Q over $R_1, \ldots, R_n$ that has the following key property: the result cardinality of R_Q is equal to COUNT_Q. In essence, the computation of the XML aggregate is reduced to counting the result-set of a join query over relational streams.

The last component, termed the SKETCHER, reads the relational streams $R_1, \ldots, R_n$ and builds a sketch-based synopsis that can estimate the result cardinality of R_Q. By virtue of the $\mathcal{X}2R$ transform, the estimate is also an approximation for COUNT_Q.

Before proceeding with our presentation, we review the basic mechanism of dyadic decompositions that is used in the development of our approximation technique. Let D be a positive integer such that the length of S is not greater than 2^D, i.e., the $[start, end]$ interval of every structural identifier falls in $[1, 2^D]$. The dyadic decomposition of $[1, 2^D]$ at level d, $0 \leq d \leq D$ consists of 2^{D-d} disjoint intervals of length 2^d, termed *dyadic intervals*, whose union is equal to $[1, 2^D]$. Figure 4 illustrates this definition for $D = 3$. It is possible to show that any interval $[a, b]$ in $[1, 2^D]$ has a unique minimal representation, termed a *dyadic cover*, as the union of disjoint dyadic intervals from different levels. Similarly, we define the *dyadic cut* of a point c in $[1, 2^D]$ as the set of dyadic intervals that contain it. Figure 4 illustrates these definitions for interval $[1, 7]$ and point 3 respectively. The cover and cut can be computed very efficiently based solely on the level D of the decomposition, i.e., the complete decomposition does not have to be stored. In what follows, we use the relational predicate $DCover(a, b, \iota)$ to denote that dyadic interval ι belongs in the cover of $[a, b]$, and define $DCut(c, \iota)$ similarly.

As shown in the work of Das et al. [4], the dyadic cut and cover provide a mechanism to check for interval containment through set intersection. More formally, point c is included in $[a, b]$ if and only if there exists a dyadic interval ι such that $DCut(c, \iota) \wedge DCover(a, b, \iota)$. Moreover, there exists at most one such interval ι. Figure 4 shows an example of this property for $c = 3$ and $[a, b] = [1, 7]$.

4.1 $\mathcal{X}2R$ Transform

As mentioned earlier, $\mathcal{X}2R$ transforms the input stream of structural identifiers ES_j to an output relational stream R_j. Moreover, it defines a query R_Q over the generated streams $R_1, \ldots, R_n$ such that the following key properties are satisfied: R_Q is a tree-join query over $R_1, \ldots, R_n$ that involves only equality join predicates; and, the result cardinality of R_Q is equal to COUNT_Q.

The details of the transform are described below. Our presentation employs a notation whereby a relation is treated as a predicate. Thus, given a relation R and a tuple τ that conforms to the schema of R, we say that $R(\tau)$ is true if and only if τ belongs in the extent of R. In this fashion, we use a relational predicate $ES_j(start, end, pstart)$ to denote that the corresponding structural identifier appears in the input stream ES_j, $1 \leq j \leq n$. Given that $start$ values are unique, for convenience we use the shorthand notation $ES_j(start)$.

Single structural constraints. Let e and e' be two arbitrary elements in S with structural identifiers $(start, end, pstart)$ and $(start', end', pstart')$ respectively. As the first step in the development of $\mathcal{X}2R$, we define a relational predicate R_α for each axis $\mathcal{X}_\alpha$ of the query model such that $\mathcal{X}_\alpha(e, e') \Leftrightarrow R_\alpha(start, start')$, i.e., the two elements satisfy the structural constraint if and only if the unique pair of the corresponding start positions appears in the extent of a relation. As described in Section 3.1, the start position can serve as a key for the structural identifier of an element. The intuition, therefore, is to translate the constraints over elements to constraints over structural identifiers.

We begin our presentation by considering the translation of a specific constraint, namely, $\mathcal{X}_{precSib}(e, e')$. As shown in Table 1, the constraint is encoded as $pstart < end' < start \wedge pstart' = pstart$. The relational predicate $R_{precSib}$ encodes this condition as follows:

$$
\begin{aligned}
R^{src}_{precSib}(Start, I, PStart) \;&\leftarrow\; StructId(Start, \_, PStart),\\
&\qquad DCover(PStart, Start, I)\\
R^{tgt}_{precSib}(Start', I', PStart') \;&\leftarrow\; StructId(Start', End', PStart'),\\
&\qquad DCut(End', I')
\end{aligned}
$$

$$
\begin{aligned}
R_{precSib}(Start, Start') \;&\leftarrow\; R^{src}_{precSib}(Start, I, PStart),\\
&\qquad R^{tgt}_{precSib}(Start', I', PStart'),\\
&\qquad I = I', PStart = PStart'
\end{aligned}
$$

(Recall that $StructId$ is the set of all structural identifiers in S. We also use the symbol "_" to denote an unnamed variable.) Assume now that $R_{precSib}(start, start')$ is true, i.e., $(start, start')$ appears in the extent of $R_{precSib}$. For this to happen, there must be witnesses $R^{src}_{precSib}(start, \iota, pstart)$ and $R^{tgt}_{precSib}(start', \iota', pstart')$ such that $\iota = \iota' \wedge pstart = pstart'$. Based on the properties of the dyadic cover and cut, $\iota = \iota'$ implies that $pstart \leq end' \leq start$. Since $pstart = pstart'$, then $pstart' \leq end'$. Since e cannot be its own parent and $start$ and end values are unique across elements, it follows that $pstart' < end'$ and hence $pstart < end'$. Since $start < end$ for all elements and $end' \leq start$, it follows e and e' must be distinct. Since e and e' are distinct and $start$ and end values are unique across elements, it follows that $end' < start$. Thus, $pstart < end' < start \wedge pstart = pstart'$, and hence it follows that $\mathcal{X}_{precSib}(e, e')$ is true. Using a similar argument, we can also show $\mathcal{X}_{precSib}(e, e') \Rightarrow R_{precSib}(start, start')$.

Overall, the main idea is to express the XML constraint as a condition on structural identifiers, and then use dyadic decompositions

Constraint	Predicate
$\mathcal{X}_{desc}$	$R^{src}_{desc}(Start, X) \leftarrow StructId(Start, End, \_),\ DCover(Start + \zeta, End, I), X = I$ $R^{tgt}_{desc}(Start', Y) \leftarrow StructId(Start', \_, \_),\ DCut(Start', I'), Y = I'$ $R_{desc}(Start, Start') \leftarrow R^{src}_{desc}(Start, X),\ R^{tgt}_{desc}(Start', Y),\ X = Y$
$\mathcal{X}_{anc}$	$R_{anc}(Start, Start') \leftarrow R_{desc}(Start', Start)$
$\mathcal{X}_{child}$	$R^{src}_{child}(Start, X) \leftarrow StructId(Start, \_, \_),\ X = Start$ $R^{tgt}_{child}(Start', Y) \leftarrow StructId(Start, \_, PStart'),\ Y = PStart'$ $R_{child}(Start, Start') \leftarrow R^{src}_{child}(Start, X),\ R^{tgt}_{child}(Start', Y), X = Y$
$\mathcal{X}_{par}$	$R_{par}(Start, Start') \leftarrow R_{child}(Start', Start)$
$\mathcal{X}_{prec}$	$R^{src}_{prec}(Start, X) \leftarrow StructId(Start, \_, \_), DCover(1, Start, I), X = I$ $R^{tgt}_{prec}(Start', Y) \leftarrow StructId(Start', End', \_), DCut(End', I'), Y = I'$ $R_{prec}(Start, Start') \leftarrow R^{src}_{prec}(Start, X), R^{tgt}_{prec}(Start', Y), X = Y$
$\mathcal{X}_{fol}$	$R_{fol}(Start, Start') \leftarrow R_{prec}(Start', Start)$
$\mathcal{X}_{precSib}$	$R^{src}_{precSib}(Start, X) \leftarrow StructId(Start, \_, PStart), DCover(PStart, Start, I), X = (I, PStart)$ $R^{tgt}_{precSib}(Start', Y) \leftarrow StructId(Start', End', PStart'),\ DCut(End', I'),\ Y = (I', PStart')$ $R_{precSib}(Start, Start') \leftarrow R^{src}_{precSib}(Start, X),\ R^{tgt}_{precSib}(Start', Y),\ X = Y$
$\mathcal{X}_{folSib}$	$R_{folSib}(Start, Start') \leftarrow R_{precSib}(Start', Start)$

Table 2: Encoding of structural constraints as relational predicates with equi-joins.

to express interval containment as an equality of dyadic intervals. For reasons that will become apparent below, each relational predicate R_α is defined as the equi-join of two relations R^{src}_α and R^{tgt}_α. The intuition is that R^{src}_α contains information on the elements that appear in the "source" position of the constraint, i.e., element e in the expression $X_\alpha(e, e')$. Similarly, R^{tgt}_α corresponds to elements that appear in the target position. The evaluation of $R^{src}_\alpha \bowtie R^{tgt}_\alpha$ satisfies the property that a pair $(start, start')$ is generated at most once, which in turn ensures that the extent of R_α is duplicate-free under bag semantics. The latter becomes important in proving the correctness of the overall transform.

One technical detail pertinent to R_{desc} is the use of a displacement $0 < \zeta < 1$ on the structural identifiers of source elements. This is necessary in order to avoid asserting $R_{desc}(start, start)$, i.e., that e is a descendant of itself. It is straightforward to show that the displacement ζ does not change the descendant relationship if $e \neq e'$ (since $0 < \zeta < 1$), and it does not alter the asymptotic complexity of computing the dyadic cut and cover.

We can generalize the previous methodology to the other structural constraints of the query model, as shown in Table 2. We state this formally with the following lemma:

LEMMA 4.1. *$R_\alpha(start, start') \Leftrightarrow X_\alpha(e, e')$, and a pair $(start, start')$ is generated at most once in the computation of R_α.*

Proof: Consider the case where $\alpha = desc$. Assume $R_{desc}(start, start')$ is true, i.e., $(start, start')$ appears in the extent of R_{desc}. For this to happen, there must be witnesses $R^{src}_{desc}(start, \iota)$ and $R^{tgt}_{desc}(start', \iota')$ such that $\iota = \iota'$. Due to the property of the dyadic cover and cut, this implies that $start + \zeta \leq start' \leq end$. Since $\zeta > 0$, then $start < start'$. Since $start \neq start'$, this means that the corresponding elements e and e' are distinct. Because $start$ and end values are unique across elements, it follows that $start' \neq end$; therefore, $start < start' < end$. Hence, it follows that $\mathcal{X}_{desc}(e, e')$.

Assume $\mathcal{X}_{desc}(e, e')$ is true, i.e., the structural constraint specified by the descendant axis is satisfied by e and e'. Hence, the condition between the structural identifiers of e and e' for the descendant axis in Table 1 holds. In this case, it holds that $start < start' < end$. Since $\zeta < 1$, $start + \zeta < start' < end$. Due to the property of the dyadic cover and cut, this implies that there is an interval ι, such that $R^{src}_{desc}(start, \iota)$ and $R^{tgt}_{desc}(start', \iota)$ and there is at most one such interval ι. Hence, it follows that $R_{desc}(start, start')$ and $(start, start)$ appears at most once in the extent of R_{desc}.

Since $R_{desc}(start, start') \Rightarrow \mathcal{X}_{desc}(e, e')$ and $\mathcal{X}_{desc}(e, e') \Rightarrow R_{desc}(start, start')$, then $R_{desc}(start, start') \Leftrightarrow \mathcal{X}_{desc}(e, e')$.

The same can be shown for the remaining axes in Table 2 in a similar manner. ∎

Composing constraints. The next step in the development of the transform is the composition of the aforementioned relational predicates in order to generate the output streams R_k, $1 \leq k \leq n$. We present the main idea with an example and then provide the general definition of the transform.

Consider a query with variables q_1, q_2, q_3 and the constraints $\mathcal{X}_{precSib}(q_1, q_2)$ and $\mathcal{X}_{desc}(q_2, q_3)$. Let e_1, e_2, e_3 be elements in S and let $(start_i, end_i, pstart_i)$ denote the structural identifier of element e_i, $i \in \{1, 2, 3\}$. We will define a relational predicate R_Q such that the following holds: (e_1, e_2, e_3) is a binding tuple if and only if $R_Q(start_1, start_2, start_3)$. First, we observe that e_i must match q_i, $i \in \{1, 2, 3\}$, in order for (e_1, e_2, e_3) to be considered as a binding tuple. This can be expressed as the relational predicate $ES_i(start_i)$ that tests for the existence of the corresponding identifier in the input stream ES_i. (Recall that each input stream ES_i contains the structural identifiers of elements that match q_i.) Second, the elements must satisfy the following constraints: $\mathcal{X}_{precSib}(e_1, e_2) \wedge \mathcal{X}_{desc}(e_2, e_3)$. Using the equivalent relational predicates, this can be checked as $R_{precSib}(start_1, start_2) \wedge R_{desc}(start_2, start_3)$. We thus arrive at the following definition for R_Q:

$$\begin{aligned} R_Q(Start_1, Start_2, Start_3) \leftarrow\ & ES_1(Start_1), \\ & ES_2(Start_2), ES_3(Start_3), \\ & R_{precSib}(Start_1, Start_2), R_{desc}(Start_2, Start_3) \end{aligned}$$

By expanding each relational predicate R_α based on its definition in Table 2, we can rewrite the previous expression as follows:

$$\begin{aligned} R_1(Start_1, X) &\leftarrow ES_1(Start_1), R^{src}_{precSib}(Start_1, X) \\ R_2(Start_2, Y, X') &\leftarrow ES_2(Start_2), R^{tgt}_{precSib}(Start_2, Y), \\ &\quad\ R^{src}_{desc}(Start_2, X') \\ R_3(Start_3, Y') &\leftarrow ES_3(Start_3), R^{tgt}_{desc}(Start_3, Y') \\ R_Q(Start_1, Start_2, Start_3) &\leftarrow R_1(Start_1, X), R_2(Start_2, Y, X'), \\ &\quad\ R_3(Start_3, Y'), X = Y, X' = Y' \end{aligned}$$

The final rewriting expresses R_Q as an equi-join query over three relations R_1, R_2, R_3. Moreover, (e_1, e_2, e_3) is a binding tuple if

and only if $(start_1, start_2, start_3)$ appears in the result of R_Q. It is interesting to note that each relation R_i captures the query constraints that reference variable q_i. As an example, R_2 involves $R^{tgt}_{precSib}$ that corresponds to the target of constraint $\mathcal{X}_{precSib}(q_1, q_2)$, and R^{src}_{desc} that corresponds to the source of $\mathcal{X}_{desc}(q_2, q_3)$. A structural identifier $(start_2, end_2, pstart_2)$ in stream ES_2 contributes a distinct tuple-set to R_2 that is formed essentially by the cross product of $R^{src}_{desc}(start_2, X')$ and $R^{tgt}_{precSib}(start_2, Y')$. (Hence, it is possible to generate R_2 on-the-fly by processing each structural identifier in ES_2, computing the aforementioned cross product, and appending it to stream R_2.) This cross product can be viewed as an encoding of the structural identifier relative to the constraints of variable q_2. Using the same intuition, we can view the join predicates in R_Q as the means to couple the encoded information for each edge (q_i, q_j) in the query.

We now define formally the transform for an arbitrary query Q. Consider a variable q_k, $1 \leq k \leq n$, and assume for simplicity that the children of q_k can be enumerated as $q_{l_k}, q_{l_k+1}, \ldots, q_{h_k}$. For each "outgoing" constraint $\mathcal{X}_\alpha(q_k, q_j)$, $l_k \leq j \leq h_k$, we define R^{src}_j as the source predicate of R_α in Table 2. For instance, $R^{src}_j(Start, X_j) \leftarrow R^{src}_{precSib}(Start, X_j)$ if the constraint is $\mathcal{X}_{precSib}(q_k, q_j)$. (Notice that we perform a renaming $X \mapsto X_j$.) Similarly, for the "incoming" constraint $\mathcal{X}_\alpha(q_p, q_k)$, we define a relation R^{tgt}_k as the target predicate of R_α. For instance, $R^{tgt}_k(Start, Y_k) \leftarrow R^{tgt}_{precSib}(Start, Y_k)$ if the constraint is $\mathcal{X}_{precSib}(q_p, q_k)$. We define R_k as the relation that composes these per-edge target and source predicates for elements that match variable q_k:

$$R_k(Start, Y_k, X_{l_k}, \ldots, X_{h_k}) \leftarrow ES_k(Start, End, PStart),$$
$$R^{tgt}_k(Start, Y_k), R^{src}_{l_k}(Start, X_{l_k}), \ldots, R^{src}_{h_k}(Start, X_{h_k})$$

Relation R_1 for the root variable q_1 is defined similarly, except that we drop variable Y_1 and predicate $R^{tgt}_1(Start, Y_1)$ since there is no incoming edge. We observe that R_k includes a distinct tuple-set for each structural identifier in ES_k and can thus be generated on-the-fly from the input stream. The tuple-set for each identifier is computed as the cross product of source and target relations corresponding to the incoming and outgoing constraints of variable q_k. Again, this cross product can be viewed as an encoding of the structural identifier in terms of these constraints.

The query R_Q of the transform joins the output streams $R_1, \ldots, R_n$, applying an equi-join predicate per edge that couples the corresponding source and target predicates:

$$R_Q(Start_1, \ldots, Start_n) \leftarrow R_1(Start_1, X_{l_1}, \ldots, X_{h_1}),$$
$$\bigwedge_{2 \leq k \leq n} R_k(Start_k, Y_k, X_{l_k}, \ldots, X_{h_k}), \quad \bigwedge_{2 \leq k \leq n} X_k = Y_k$$

The following result states formally that the transform achieves the properties stated at the beginning of the section.

THEOREM 4.1. *R_Q is a tree-join query over $R_1, \ldots, R_n$ with equi-join predicates. The result cardinality of R_Q under bag semantics is equal to* COUNT_Q.

Proof: The first property stems directly from the tree structure of Q and the fact that an equi-join predicate is inserted per query edge. The proof of the second property is based on Lemma 4.1.

First, we want to show that $|Q(S)| \leq |R_Q|$. Let $\mathbf{t} = (e_1, \ldots, e_n)$ be a binding tuple in $Q(S)$. Since $\mathbf{t} \in Q(S)$, then all $e_i \in \mathbf{t}$ have the same label as the corresponding query variable q_i and match any corresponding value predicate; hence, $start_i$ appears in the stream ES_i. In addition, for every $(q_i, q_j) \in Q$, it holds that $X_\alpha(e_i, e_j)$ since the structural constraint $X_\alpha(q_i, q_j)$ of the edge is satisfied by $\mathbf{t}$. It follows from the second property stated above that $R_\alpha(start_i, start_j)$.

Without loss of generality assume q_1 is the root of Q. For each variable $q_i \in Q$ there is a set of tuples R_i corresponding to $e_i \in \mathbf{t}$. For e_1, there is

$$R_1(start_1, X_{l_1}, \ldots, X_{h_1}) \leftarrow ES_1(start_1, end_1, pstart_1),$$
$$R^{src}_{l_1}(start_1, X_{l_1}), \ldots, R^{src}_{h_1}(start_1, X_{h_1})$$

and for $e_2, \ldots, e_n$

$$R_i(start_i, Y_i, X_{l_i}, \ldots, X_{h_i}) \leftarrow ES_i(start_i, end_i, pstart_i),$$
$$R^{tgt}_i(start_i, Y_i), R^{src}_{l_i}(start_i, X_{l_i}), \ldots, R^{src}_{h_i}(start_i, X_{h_i}).$$

We can rewrite

$$R_1(start_1, X_{l_1}, \ldots, X_{h_1}),$$
$$\bigwedge_{2 \leq k \leq n} R_k(start_k, Y_k, X_{l_k}, \ldots, X_{h_k}), \quad \bigwedge_{2 \leq k \leq n} X_k = Y_k$$

as

$$\bigwedge_{(q_i, q_j) \in Q} R^{src}_i(start_i, X_i), R^{tgt}_j(start_j, Y_j), X_i = Y_j,$$
$$\bigwedge_{1 \leq k \leq n} ES_k(start_k, end_k, pstart_k)$$

From the definition of $R_\alpha(Start_i, Start_j)$ and the conjunction of $R_\alpha(start_i, start_j)$ over all $(q_i, q_j) \in Q$, it follows that for the binding tuple $\mathbf{t}$, there is at least one binding tuple in R_Q. Hence, under bag semantics $|Q(S)| \leq |R_Q|$.

Let $\mathbf{r} = (start_1, \ldots, start_n)$ be a tuple in R_Q. It follows that the expression above is satisfied. Then for every $(q_i, q_j) \in Q$, there exists elements e_i and e_j that satisfy $R^{src}_i(start_i, X_i)$, $R^{tgt}_j(start_j, Y_j), X_i = Y_j$. Hence, they also satisfy $X_\alpha(e_i, e_j)$. Since they appear in ES_i and satisfy the edge constraints in Q, then $(e_1, \ldots, e_n)$ must be a binding tuple in $Q(S)$. Hence, $|R_Q| \leq |Q(S)|$.

Since $|R_Q| \leq |Q(S)|$ and $|Q(S)| \leq |R_Q|$, then $|R_Q| = |Q(S)|$. ∎

As mentioned earlier, it is possible to compute R_k on-the-fly from the input stream of structural identifiers. A natural question, however, concerns the efficiency of the transform, as the contribution of each structural identifier involves the computation of a cross product of several relations. We address this issue of efficiency in Section 4.2, where we present an optimization that avoids completely the computation of the cross product.

It should be noted that previous studies have proposed similar transforms from XML to relational queries. The distinguishing property of the $\mathcal{X}2R$ transform is that it generates relational rewritings in the specific class of tree-join queries with equi-join predicates. Existing transforms, such as start/end/level, pre/post/level, or Dewey, do not enable this property for all the XPath axes that we consider in the query model. For instance, the previous transforms would translate a following-/preceding-sibling constraint to a join predicate with a range condition.

Extensions to Unbounded Streams. Up to this point, we have assumed that the length of the document stream is bounded by 2^D. There are cases, however, where this a-priori bounding is not possible, e.g., when the stream is infinite or its length is unknown. It is

straightforward to overcome this limitation by using virtual copies of the decomposition as the stream grows. Returning to the example of Figure 4, this would amount to using a virtual copy of the interval hierarchy in order to extend the decomposition to the interval $[1, 16]$. The downside is that the dyadic cover can become larger compared to a decomposition that uses a higher level D. We note that the dyadic cut and cover are computed using a similar algorithm as in the case of a single decomposition. In addition to the level D, the algorithm also keeps track of the number of virtual copies that have been added so far. (Again, it is not necessary to materialize the decomposition or any of the copies.)

The choice of the specific D depends heavily on the characteristics of the data. A useful feature of the $\mathcal{X}2R$ transform is that it can employ a different decomposition level per query edge and thus tailor it to the characteristics of the corresponding axis. For the case of a descendant edge, for instance, the intervals $[start, end]$ tend to be narrow in practice (since XML trees are typically shallow and bushy) and hence a low dyadic level yields small dyadic covers and cuts. As a different example, the preceding axis involves intervals of the form $[1, start]$ which tend to be wider. In this case, a higher D is likely to compress the representation of these large intervals.

4.2 Sketch-Based Approximation

At an abstract level, the SKETCHER solves the following problem: Maintain a synopsis of streams $R_1, \ldots, R_n$ that can estimate the cardinality of R_Q (equivalently, COUNT_Q). This particular problem has been the topic of active research in recent years and previous studies have introduced several techniques that match our setting [6, 8, 17]. In this paper we adopt the techniques proposed by Dobra et al. [6], but we stress that this choice is orthogonal to the other components of our framework.

Basic Sketching Technique [6]. Consider a pair of joining variables X_k and Y_k in R_Q, and let $\{1, \ldots, N_k\}$ denote their domain. (For instance, if X_k and Y_k correspond to the R_{desc} predicate, then $\{1, \ldots, N_k\}$ is simply an enumeration[1] of the intervals in the dyadic decomposition.) We assume the existence of a family $\xi^k = \{\xi_i^k | 1 \leq i \leq N_k\}$ of four-wise independent random variables such that $\xi_i^k \in \{-1, 1\}$ and $Prob[\xi_i^k = -1] = Prob[\xi_i^k = 1] = 1/2$. Informally, four-wise independence implies that the probability of any combination $(\xi_{i_1}^k, \xi_{i_2}^k, \xi_{i_3}^k, \xi_{i_4}^k)$ matching a specific 4-tuple of $\{-1, +1\}$ is equal to $1/16$, i.e., all 4-tuples of $\{-1, +1\}$ are equiprobable. By employing known tools (e.g., orthogonal arrays) for the explicit construction of small sample spaces supporting four-wise independence, such families can be efficiently constructed on-line using only $\mathcal{O}(\log N_k)$ space [1].

For each input stream R_k, $1 \leq k \leq n$, we define a counter Ξ_k, termed an *atomic sketch*, as follows:

$$\Xi_k = \sum_{R_k(Start, Y_k, X_{l_k}, \ldots, X_{h_k})} \xi_{Y_k}^k \prod_{l_k \leq j \leq h_k} \xi_{X_j}^j \qquad (1)$$

Accordingly, we define the random variable $\Phi = \prod_{1 \leq j \leq n} \Xi_j$ as the product of the per-relation atomic sketches and refer to it as an *atomic estimator*. As shown in the work of Dobra et al. [6], Φ is an unbiased, bounded-variance estimator of the cardinality of R_Q. The variance of the estimator is bounded as $Var[\Phi] \leq 2^{2(n-1)} \prod_{j=1}^{n} SJ(R_j)$, where $SJ(R_j) = |R_j \bowtie R_j|$ is the self-join size of relation R_j.

Application to XML Streams. The realization of the XML stream synopsis is a straightforward application of the basic sketching technique. More concretely, the synopsis comprises the atomic sketches $\Xi_1, \ldots, \Xi_n$ that are maintained incrementally over the stream. When an estimate is requested, the stream S is first "patched" with a virtual `close` for each currently open element. This has the effect of "pushing" all currently open elements through the $\mathcal{X}2R$ transform and thus to the atomic counters. After this update, the estimate is computed as the product of the atomic sketch counters. By virtue of the $\mathcal{X}2R$ transform and the properties of randomized sketches, we can assert the following property.

PROPOSITION 4.1. *$\Phi = \prod_{j=1}^{n} \Xi_j$ is an unbiased estimator of* COUNT_Q.

Proof: Recall from Section 4.2 that $\Phi = \prod_{1 \leq k \leq n} \Xi_k$ where Ξ_k is given by Equation 1. Here we will use the short hand notation R_1 to represent $R_1(Start, X_{l_1}, \ldots, X_{h_1})$ and R_k to represent $R_k(Start, Y_k, X_{l_k}, \ldots, X_{h_k})$ for $k = 2, \ldots, n$. Thus,

$$\Phi = (\sum_{R_1} \prod_{l_1 \leq j \leq h_1} \xi_{X_j}^j) \prod_{2 \leq k \leq n} \left[\sum_{R_k} (\xi_{Y_k}^k \prod_{l_k \leq j \leq h_k} \xi_{X_j}^j) \right].$$

The expression for Φ can be rewritten as follows:

$$\Phi = \sum_{R_1 \times \cdots \times R_n} \prod_{2 \leq k \leq n} \xi_{Y_k}^k \xi_{X_k}^k$$

Furthermore, the expression for Φ can be separated into the sum over the tuples in the cross product of the streams $R_1 \ldots R_n$ where $\forall k : Y_k = X_k$ plus the sum over the tuples where $\exists k : Y_k \neq X_k$. In addition, since the random variable $\xi_i^k \in \{-1, 1\}$ for a value i, then $(\xi_i^k)^2 = 1$. Therefore,

$$\begin{aligned} \Phi = & \sum_{\substack{R_1 \times \cdots \times R_n \\ \exists k : Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi_{Y_k}^k \xi_{X_k}^k \\ & + \sum_{\substack{R_1 \times \cdots \times R_n \\ \forall k : Y_k = X_k}} \prod_{2 \leq k \leq n} 1 \end{aligned} \qquad (2)$$

Because different ξ^k families are independent, it follows that

$$\begin{aligned} E[\Phi] = & \sum_{\substack{R_1 \times \cdots \times R_n \\ \exists k : Y_k \neq X_k}} \prod_{2 \leq k \leq n} E[\xi_{Y_k}^k \xi_{X_k}^k] \\ & + \sum_{\substack{R_1 \times \cdots \times R_n \\ \forall k : Y_k = X_k}} \prod_{2 \leq k \leq n} 1. \end{aligned}$$

Because random variables within each family ξ^k are four wise independent, the first set of terms sums to 0. In addition, the second set of terms sums to COUNT_Q since the corresponding tuple for each term is such that $y_k = x_k$ for all k and thus it contributes 1 to the sum.

$$E[\Phi] = \sum_{\substack{R_1 \times \cdots \times R_n \\ \forall k : Y_k = X_k}} \prod_{2 \leq k \leq n} 1 \qquad (3)$$

$$E[\Phi] = \text{COUNT}_Q$$

∎

After the estimate is generated, the synopsis is brought to a consistent state by re-processing the virtual `close` events and decrementing the atomic sketch counters by the corresponding ξ_i^k's.

The accuracy of approximation can be improved through a standard boosting technique [6] that combines several i.i.d. atomic estimates. More concretely, the synopsis computes $s_1 s_2$ atomic estimates Φ_{ij}, $1 \leq i \leq s_1$, $1 \leq j \leq s_2$ as described previously.

[1] This enumeration can be generated on-the-fly by using pairing functions or approximate techniques such as Rabin fingerprints.

The final estimator is generated as the median of s_2 estimators $\bar{\Phi}_1, \ldots, \bar{\Phi}_{s_2}$, where each $\bar{\Phi}_j$ is computed as the average of s_1 estimators Φ_{ij}, $1 \leq i \leq s_1$. The following proposition describes the error guarantees that can be achieved by this process and follows directly from Theorem 3.1 in [6]:

PROPOSITION 4.2. *Let ϵ and δ be constants in the range (0,1]. Setting $s_1 = 8\frac{2^{2(n-1)}\prod_{j=1}^{n} SJ(R_j)}{\mathrm{COUNT}_Q^2 \epsilon^2}$ and $s_2 = 2log(1/\delta)$, it is possible to approximate COUNT_Q so that the relative error of the estimate is at most ϵ with probability at least $1-\delta$.*

One caveat of the previous guarantees is that they involve the self-join sizes of the input relations and also the value of COUNT_Q, which is the precise quantity that we wish to estimate. This paradox is common in randomized approximation techniques, and it is typically handled by employing bounds on the unknown factors. (Such bounds can be derived using domain knowledge or historic data.) We also observe that the required storage increases sharply with the number of variables n participating in the query. While it is difficult to predict a-priori the maximum number of variables used in Q, it is reasonable to expect that real-world queries will have a low number of variables.

Maintenance of Atomic Sketches. The $\mathcal{X}2R$ transform maps each identifier in stream ES_k to a tuple-set in the corresponding relation stream R_k, which is in turn translated to several updates (one per tuple) to the corresponding atomic counter Ξ_k. We now discuss two optimizations that we incorporate in the SKETCHER in order to improve the efficiency of updating the atomic counters for each structural identifier in ES_k.

The first optimization reduces the size of the tuple-set by removing unnecessary tuples. More specifically, it is straightforward to show that the properties of the $\mathcal{X}2R$ transform are not compromised if we omit unit-length intervals from $DCover$ and $DCut$. (This stems from the fact that $start$ and end positions are unique and therefore the equi-join of $DCover$ and $DCut$ will never contain unit-length intervals.) This optimization may reduce the size of the cross product for each identifier, and thus lead to fewer updates of the atomic sketches.

The second optimization targets the update of the atomic counter for the tuples in the cross product. The main idea is to avoid updating the atomic counter for each tuple in the cross product, but rather to perform one cumulative update. More concretely, let $R_k(start, Y_k, X_{l_k}, \ldots, X_{h_k})$ be the tuple-set generated by the $\mathcal{X}2R$ transform for a particular identifier $sid \equiv (start, end, pstart)$. Let $\Delta_k(sid)$ be the total change to Ξ_k after sketching all the tuples in the tuple-set according to Equation 1. By performing standard algebraic manipulations on the expression that defines $\Delta_k(sid)$, we arrive at the following result:

$$\Delta_k(sid) = \left(\sum_{R_k^{tgt}(start, Y_k)} \xi_{Y_k}^k \right) \prod_{l_k \leq j \leq h_k} \left(\sum_{R_j^{src}(start, X_j)} \xi_{X_j}^j \right) \quad (4)$$

Essentially, the update can be realized by sketching separately the tuples in the source and target predicates and then adding their product to Ξ_k. This suggests the following optimization: Instead of computing a cross product for each identifier in ES_k, the $\mathcal{X}2R$ operator pushes directly the tuple-sets $R_k^{tgt}(start, Y_k)$ and $R_j^{src}(start, X_j)$, $l_k \leq j \leq h_k$, to the SKETCHER; in turn, the SKETCHER employs the previous expression to compute the change to Ξ_k. This optimization implies a linear complexity $\mathcal{O}(h_k - l_k)$ to process an identifier in ES_k through the $\mathcal{X}2R$ transform and the SKETCHER, whereas the complexity of the straightforward implementation (based on the computation of the cross product) has an exponential dependency to $h_k - l_k$. We have found in our empirical study that this optimization can yield significant time savings in the maintenance of the synopsis.

4.3 Extension to Sum

In this section, we discuss the extension to *Sum* . Note that once estimates for *Sum* and *Count* are obtained, an estimate for *Average* can be obtained. As before, we have a tree-pattern query Q, an XML Stream S, and the set of binding tuples $Q(S)$. As mentioned earlier, *Sum* operates over the values of elements that are found in a specific slot of the binding tuples $Q(S)$. We denote the value of *Sum* over the values of the elements in the specific slot i of $Q(S)$ as SUM_{Q_i}.

We extend the conceptual relation $StructId$ to include the value of the elements as follows: $StructId(Start, End, PStart, Value)$. The stream ES_i of structural identifiers that correspond to the ith position is extended to include the value of the elements, and we use the relational predicate $ES_i(start, end, pstart, value)$ to denote the corresponding structural identifiers and values that appear in ES_i. The relational stream R_i that is output by the $\mathcal{X}2R$ operator for the specified position i is defined as follows:

$$R_i(Start, Value, Y_i, X_{l_i}, \ldots, X_{h_i}) \leftarrow ES_i(Start, End, PStart, Value), \\ R_i^{tgt}(Start, Y_i), R_{l_i}^{src}(Start, X_{l_i}), \ldots, R_{h_i}^{src}(Start, X_{h_i})$$

All other streams ES_k and R_k, for $k \neq i$, are defined as before. We can define a *Sum* query R_Q over $R_1 \ldots R_n$ with the same equi-join predicates as for the *Count* query except now we sum over the $Value$ attribute of R_i. The result of the *Sum* query R_Q is equal to SUM_{Q_i}.

We define the atomic sketch Ξ_i for the input relational stream R_i as follows:

$$\Xi_i = \sum_{R_i(Start, Value, Y_i, X_{l_i}, \ldots, X_{h_i})} Value \cdot \xi_{Y_i}^i \prod_{l_i \leq j \leq h_i} \xi_{X_j}^j \quad (5)$$

All other atomic sketches Ξ_k, for $k \neq i$, are defined as before. We define the random variable Φ as before but with the new definition for the atomic sketch Ξ_i. As shown by Dobra et al. [6], this is an unbiased estimator for the *Sum* query R_Q and hence for SUM_{Q_i}.

5. STRUCTURAL SIEVING

Up to this point, we have assumed the basic sieving strategy that routes a structural identifier to stream ES_j if the corresponding element matches variable q_j, $1 \leq j \leq n$. In this section, we introduce a more elaborate strategy, termed structural sieving, that improves the accuracy of the sketch-based approximation by reducing the variance of the estimator. The new strategy is based on two techniques: (a) structural filtering, which utilizes stricter criteria to match elements to variables and thus reduces the volume of data that is sketched, and (b) structural partitioning, which divides S into sub-streams that can be sketched more accurately.

Figure 5 depicts the pseudo-code for the new SIEVE operator. Given an element e that is currently open at depth h, the SIEVE maintains the current number of preceding and descendant elements in $precCount[h]$ and $descCount[h]$ respectively. The SIEVE also maintains a set $currentMatch[h]$ with the variables matched by e. The contents of $currentMatch[h]$ are initialized when e opens, and they are used to route e to the output streams when the element closes. In between, the application of structural filtering

(*FilterOpen* and *FilterClose*) may reduce the set of matching variables thus routing e to fewer streams. In addition, after an element is routed to the output streams, the SIEVE invokes structural partitioning (*CheckPartition*) to check whether e identifies a portion of the stream that can be sketched separately. The following sections discuss these two techniques in more detail.

Procedure STRUCTURALSIEVE.$processEvent(ev)$
Input: An event ev in the document stream.
Global Variables: Level h of current element in S;
int array $descCount$; int array $precCount$.
Initialization: $h \leftarrow 0$; $descCount[0] \leftarrow 0$; $precCount[0] \leftarrow 0$
begin
1.**if** $ev = \texttt{open}(L)$ **then**
2. $h \leftarrow h + 1$
3. $precCount[h] \leftarrow precCount[h-1] + descCount[h-1]$
4. $descCount[h] \leftarrow 0$
5. $currentMatch[h] \leftarrow \{q \mid q \text{ matches } L\}$
6. $FilterOpen()$
7.**else if** $ev = \texttt{value}(V)$ **then**
8. remove q from $currentMatch[h]$ if predicate does not match V
9.**else if** $ev = \texttt{close}(L)$
10. $FilterClose()$
11. **for each** $q \in currentMatch[h]$ **do**
12. append the structural identifier to $ES_j : q_j \equiv q$
13. **done**
14. $CheckPartition()$
15. $descCount[h-1] \leftarrow descCount[h-1] + descCount[h] + 1$
16. $h \leftarrow h - 1$
17.**fi**
end

Figure 5: Structural Sieve pseudo-code.

5.1 Structural Filtering

Structural filtering reduces the size of streams $ES_1, \ldots, ES_n$ by removing elements that are irrelevant to the computation of binding tuples. This has the effect of reducing the size of relations $R_1, \ldots, R_n$ and hence the self-join factors that have a crucial effect on the variance of the estimator.

Figure 6 shows the pseudo-code for the structural filtering algorithm. The algorithm maintains state for the currently open elements using arrays $ancMatch$, $descMatch$, $childMatch$, and $prevMatch$. Given an element that is currently open at level h, the entries $ancMatch[h]$, $descMatch[h]$, $childMatch[h]$, and $prevMatch[h]$ store the set of query variables that may match with ancestor, descendant, child, and preceding elements respectively.

The state of an element e is initialized when it is opened, based on the state of its parent element. This initialization includes the previously mentioned arrays and two local variable-sets $parentMatch$ and $prevSiblingMatch$ that track the variables matched by the parent and the preceding-sibling elements respectively. Subsequently, the algorithm examines each q in $currentMatch[h]$ and the edges that are adjacent to it. The goal is to verify that e can indeed appear in a binding tuple along with other elements so that the structural constraints of the edges are satisfied. As an example, assume that Q contains the constraint $\mathcal{X}_{desc}(q', q)$. If there is no ancestor element of e that is a match for q', i.e., $q' \notin ancMatch[h]$, then it is clear that e cannot appear in $Q(S)$ as a binding of q. Similar conditions are checked for the other axes using information that is known when e is opened, namely, the matches for ancestor, parent, preceding, and preceding sibling elements. If any condition fails, then q is removed from $currentMatch[h]$. We note that it is not possible to use information on the descendants or following elements of e, since those have not been explored yet. Hence, the edges that rely on this information cannot be filtered at this point in time.

Procedure *FilterOpen*()
Global Variables: Arrays of variable-sets $childMatch$, $descMatch$, $prevMatch$, $ancMatch$
Initialization: $childMatch[0] \leftarrow \emptyset$; $ancMatch[0] \leftarrow \emptyset$; $prevMatch[0] \leftarrow \emptyset$; $descMatch[0] \leftarrow \emptyset$
begin
1.$parentMatch \leftarrow currentMatch[h-1]$
2.$ancMatch[h] \leftarrow ancMatch[h-1] \cup parentMatch$
3.$prevMatch[h] \leftarrow prevMatch[h-1] \cup descMatch[h-1]$
4.$prevSiblingMatch \leftarrow childMatch[h-1]$
5.$descMatch[h] \leftarrow childMatch[h] \leftarrow \emptyset$
6.**for each** $q \in currentMatch[h]$ **do**
7. $match \leftarrow \texttt{true}$
8. **for each** constraint $\mathcal{X}_\alpha(q', q) : \alpha \in \{desc, child, fol, folSib\}$ **do**
9. $match \leftarrow (\alpha = desc \wedge q' \in ancMatch[h]) \vee$
$(\alpha = child \wedge q' \in parentMatch) \vee$
$(\alpha = fol \wedge q' \in prevMatch[h]) \vee$
$(\alpha = folSib \wedge q' \in prevSiblingMatch)$
10. **done**
11. **for each** constraint $\mathcal{X}_\alpha(q, q') : \alpha \in \{par, anc, prec, precSib\}$ **do**
12. $match \leftarrow match \wedge ((\alpha = par \wedge q' \in parentMatch) \vee$
$(\alpha = anc \wedge q' \in ancMatch[h]) \vee$
$(\alpha = prec \wedge q' \in prevMatch[h]) \vee$
$(\alpha = precSib \wedge q' \in prevSiblingMatch))$
13. **done**
14. **if** $\neg match$ **then** remove q from $currentMatch[h]$ **end if**
15.**done**
end

Procedure *FilterClose*()
begin
1.**for each** $q \in currentMatch[h]$ **do**
2. $match \leftarrow \texttt{true}$
3. **for each** constraint $\mathcal{X}_\alpha(q', q) : \alpha \in \{par, anc\}$ **do**
4. $match \leftarrow (\alpha = anc \wedge q' \in descMatch[h]) \vee$
$(\alpha = par \wedge q' \in childMatch[h])$
5. **done**
6. **for each** constraint $\mathcal{X}_\alpha(q, q') : \alpha \in \{desc, child\}$ **do**
7. $match \leftarrow match \wedge ((\alpha = desc \wedge q' \in descMatch[h]) \vee$
$(\alpha = child \wedge q' \in childMatch[h]))$
8. **done**
9. **if** $\neg match$ **then** remove q from $currentMatch[h]$ **end if**
10.**done**
11.$subtreeMatch \leftarrow descMatch[h] \cup currentMatch[h]$
12.$descMatch[h-1] \leftarrow descMatch[h-1] \cup subtreeMatch$
13.$childMatch[h-1] \leftarrow childMatch[h-1] \cup currentMatch[h]$
end

Figure 6: Structural filtering.

When the element is closed, the algorithm performs a final set of checks for each variable in $currentMatch[h]$ using information from the sub-tree of e, i.e., $childMatch[h]$ and $descMatch[h]$. For instance, it becomes possible to filter an outgoing edge $\mathcal{X}_{desc}(q, q_c)$ by checking if $q_c \in descMatch[h]$. As the final step, the algorithm updates the matched variables in the parent element's $descMatch[h-1]$ and $childMatch[h-1]$.

The filtering algorithm ensures the following property: if a variable q is not present in $currentMatch[h]$ after e closes, then e cannot bind to q in any tuple in $Q(S)$. This essentially preserves the properties of the $\mathcal{X}2R$ transform and guarantees that the SKETCHER will generate an unbiased estimator. Moreover, the algorithm can be implemented efficiently in practice, as its complexity per element is linear to the number of variables in the query and the number of variables in the query is expected to be small in real-word queries.

5.2 Structural Partitioning

The idea behind structural partitioning is to break S into disjoint sub-streams $S_1, \ldots, S_m$ such that $\text{COUNT}_Q = \sum_{i=1}^{m} \text{COUNT}_Q(S_i)$ and the aggregate of each sub-stream S_i is estimated separately using the sketch-based synopsis. This approach can yield a more accurate approximation compared to the "monolithic" estimator that is based on the sketch of the complete stream.

Partitioning Points. The development of our technique is based on the concept of a continuation S' of S which is defined as any valid XML stream that includes S as a prefix. Intuitively, a continuation S' represents one possibility for the contents of S at a future point in time. Now, let e be an element whose `close` event appears in S and let S_e be the sub-stream that contains the parsing events for the sub-tree of e. We say that e is a *partitioning point* with respect to S and Q if $Q(S') = Q(S_e) \cup Q(S' - S_e)$ for any continuation S' of S. This property essentially states that there is no binding tuple that combines elements inside and outside of S_e, and this holds independently of the future elements that appear in the stream. In turn, this implies that S_e makes a separate contribution to $\text{COUNT}_Q(S')$ for any continuation S'. As an example, consider again the stream and query shown in Figure 1. We can verify that every `s` element is a partitioning point, since it matches the root variable q_1 and the constraints of the query specify that the binding tuples can only contain elements from the sub-tree of q_1. Accordingly, the sub-tree of each `s` element makes a separate contribution to the total count of binding tuples.

Let us now consider a stream S where the last parsing event corresponds to the `close` of a partitioning element e and e does not have any preceding elements. Let S' denote an arbitrary continuation of S. Due to the partitioning property, it follows that $\text{COUNT}_Q(S') = \text{COUNT}_Q(S_e) + \text{COUNT}_Q(S' - S_e)$ for any continuation S'. We observe that the current state of the synopsis is formed solely by the elements in S_e, since e does not have any preceding elements and it is the last element to be closed. Hence, the SKETCHER readily provides an unbiased estimator for $\text{COUNT}_Q(S_e)$. Moreover, a reset[2] of the sketches removes the information on S_e from the synopsis, thus ensuring that the SKETCHER will provide an unbiased estimator of $\text{COUNT}_Q(S' - S_e)$ when the processing of S' is finished. Hence, the estimate of $\text{COUNT}_Q(S')$ can be generated as the sum of two sub-estimates: an estimate of $\text{COUNT}_Q(S_e)$ that is obtained when e is closed, and an estimate of $\text{COUNT}_Q(S' - S_e)$ that is obtained when S' has been processed. The crucial point is that the sub-estimates are obtained using the full resources of the SKETCHER on streams that are smaller than S'. Thus, the $\text{COUNT}_Q(S_e)+\text{COUNT}_Q(S'-S_e)$ estimator has higher accuracy compared to the estimator of $\text{COUNT}_Q(S')$ that is based on a complete sketch of S'.

Of course, the same method can be applied recursively on $S'-S_e$ in order to improve the approximation of $\text{COUNT}_Q(S' - S_e)$. This leads to the idea of a sequence of partition points $e_1, \ldots, e_m$ that contribute an independent count estimate to the estimator for $\text{COUNT}_Q(S')$. In fact, as the following theorem states, each partitioning point that is identified and exploited, provides an opportunity to reduce the overall variance of the estimator.

THEOREM 5.1. *Let Φ be the sketch-based estimator for the continuation stream S' without partitioning, and let Φ' be the sketch-based estimator where the partitioning point e is identified. Let $\Phi' = \Phi_1 + \Phi_2$, where Φ_1 is the sketch-based unbiased estimator for $\text{COUNT}_Q(S_e)$ and Φ_2 is the sketch-based unbiased estimator for $\text{COUNT}_Q(S' - S_e)$. Then, Φ' is an unbiased estimator for $\text{COUNT}_Q(S')$. In addition, Var $\Phi' \leq$ Var Φ.*

Proof: Since Φ_1 and Φ_2 are unbiased estimators, by the linearity of expectation it follows that Φ' is an unbiased estimator for $\text{COUNT}_Q(S')$.

By definition Var $\Phi = \text{E}[(\Phi - \text{E}[\Phi])^2]$. Using the expressions for Φ and E$[\Phi]$ from Equations 2 and 3, respectively, it follows that

$$\text{Var } \Phi = \text{E}[(\sum_{\substack{R_1 \times \cdots \times R_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k})^2].$$

Since Φ_1 and Φ_2 are independent, Var Φ' = Var Φ_1 + Var Φ_2. In a similar manner, we obtain expressions for Var Φ_1 and Var Φ_2, then by substitution we obtain the following expression for Var Φ':

$$\begin{aligned} \text{Var } \Phi' &= \text{E}[(\sum_{\substack{R'_1 \times \cdots \times R'_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k})^2] \\ &+ \text{E}[(\sum_{\substack{R''_1 \times \cdots \times R''_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k})^2]. \end{aligned}$$

Observe that since $R_i = R'_i \cup R''_i$ for $i = 1, \ldots n$ using bag semantics, then it follows that $R'_1 \times \cdots \times R'_n \cup R''_1 \times \cdots \times R''_n \subseteq R_1 \times \cdots \times R_n$. Therefore, we can write the expression for Var Φ as follows where MT_0 represents 0 or more terms:

$$\begin{aligned} \text{Var } \Phi = \text{E}[(&\sum_{\substack{R'_1 \times \cdots \times R'_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k} \\ &+ \sum_{\substack{R''_1 \times \cdots \times R''_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k} + MT_0)^2]. \end{aligned}$$

Note that a partitioning point e is identified at the time that e is closed; therefore, it is possible that $S' - S_e$ is empty. In this case, R''_i for $i = 1, ..., n$ would be empty and MT_0 would represent 0 terms.

We can rewrite the above expression for Var Φ as follows where MT_1 represents the remaining sum of terms after expanding the square:

$$\begin{aligned} \text{Var } \Phi = \text{E}[(&\sum_{\substack{R'_1 \times \cdots \times R'_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k})^2] \\ &+ \text{E}[(\sum_{\substack{R''_1 \times \cdots \times R''_n, \\ \exists k: Y_k \neq X_k}} \prod_{2 \leq k \leq n} \xi^k_{Y_k} \xi^k_{X_k})^2] + \text{E}[(MT_1)]. \end{aligned}$$

Note that

$$\begin{aligned} MT_1 = &MT_0^2 + 2(\Phi_1 - \text{E}[\Phi_1])(\Phi_2 - \text{E}[\Phi_2]) \\ &+ 2(\Phi_1 - \text{E}[\Phi_1])MT_0 + 2(\Phi_2 - \text{E}[\Phi_2])MT_0 \end{aligned}$$

Observe that after fully expanding the squares and MT_1 in the above expression for Var Φ, each term can have a product of at most four different random variables for each family of random variables ξ^k. Thus, since each family of random variables ξ^k is independent for different k and each family is four-wise independent, the expected value of each term after expanding will be either 0 or 1. In addition, the expected value of each term after expanding is determined by values of the attributes in the corresponding tuples. Since the expected value of every term in MT_1 after expanding is either 0 or 1, it follows that Var $\Phi' \leq$ Var Φ. ∎

Algorithm Description. Figure 7 shows the pseudo-code for the structural partitioning algorithm based on the previous method. Recall that the algorithm is invoked after the matches of the element

[2]A reset is achieved by zeroing the atomic sketches and obtaining new ξ families

Procedure $CheckPartition()$
begin
1.**if** $\neg(B_{anc} \vee B_{par} \vee B_{fol} \vee B_{folSib}) \wedge\ precCount[h] = 0$ **then**
2. Accumulate estimate from SKETCHER and reset sketches
3. $descMatch[h-1] \leftarrow childMatch[h-1] \leftarrow \emptyset$
4. $descCount[h-1] \leftarrow descCount[h] \leftarrow 0$
5.**end if**
end

Figure 7: Structural partitioning.

have been finalized in the SIEVE (Figure 5). The algorithm examines the current element e and verifies whether it is a partitioning point with no preceding elements. If so, the SKETCHER accumulates the current estimate, resets the sketches, and sets $descMatch[h-1]$ and $descCount[h-1]$ to indicate the removal of S_e from the state of the synopsis.

The algorithm requires some means to verify that e is indeed a partitioning point. Since the partitioning property involves reasoning about the possible continuations of S, the derivation of a tight condition is likely to be a hard problem. In this paper, we develop a sufficient condition that works well in practice. More concretely, we define a boolean condition $B_{anc} \vee B_{par} \vee B_{fol} \vee B_{folSib}$ that checks whether an element in S_e can participate in a binding tuple with an ancestor, parent, following, or following-sibling element of e respectively. (Preceding elements are not considered as e is assumed to not have any.) The definition of these conditions employs the data structures maintained by the structural filter at the time that the `close` event of e is processed (Figure 6). Their definition is as follows:

$$B_{anc} \equiv \bigvee_{q \in subtreeMatch} q' \in ancMatch[h] \wedge (\mathcal{X}_{desc}(q', q) \vee \mathcal{X}_{anc}(q, q'))$$

$$B_{par} \equiv \bigvee_{q \in currentMatch} q' \in parentMatch \wedge (\mathcal{X}_{child}(q', q) \vee \mathcal{X}_{par}(q, q'))$$

$$B_{fol} \equiv \bigvee_{q \in subtreeMatch} q' \in Q \wedge (\mathcal{X}_{fol}(q, q') \vee \mathcal{X}_{prec}(q', q))$$

$$B_{folSib} \equiv \bigvee_{q \in currentMatch} q' \in Q \wedge\ (\mathcal{X}_{folSib}(q, q') \vee \mathcal{X}_{precSib}(q', q))$$

For instance, B_{anc} checks whether there is a constraint $\mathcal{X}_{desc}(q', q)$ or $\mathcal{X}_{anc}(q, q')$ such that q has a match in S_e and q' has a match to an ancestor e' of e. If no such constraint exists, then there can be no binding tuple that combines an element of S_e and an ancestor of e in the same binding tuple in order to satisfy an ancestor/descendant relationship. More generally, we can formulate the following implication:

PROPOSITION 5.1. *If e does not have any preceding elements and $\neg(B_{anc} \vee B_{par} \vee B_{fol} \vee B_{folSib})$, then e is a partitioning point.*

Proof: By contradiction. Assume that e is not a partitioning point. Since e is not a partitioning point with respect to the XML stream S, there exists a continuation of S' and there exists a binding tuple $\mathbf{t}$ and there exists an edge $(q', q) \in Q$ such that ($t[q] \in S_e$ and $t[q'] \notin S_e$) or ($t[q] \notin S_e$ and $t[q'] \in S_e$). We distinguish the following cases for the axis of (q', q).

Case $\mathcal{X}_{desc}(q', q)$: Assume $t[q] \notin S_e$ and $t[q'] \in S_e$. Since $t[q']$ is the ancestor of $t[q]$, $t[q]$ is in the subtree rooted by $t[q']$. This means that $t[q'] \in S_e$ which is a contradiction; therefore, $t[q] \in S_e$ and $t[q'] \notin S_e$. Since $t[q] \in S_e$, it follows that $q \in subtreeMatch$. Since $\mathcal{X}_{desc}(q', q)$ and $\mathbf{t}$ is a valid binding tuple, then $t[q']$ is an ancestor of $t[q]$ that binds q'; therefore, $ancMatch[h] \neq \emptyset$. Hence, $B_{anc} = true$ which is a contradiction.

Case $\mathcal{X}_{anc}(q', q)$: This is similar to $\mathcal{X}_{desc}(q, q')$.

Case $\mathcal{X}_{child}(q', q)$: Similar to $\mathcal{X}_{desc}(q, q')$, we can show that $t[q] \in S_e$ and $t[q'] \notin S_e$. In addition, since $t[q']$ is the parent of $t[q]$, it must be that $t[q] = e$. From $t[q] = e$, it follows that $q \in currentMatch$. Since $\mathcal{X}_{child}(q', q)$ and $\mathbf{t}$ is a valid binding tuple, it follows that $q' \in parentMatch$. Hence, $B_{anc} = true$ which is a contradiction.

Case $\mathcal{X}_{par}(q', q)$: This is similar to $\mathcal{X}_{child}(q, q')$.

Case $\mathcal{X}_{fol}(q', q)$: Assume that $t[q] \in S_e$ and $t[q'] \notin S_e$. Then $t[q']$ is either an ancestor of $t[q]$ or it follows $t[q]$. It cannot precede $t[q]$ because e does not have any preceding elements. But this means that $\mathcal{X}_{fol}(t[q'], t[q])$ is false and hence $\mathbf{t}$ cannot be a binding tuple. Therefore, $t[q] \notin S_e$ and $t[q'] \in S_e$. Since $t[q'] \in S_e$, it follows that $q' \in subtreeMatch$. In turn, given that q is a variable such that $\mathcal{X}_{fol}(q', q)$, it follows that $B_{fol} = true$ which is a contradiction.

Case $\mathcal{X}_{prec}(q', q)$: This is similar to $\mathcal{X}_{fol}(q, q')$.

Case $\mathcal{X}_{folSib}(q', q)$: Similar to $\mathcal{X}_{fol}(q', q)$ we can show that $t[q'] \in S_e$ and $t[q] \notin S_e$. Given that $t[q]$ is the sibling of $t[q']$, this can happen only if $t[q'] = e$. Since $t[q'] = e$, it follows that $q \in currentMatch$. Moreover, the variable q is such that $\mathcal{X}_{desc}(q, q')$. Hence, $B_{folSib} = true$ which is a contradiction.

Case $\mathcal{X}_{precSib}(q', q)$: This is similar to $\mathcal{X}_{folSib}(q, q')$. ∎

The complexity of evaluating the condition is linear to the number of query variables, which is expected to be low in practice. Hence, structural partitioning can be accomplished efficiently in real-world applications.

Data Set	#Elements	Depth	Type	Size (MB)
XMark	167864	11	Synthetic	10
IMDB	155898	5	Real-Life	7
TreeBank	2437666	35	Real-Life	29

Table 3: Data set characteristics.

6. EXPERIMENTS

This section presents the empirical study that we conducted to evaluate the proposed approximation framework. We employ a prototype implementation of the framework in C++. The SKETCHER is implemented with the sketching technique of Dobra et al.[6], fixing $s_2 = 3$ and setting s_1 according to the total size of the synopsis. We note that the prototype works on unbounded streams (Section 4.1).

6.1 Methodology

Data Sets. Table 3 summarizes the characteristics of the data sets employed in our study. The chosen data sets cover a wide range of properties, e.g., real-life versus synthetic, small versus big, semi-regular versus irregular, and are sufficiently complex to provide an interesting testbed for experimentation. To model a stream, the data sets are accessed through a SAX parser.

Workloads. For each data set, we generate three types of workloads of increasing complexity: *Basic* contains queries that employ only the ancestor/descendant and parent/child axes; *Basic+Sib* augments *Basic* with the preceding-/following-sibling axes; and *Basic+Fol* augments *Basic* with the following/preceding axes. All workloads contain 100 randomly generated tree-patterns of 4 variables. We ensure that each query has at most one occurrence of the child axis so that the workload is sufficiently complex. We only

consider test queries that generate at least one binding tuple (i.e., positive queries), since our techniques yield near-zero estimates for negative queries.

Evaluation Metrics. We measure the performance of an approximation technique in terms of the sanitized relative error of estimation. For a single query Q, the error metric is defined as $|\mathrm{COUNT}_Q - Est| / \max(\mathrm{COUNT}_Q, s)$, where Est is the estimated value of COUNT_Q, and s is a sanity bound that avoids artificially high errors when COUNT_Q is low. Following common practice, we set s to the 10-percentile of true counts in the workload. We have also performed experiments with the absolute error metric $|\mathrm{COUNT}_Q - Est|$, and the symmetric relative error metric $|\mathrm{COUNT}_Q - Est| / \min(\mathrm{COUNT}_Q, Est)$. In all cases, the results remained qualitatively the same as with the sanitized relative error.

We report the error for a given workload using the cumulative frequency distribution (CFD) of the per-query metrics. A point (x, y) in the distribution denotes that y percent of the workload has an error that is less than or equal to x. Thus, an approximation technique A is more accurate than technique B on a specific workload if the CFD of A dominates the CFD of B. We adopt this reporting method as it provides greater detail compared to a single statistic (e.g., average or median) over the per-query metrics.

6.2 Results

Sensitivity Analysis. We first evaluate the accuracy of our technique with respect to the synopsis size, the workload type, and the use of structural sieving. By default, the synopsis size is set to 10KB, *Basic-Sib* is used as the workload, and structural sieving is on. All the experiments that follow employ the synthetic XMark data set.

Figure 8 depicts the CFD of the estimation error as we vary the size of the synopsis. Overall, the results show that the proposed technique can provide accurate estimates for the vast majority of queries in the workload. For instance, the 10KB synopsis results in less than 10% error for 70% of the test queries. Moreover, we observe a clear trend of diminishing errors as more space is allocated to the synopsis. Essentially, increasing the number of atomic estimates reduces the variance of the estimator, which in turn yields more accurate approximations.

Figure 9 shows the effect of structural sieving on the estimation error. The CDF shows clearly that the use of filtering and partitioning yield a significant improvement (close to 5 orders of magnitude) on the vast majority of test queries. Overall, the results validate the effectiveness of structural sieving and basically show that it is essential in obtaining accurate estimates.

Figure 10 depicts the estimation error for the three different types of workload. We note that the results are generated using a smaller version (1MB) of the XMark data set, as we were not able to obtain reliable estimates for the *Basic+Fol* workload on the bigger data set. We observe that our technique has similar high accuracy for the *Basic* and *Basic+Sib* workloads, but the performance deteriorates significantly for *Basic+Fol*. Essentially, the following/preceding axes cancel the ability to perform structural partitioning, as it is always possible to construct a continuation S' such that a binding tuple combines an element in S and a following element in $S' - S$. The power of structural filtering is also reduced, since it is not possible to filter elements along the following/preceding axis. Yet another factor is that the conditions on the full order axes employ larger structural intervals and thus contribute to larger sizes (and in effect, self-join sizes) of the relations of the $\mathcal{X}2R$ transform.

Overall, the results demonstrate that our technique enables accurate approximations for a large class of practical queries. At the same time, it is clear that the following/preceding axes increase significantly the complexity of the problem. One possible solution is to allow the query to constrain the following/sibling axes to a "window" over the XML data tree, e.g., within the sub-tree of elements with a specific tag. This query model may enable a refined definition of structural partitioning that can lead to higher accuracy. Moreover, a constrained following/preceding axis feels more practical for real-world applications, since the unconstrained axis essentially combines elements that are arbitrarily far in the stream. We intend to investigate this idea as part of our future work.

Timing Experiments. The next set of experiments evaluates the performance of our technique in terms of execution time. We employ the same experimental settings as the previous section. The experiments were carried out on an Intel Xeon 3.40GHz CPU with 3GB of RAM, running Fedora Core R5.

Figure 11 shows a breakdown of the average execution time per test query. The breakdown includes the following components: updating the atomic sketch counters (sketching); filtering; partitioning, which includes obtaining estimates at partitioning points and resetting the sketches; and, computing the $\mathcal{X}2R$ transform. The three bars refer to different variants of our technique: NoOpt employs basic sieving without the optimization of Section 4.2; Basic applies the optimization; and, Struct employs structural sieving with the optimization.

We focus first on the last bar that depicts the total execution time of the full technique. The results show that sketching is clearly the dominant component in terms of execution cost. We note that this component can be readily improved by using more sophisticated sketching techniques to implement the SKETCHER [17]. Even so, our prototype achieves a low average processing time of 15.93 microseconds per element, which includes the overhead of the XML parser. This translates roughly to an average throughput of 60K elements/second. Another interesting observation is that the $\mathcal{X}2R$ transform is really efficient to compute, incurring a cost that is less than 1% of the total execution time

Comparing across the different variants, it becomes clear that the optimized maintenance of atomic sketches reduces significantly the total execution time. This optimization reduces the number of sketch-update operations which form the dominant cost factor. A similar observation holds for structural sieving (in particular, structural filtering), even though the improvement is less drastic.

Results on Real-life data sets. We next evaluate the performance of our technique on the real-life data sets. We restrict the synopsis size to 10KB and employ the *Prec-Sib* workload.

Figure 12 shows the CFD of the estimation error on the IMDB and TreeBank data sets. (We repeat the results of the XMark data set for comparison.) As shown, our technique enables estimates with low error for the majority of test queries. Even for the TreeBank data set, which is known as a difficult test case for summarization, the estimation error is less than 20% for 60% of the test queries. This level of accuracy is significant if we take into account the complexity of the data and the workload (twig queries with a combination of XPath axes). Moreover, it is important to note that our technique supports probabilistic error bounds that allow the user to gauge the accuracy of the estimate.

Comparison to SketchTree. We compare our approach to the SketchTree summary of Rao and Moon [16]. Due to the limitations of SketchTree, the comparison is limited to queries that employ solely the child and following-sibling axis. We employ the TreeBank data set on which SketchTree has been tested successfully, and we form a test workload by picking at random 100 queries

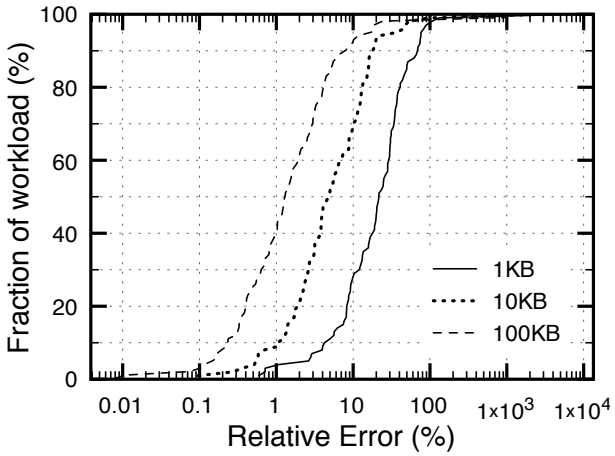


Figure 8: Effect of synopsis size on estimation error.

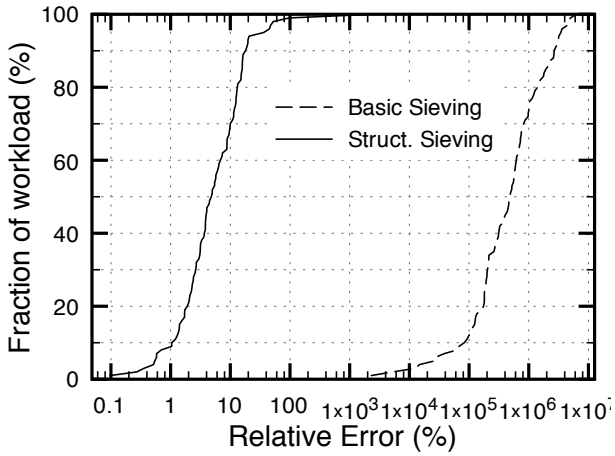


Figure 9: Effect of sieving on estimation error.

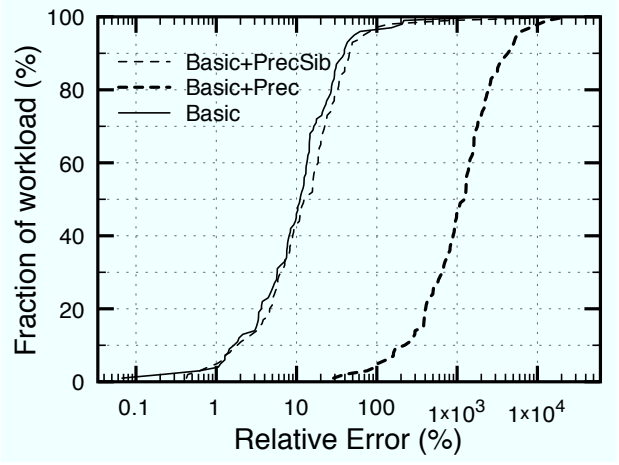


Figure 10: Effect of workload-type on estimation error.

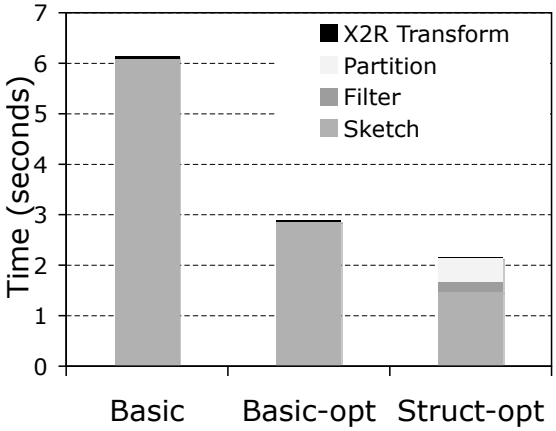


Figure 11: Execution time breakdown.

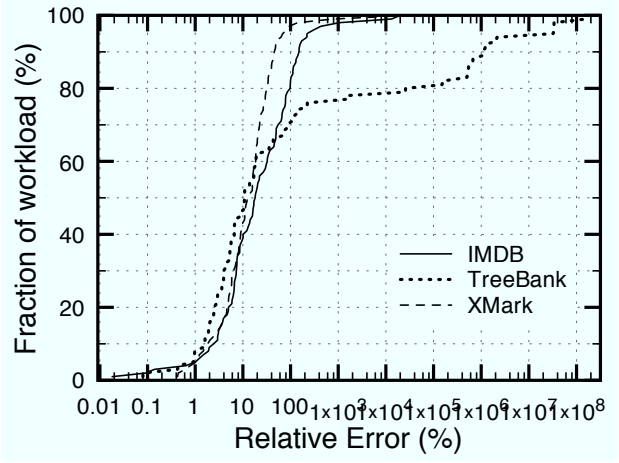


Figure 12Estimation error on TreeBank and IMDB.

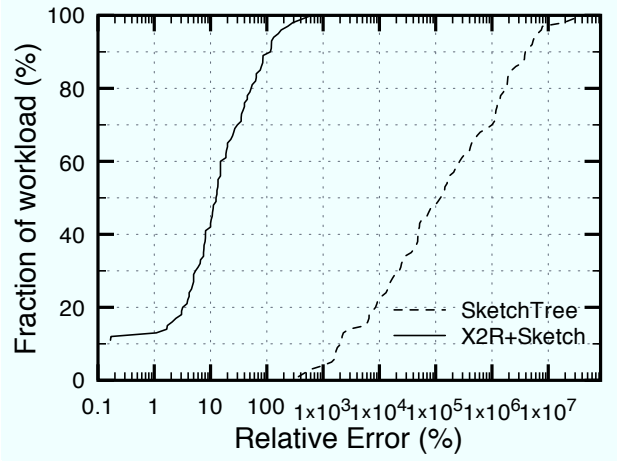


Figure 13: Comparison against SketchTree.

from a workload provided by the authors[3]. Following the published study, the construction parameters of the SketchTree are set to $s_1 = 50$, $s_2 = 7$, and $top - k = 50$. This yields a synopsis that occupies 460KB and that can estimate the count of any tree-pattern query with child and following-sibling constraints. Since our technique works on a per-query basis, we restrict the size of our sketch synopsis to 1% of the total size of the SketchTree summary.

Figure 13 shows the estimation error of the two techniques on the TreeBank data set. The results clearly demonstrate that our sketch-based synopsis enables significantly lower estimation errors compared to SketchTree. Essentially, SketchTree can estimate the count of any tree-pattern up to a specific size and, in that sense, the statistical information that it stores is "diluted" over a very big set of queries. The sketching technique that we propose is targeted towards a specific continuous query that is installed over the stream, and thus irrelevant information can be filtered.

Comparison to MXQuery. Finally, we compare our technique against MXQuery [2], a lightweight XQuery engine that supports streaming XML data and has a low memory footprint. We focus the experiment on the memory consumed by query processing, since this forms the primary motivation behind our sketch-based technique. We do not report on the CPU time requirements because we could not obtain reliable measurements using the statistics supplied by MXQuery itself. (We note that the CPU usage reported by the Linux *time* command shows that we outperform MXQuery in this respect.)

Our test was conducted on an Intel Xeon 3.40GHz CPU with 3GB of RAM, running Fedora Core R5. MXQuery was executed using J2RE v1.5.0. We used the XMark dataset, and we restricted the comparison to the *Basic* workload because MXQuery does not support the order axes. Since MXQuery does not report its memory consumption, we resorted to the Linux *top* command in order to measure the maximum data resident set size (data + stack size) required per query for each technique. We averaged the metric over 10 trials for each query.

The results show that, on the average, our sketch-based approximation requires 618KB of memory compared to 962,757KB for the full query evaluation performed by MXQuery. At the same time, our technique provides estimates of high accuracy, following the error trends shown in Figure 10 for the *Basic* workload. We note that the memory measurement for MXQuery includes the overhead of the JVM, but this alone does not account the large difference of 3 orders of magnitude. Overall, the results demonstrate the efficacy of our technique as a substitute for full query evaluation in environments where resources are limited and an approximate answer is sufficient.

7. CONCLUSIONS

In this paper, we present a novel technique for approximate query answering over XML streams. Our technique advances the state-of-art as it provides provable approximation guarantees for the class of aggregate XML queries that employ the ordered XML model and recursive structural constraints. The foundation of our approach is the $\mathcal{X}2R$ transform that essentially reduces an aggregate XML query to a relational query with equi-join predicates. This rewriting enables us to develop a XML stream synopsis by leveraging existing relational techniques. We enhance this synopsis with structural filtering and partitioning, two variance-reduction techniques that improve significantly the accuracy of approximation. Experiments on real-life and synthetic data sets have verified the effectiveness of our approach.

[3]We are grateful to the authors for providing us with their source-code and experimental data, and for advising us in the evaluation of SketchTree.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *STOC*, 1996.

[2] I. Botan, D. Kossmann, P.M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius. Extending XQuery with window functions. In *VLDB*, 2007.

[3] J. Clark and S. DeRose. XML Path Language (XPath), Version 2.0. W3C Recommendation, 2007.

[4] A. Das, J. Gehrke, and M. Riedewald. Approximation techniques for spatial data. In *SIGMOD*, 2004.

[5] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *TODS*, 28(4), 2003.

[6] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *SIGMOD*, 2002.

[7] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *SIGMOD*, 2002.

[8] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing data-stream join aggregates using skimmed sketches. *EDBT*, 2004.

[9] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. How to Summarize the Universe: Dynamic Maintenance of Quantiles. In *VLDB*, 2002.

[10] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *SIGMOD*, 2003.

[11] V. Josifovski, M. Fontoura, and A. Barta. Querying xml streams. *The VLDB Journal*, 14(2), 2005.

[12] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: An On-Line Self-Tuning Markov Histogram for XML Path Selectivity Estimation. In *VLDB*, 2002.

[13] V. Mayorga and N. Polyzotis. Sketch-based summarization of ordered xml streams. Technical report, UC Santa Cruz, 2008.

[14] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate xml query answers. In *SIGMOD*, 2004.

[15] M. Ramanath, L. Zhang, J. Freire, and J. R. Haritsa. Imax: Incremental maintenance of schema-based xml statistics. In *ICDE*, 2005.

[16] P. Rao and B. Moon. Sketchtree: Approximate tree pattern counts over streaming labeled trees. In *ICDE*, 2006.

[17] F. Rusu and A. Dobra. Fast range-summable random variables for efficient aggregate estimation. In *SIGMOD*, 2006.

[18] Z. Vagena, N. Koudas, D. Srivastava, and V. J. Tsotras. Answering order-based queries over xml data. In *WWW*, 2005.

[19] H. L. J. X. Y. Wei Wang, Haifeng Jiang. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB*, 2004.

[20] Y. Wu, J. M. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *EDBT*, 2002.

[21] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 30(2), 2001.

[22] N. Zhang, M. T. Ozsu, A. Aboulnaga, and I. F. Ilyas. Xseed: Accurate and fast cardinality estimation for xpath queries. In *ICDE*, 2006.