

**XML STRUCTURE-BASED CLUSTERING AND ITS APPLICATION
IN SELECTIVITY ESTIMATION**

BY
Ahmad Faisal Barradah

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In
COMPUTER SCIENCE

December 2013

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

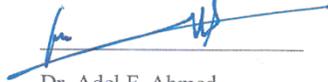
DEANSHIP OF GRADUATE STUDIES

This thesis, written by Ahmad Faisal Barradah under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

Sayed 24/12/2013

Dr. El-Sayed M. El-Alfy
(Advisor)



Dr. Adel F. Ahmed
Department Chairman



Dr. Salahadin A. Mohammed
(Co-Advisor)



Dr. Salam A. Zummo
Dean of Graduate Studies



Dr. Muhammed S. Al-Mulhem
(Member)

6/1/14

Date





Dr. Moataz Ahmed
(Member)



Dr. Sajjad Mahmood
(Member)

© Ahmad Faisal Barradah

2013

Dedication

To My Beloved Parents

ACKNOWLEDGMENTS

First and foremost, all praises and thanks be to Allah Almighty for all his blessings and for giving me the strength, passion, and knowledge to complete this work.

Thereafter, I would like to express my appreciation to King Fahd University of Petroleum & Minerals (KFUPM) and Saudi Aramco EXPEC Computer Center (ECC) for their support during this work. I would also like to express my deepest gratitude to my thesis advisors Dr. El-Sayed M. El-Alfy and Dr. Salahadin A. Mohammed for their continuous motivation, encouragement, untiring efforts to guide me through this work, inspiring ideas to improve this work, and for dedicating their valuable time throughout the different phases of this research.

I would also like to acknowledge the committee members Dr. Muhammed S. Al-Mulhem, Dr. Moataz Ahmed, and Dr. Sajjad Mahmood for their sincere and valuable comments to enhance this research. I would also like to thank Dr. Cheng Luo at Coppin State University for providing the code of the Sampling algorithm.

Finally, deepest and utmost gratitude and appreciation go to my parents; Mr. Faisal Hashim Barradah and Mrs. Samia Hussain Barradah and also to my brothers, sisters, and friends for their love, prayers, encouragement, and endless support.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES	IX
LIST OF FIGURES	X
ABSTRACT	XIII
ABSTRACT (ARABIC)	XV
1. CHAPTER 1 INTRODUCTION	1
1.1. Motivation & Problem Definition	2
1.2. Thesis Objectives	5
1.3. Thesis Outline	6
2. CHAPTER 2 BACKGROUND	7
2.1. XML Model	8
2.1.1. Range-Encoding Scheme	9
2.1.2. Prime-Number Labeling Scheme	9
2.2. XML Query & Query Selectivity Count	12
2.2.1. XPath	12

2.2.2.	XQuery.....	13
2.2.3.	Types of XML Queries.....	14
2.2.4.	Query Selectivity Count.....	14
3.	CHAPTER 3 RELATED WORK.....	17
3.1.	Synopsis-Based Approaches	18
3.2.	Histogram-Based Approaches.....	27
3.3.	Statistical Approaches	30
4.	CHAPTER 4 XML STRUCTURE-BASED SUMMERIZATION.....	33
4.1.	Summarization Based on Prime-Number Labeling.....	33
4.1.1.	SynopGenPrime Preliminaries.....	33
4.1.2.	Construction	35
4.2.	Summarization Using Fingerprinting (<i>SynopGen</i>).....	41
4.2.1.	Construction	42
4.2.2.	Inner Node Labeling Scheme.....	43
4.2.3.	Selection of the Parameters K , B and M	45
4.2.4.	Implementation of the Summary-Tree	49
5.	CHAPTER 5 SELECTIVITY COUNT ESTIMATION.....	51
5.1.	Selectivity Count of Linear Queries.....	52
5.2.	Selectivity Count of Existential Twig Queries	54
5.3.	Selectivity Count of Regular Twig Queries	54

6. CHAPTER 6 PERFORMANCE STUDY	57
6.1. Experimental Settings.....	57
6.2. Summary-Tree Generation Time	58
6.3. Estimation Error Rate and Storage Size.....	61
7. CHAPTER 7 HANDELING STORAGE LIMITATION.....	70
7.1. Pruning the Summary-Tree.....	70
7.2. Selectivity Count Estimation With Node-Count Ratio	71
7.2.1. Experiments	73
7.3. Statistical Approach for Selectivity Count Estimation	74
7.3.1. Regular Twig and Linear Path Queries	74
7.3.2. Existential Twig Queries	75
7.3.3. Experiments	77
7.4. Hybrid Approach for Selectivity Count Estimation	78
7.4.1. Experiments	81
8. CHAPTER 8 CONCLUSION AND FUTURE WORK.....	95
REFERENCES.....	97
CURRICULUM VITAE.....	103

LIST OF TABLES

Table 1 Selectivity count estimation techniques.....	19
Table 2: Summary-tables implementation of the summary-tree in Figure 17	50
Table 3: Some characteristics of the adopted datasets	58
Table 4: Summary-tree generation times and number of collisions for different datasets and various approaches of selecting K	60
Table 5: Sample twig queries.....	65
Table 6: Regular twig estimates for the queries in Table 5	65
Table 7: Summary generation time.....	67
Table 8: Sample queries and the Hybrid results on the XMark	93
Table 9: Sample twig queries and TreeSketch results on the Uniprot at SSR 0.01.....	93

LIST OF FIGURES

Figure 1: Query estimation importance: a) XML tree, b) Path query, and c) Twig query . 4

Figure 2: Example of an XML document 8

Figure 3: Example of range encoding scheme a) XML tree with the range encoding, b)
Simple path parent-child query, and c) Simple path ancestor-descendant query
..... 10

Figure 4: A tree labeled with the bottom-up prime-number labeling scheme 11

Figure 5: A tree labeled with the top-down prime-number labeling scheme 11

Figure 6: a) XML tree, and b) Twig query and its corresponding XPath expression..... 16

Figure 7: a) XML tree with the counts of nodes at the edges, b) XSketch graph for the
tree in a, and c) edge distribution histograms 21

Figure 8: Count stable TreeSketch graph for the XML tree in Figure 7(a) 22

Figure 9: Path tree example: a) Sample path tree, and b) Sibling* summarization..... 23

Figure 10: a) XML tree, b) Encoding table, c) Bit-Seq table, and d) PathID-freq table .. 25

Figure 11: XSeed example a) XML tree, and b) XSeed kernel 26

Figure 12: Example of a path count table and its corresponding bloom histogram 29

Figure 13: XML tree with the corresponding statistics 32

Figure 14: A sample data tree for an XML document 35

Figure 15: a) XML summary tree of the document, b) Leaf nodes and their prime-number
labels, and c) Inner nodes and their labels 36

Figure 16: Two siblings with the same IDs and different structures 42

Figure 17: The summary tree of the XML data tree in Figure 14..... 50

Figure 18: Selectivity count of a linear query.....	53
Figure 19: Selectivity count of an existential twig query	55
Figure 20: a) A twig query, and b) Sub-trees that match the twig query.....	56
Figure 21: Count of elements having a given number of distinct children.....	60
Figure 22: Impact of randomly selecting K in range 3 to 50 on the summary-tree generation time for the proposed approach.....	61
Figure 23: ER for proposed, Sampling, & TreeSketch on linear queries	67
Figure 24: ER for proposed, Sampling, & TreeSketch on existential twig queries	68
Figure 25: ER for proposed, Sampling, & Treesketch on regular twig queries.....	68
Figure 26: Required storage: a) Comparison of SSR for <i>SynopTech</i> , Sampling, and TreeSketch algorithms, and b) Effect of XMark dataset size on the SSR for <i>SynopTech</i> algorithm	69
Figure 27: P-C & A-D error rates for the node-factor method on XMark with different summary sizes.....	73
Figure 28: Error rates for the extended statistical approach	78
Figure 29: P-C & A-D error rates on XMark using summary-delta.....	82
Figure 30: Gen time for Sampling, TreeSketch and Hybrid approaches on XMark	83
Figure 31: Gen time for Sampling, TreeSketch and Hybrid approaches on Uniprot (** TreeSketch needed only SSR of 0.5 to store the summary)	83
Figure 32: Gen time for Sampling, TreeSketch & Hybrid on DBLP & Ssplays	84
Figure 33: Overall error rates for all queries on DBLP and Ssplays	86
Figure 34: P-C & A-D queries error rates for Sampling, TreeSketch and Hybrid (query- delta) on XMark at SSR 1.7.....	90

Figure 35: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on XMark at <i>SSR</i> 0.08.....	90
Figure 36: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on XMark at <i>SSR</i> 0.02.....	91
Figure 37: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at <i>SSR</i> 0.7.....	91
Figure 38: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at <i>SSR</i> 0.07.....	92
Figure 39: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at <i>SSR</i> 0.01.....	92
Figure 40: Overall error rates for all queries on XMark and Uniprot.....	93

ABSTRACT

Full Name : Ahmad Faisal Barradah
Thesis Title : XML Structure-Based Clustering And Its Application In Selectivity Estimation
Major Field : Master in Computer Science
Date of Degree : December 2013

With the increasing popularity of XML and database applications, the demand for efficient query processing is becoming very essential. The performance of XML query optimizers depend heavily on selectivity count estimation to choose the best query execution plan. Most of the existing estimators address the problem of linear path and existential twig query selectivity count estimation while very few address the problem of regular twig query selectivity count estimation. In this work, we propose and evaluate a general selectivity count estimator based on a structural synopsis called, *SynopTech*, that can estimate the selectivity counts for the three query types. We also propose two novel approaches to generate structural summaries of XML data trees which can be used by *SynopTech* for selectivity count estimation. The main idea of the first summarization approach is to use a fingerprinting function to label nodes in the data tree and cluster similar sub-trees to generate a summary tree. The second approach is based on clustering the nodes using the prime-number labeling scheme to generate the summary tree. The experimental results showed very low error rates by the proposed approach for XML documents in four benchmark datasets with different structural characteristics including non-uniform documents and multi-level queries. Comparing with the Sampling algorithm and TreeSketch, two state-of-the-art algorithms for selectivity count estimation,

SynopTech achieved lower selectivity count estimation error rates on most datasets, yet with very low memory budget. For example, for linear and existential queries, *SynopTech* had perfect estimations whereas the Sampling algorithm had an overall error rate of more than 85%. For regular twig queries, *SynopTech* had a maximum error rate of 0.8% whereas the TreeSketch algorithm had more than 15% on some datasets. Moreover, we present a scalable hybrid approach for selectivity count estimation by combining a statistical technique with *SynopTech*. This hybrid approach can work under limited storage budget but at the expense of lowering its estimation accuracy.

ملخص الرسالة

الاسم الكامل: أحمد فيصل براده

عنوان الرسالة: تقسيم شجرات XML بنيويا وتطبيقاته في تقدير الانتقائية الاستعلامات

التخصص: علوم الحاسب

تاريخ الدرجة العلمية: ديسمبر 2013

مع ازدياد شعبية تطبيقات XML وقواعد البيانات، أصبحت معالجة الاستعلامات بكفاءة ضرورية جدا. ويعتمد أداء معالجات الاستعلامات لشجرات XML بشكل كبير على تقدير الانتقائية لاختيار أفضل خطة لتنفيذ الاستعلام. معظم الحلول الموجودة تعالج تقدير الانتقائية للاستعلام عن المسارات الخطية والاستعلام عن المسارات الغصينية الوجودية بينما يتطرق عدد قليل منها للمسارات الغصينية العادية. نقترح في هذه الدراسة مقدر انتقائية عام يمكنه تقدير الانتقائية لأنواع الاستعلامات الثلاث ويعتمد على التلخيص البنيوي لشجرة XML باستخدام دالة بصمة (fingerprint) أو الأعداد الأولية لترقيم أوراق شجرة البيانات وتجميع الأشجار الفرعية المتماثلة لتوليد شجرة التلخيص. وتم تقييم الطريقة المقترحة ومقارنتها مع خوارزميات أخرى على قواعد بيانات ذات خصائص مختلفة ومجموعات مختلفة من الاستعلامات متعددة المستويات. وقد أظهرت النتائج التجريبية معدلات خطأ منخفضة جدا للطريقة المقترحة مقارنة بالطرق الأخرى. ولزيادة مرونة الطريقة المقترحة للتعامل مع الحالات التي فيها قيود على ذاكرة التخزين تم اختصار شجرة التلخيص مما أدى إلى انخفاض دقة التقدير، ولتعويض بعض الفقد الناتج تم دمجها مع أسلوب إحصائي لتحسين الأداء نسبيا.

CHAPTER 1

INTRODUCTION

The eXtensible Markup Language (XML) [1] is becoming increasingly popular as a document formatting standard in various applications especially in the World-Wide Web (WWW). It is also a de facto format for data exchange among heterogeneous systems. As a result, vast amount of XML data is available and the demand for an efficient online query processing is growing by the day. Abundant research has recently been directed towards building query optimizers for XML database management systems as [2 – 8, 54]. Choosing the best possible query execution plan is what database query optimizers attempt to achieve. Like relational query optimizers, XML query optimizers use selectivity count estimators to estimate the size of the intermediate results to be generated by a query execution plan. An efficient selectivity count estimator must have low CPU cost, low memory cost, and low estimation error rate. To achieve that, a selectivity count estimator often doesn't run against the source document, but it uses a summary structure of the source document or some statistics generated from it.

XML queries are classified into linear (path) queries and twig queries. Twig queries are classified further into regular and existential since their selectivity count estimations are computed differently [9]. A number of query estimators have been proposed in the literature but they mostly focus on path queries and/or existential twig queries. Existing linear query selectivity count estimators can have acceptable results. However, the

selectivity count estimation techniques of twig queries, especially regular twig queries, are few and can suffer from high error rate especially for extended queries with multiple levels or when the source XML document is non-uniform [10].

The main goal of this thesis is to develop a synopsis-based technique for XML query estimation that can work well under various structural characteristics of XML documents even with complicated queries. Our approach is based on summarizing the XML data tree by grouping similar structures. We propose two summarization techniques: the first one is based on the prime-number labeling scheme and the second is inspired by the work conducted on string pattern matching using fingerprinting. The generated summary tree by either technique is then used for query estimation. We also propose a hybrid approach that extends the proposed summarization technique with a statistical method to manage the size of the generated summary tree.

1.1. Motivation & Problem Definition

Selectivity count estimation plays a major role in any query optimization model. It is an essential component that enables the optimizer to find the best possible execution plan that reduces the query execution time. In addition, a vast amount of XML data is available in the web and the demand is rising for efficient online query processing such as the Niagara system presented in [11]. For this kind of applications, the selectivity count estimation might be of great interest to the user because it shows whether the query needs refining before returning the full set of results matching his query.

XML query selectivity count estimation is the process of computing the expected intermediate number of nodes in an XML document that match the given query. This

problem is challenging and demands an efficient solution methodology. Figure 1 is an example where selecting an efficient query plan is crucial and may save a great deal of the execution time. The XML document in Figure 1(a) has a single type *C* node which has more than 100,001 children of type *B* and only one of them has a child of type *A*. Assume we want to estimate the number of type *A* nodes that have types *B* and *C* as their ancestors as shown in the query in Figure 1(b). If *B* is joined with *A* first, one node will be retrieved as an intermediate result before it is joined with *C*. On the other hand, if *C* is joined with *B* first, then 100,001 nodes will be produced and joined with type *A* nodes. Also, consider the twig query in Figure 1(c). If this is a regular twig query, then the query processor has to retrieve all *D* and *F* combinations that have types *B* and *E* as their ancestors. If the processor tries to retrieve the twig part of the query first (i.e. $B[/D][/F]$), more than 10^{10} nodes will be retrieved as intermediate results. This is because the type *C* node has 100,000 twigs rooted at type *B* and each twig has 100,000 combinations of node types *D* and *F*. In addition the type *E* node has a single twig rooted at type *B* which has 2000 combinations of types *D* and *F*. On the other hand, if the processor tries to retrieve the linear part of the query first (i.e. E/B), a single *B* node will be retrieved as an intermediate result before the 2000 combinations of *D* and *F* are retrieved. The job of the selectivity count estimator in that case is to provide the query optimizer with enough information about the counts of both the linear and the twig parts of the query in order to choose the optimum query plan (the plan with the lowest storage and/or CPU cost). In other words, all node statistics and estimates should be efficiently maintained and made available for the query optimizer for the purpose of making accurate decisions when selecting a query plan. From the above example, we can see that estimating the selectivity

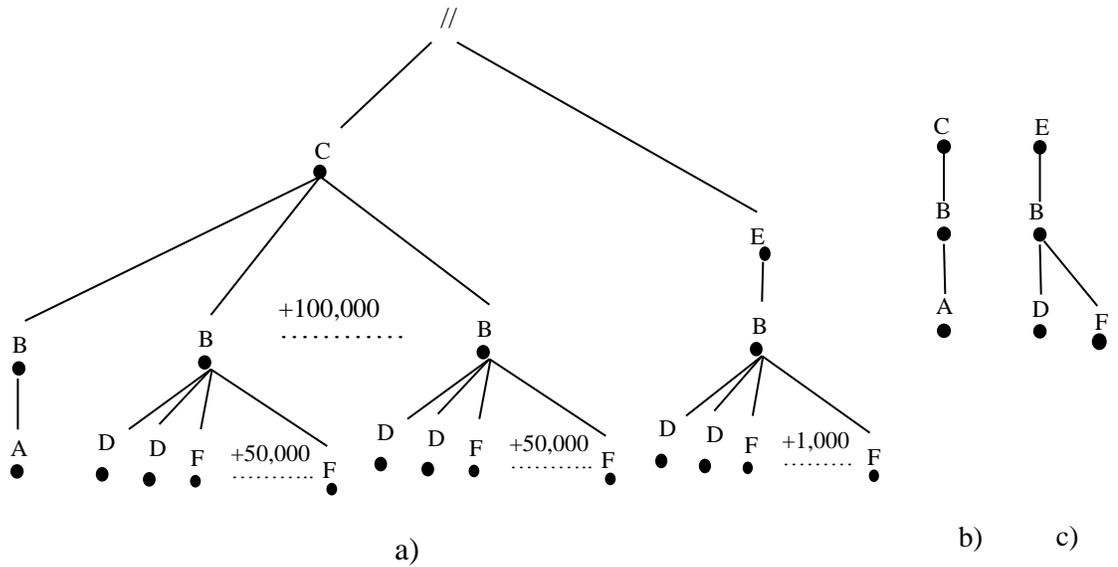


Figure 1: Query estimation importance: a) XML tree, b) Path query, and c) Twig query

counts of regular twig queries can save a great deal of execution time and storage requirements. Since a few estimators in the literature are capable of estimating the counts of regular twig queries, we conducted our research to develop a general estimator that can provide more accurate estimates for regular twig queries in addition to existential twig and linear queries.

That being said, the following are some of the important qualities and characteristics that need to be preserved and addressed when implementing a selectivity count estimation model [9]:

- **Space efficiency:** the algorithm should use minimal storage for the model and its data.
- **Generality:** it can be defined as the capability of the algorithm to handle different types of queries and provide estimates for them. This is indicated by supporting both

path and twig queries that target the most common structures or XML axes (e.g. parent/child, ancestor//decendent..., etc.).

- **Estimation accuracy:** which is normally measured by calculating the error rate produced by the estimator.
- **Time Efficiency:** this can include the summary generation time, the estimation time, and the overall time.

1.2. Thesis Objectives

In this thesis, we propose a model for selectivity count estimation that efficiently addresses the characteristics mentioned in the previous section. The main objective of our research is to provide a framework for more efficient and accurate selectivity count estimation of XML queries including linear, existential and regular twig queries. To achieve this objective, we will proceed as follows:

- 1) **Survey of the literature:** an extensive review of the existing techniques on XML selectivity count estimation in order to identify their strengths and shortcomings.
- 2) **Design & implementation of new algorithms:** design new algorithms to summarize XML trees and estimate the selectivity counts of various types of queries using the summary structure. Also, implement a scalable hybrid approach for selectivity count estimation to handle the memory budget constraints.
- 3) **Performance study:** carefully study the performance of the implemented algorithms using real and synthetic data. Also, the algorithms and the overall framework will be evaluated against some evaluation criteria such as storage requirements and estimation accuracy.

- 4) **Comparison:** compare the performance of the proposed model with existing state-of-the-art selectivity count estimation techniques.

1.3. Thesis Outline

The rest of this thesis is as follows: Chapter 2 gives a general background of the XML model and the XML query processing. The related work is reviewed in Chapter 3. The proposed summarization techniques and the selectivity count estimation algorithm are explained in details in chapters 4 and 5, respectively. Chapter 6 includes the performance study of our approach and the comparisons with other approaches. Chapter 7 introduces a hybrid approach for selectivity count estimation with experimental results. Finally, Chapter 8 concludes the thesis and outlines the future work.

CHAPTER 2

BACKGROUND

XML stands for eXtensible Markup Language [1]. It has the ability to represent structured, semi-structured, and completely unstructured data. Figure 2 presents an example of an XML document which is a sample taken from the Ssplays dataset [52].

The main components of an XML document are:

- **Elements:** An element in the XML document is represented by a start tag “<element>” and an end tag “</element>”. An element can contain other elements, text, or attributes. For instance, in Figure 2, every instance of the TITLE element is enclosed between the start tag <TITLE> and the end tag </TITLE>. Also, each instance of the TITLE element contains text such as “The Tragedy of Antony and Cleopatra”. The instances of the PGROUP element contain two other element types, namely PERSONA and GRPDESCR.
- **Attributes:** These are included in the element’s start tag and they provide additional information about the element. For example, the PGROUP element instances have an attribute called “id” in their start tags : <PGROUP id=”001”> and <PGROUP id=”002”>.
- **Values:** They represent the data enclosed by the start and end tags of the element and that are not other elements. For example, the value of the first PERSONAE instance is the text “MARK ANTONY”. Attributes also have values, e.g. the value of the first

```

<SSPLAYS>
<PLAY>
  <TITLE>The Tragedy of Antony and Cleopatra</TITLE>
  <PERSONAE>
  <TITLE>Dramatis PERSONAE</TITLE>
  <PGROUP id="001">
    <PERSONA>MARK ANTONY</PERSONA>
    <PERSONA>OCTAVIUS CAESAR</PERSONA>
    <PERSONA>M.AEMILIUS LEPIDUS</PERSONA>
    <PERSONA>POMPEIUS</PERSONA>
    <GRPDESCR>triumvirs.</GRPDESCR>
  </PGROUP>
  <PGROUP id="002">
    <PERSONA>DOMITIUS ENOBARBUS</PERSONA>
    <PERSONA> VENTIDIUS</PERSONA>
    <PERSONA>EROS</PERSONA>
    <PERSONA>SCARUS</PERSONA>
    <PERSONA>DERCETAS</PERSONA>
    <PERSONA>DEMETRIUS</PERSONA>
    <PERSONA>PHILO</PERSONA>
    <GRPDESCR>friends to Antony.</GRPDESCR>
  </PGROUP>
  </PERSONAE>
</PLAY>
</SSPLAYS>

```

Figure 2: Example of an XML document

“id” attribute is “001”. The types of the values as well as the structure of the XML document can be described using data definition languages such as the DTD [12] or XML Schema [13].

2.1. XML Model

The XML document is usually modeled with a labeled tree structure where every node is associated with a type or a value and a label. An edge between two nodes in the tree

represents a parent-child (P-C) relationship between the two nodes. The labels on the nodes normally describe the ordered position of the nodes in the data tree. Furthermore, the labels in some labeling schemes can be used to describe some structural relationships between arbitrary nodes in the tree, such as, parent-child (P-C), ancestor-descendant (A-D), and so on. In what follows, we present some common labeling schemes.

2.1.1. Range-Encoding Scheme

This scheme is also known as containment or interval-based labeling scheme [14]. In the range encoding scheme, every node is labeled with a 3-tuple of integers ($start$, end , $level$). For any two nodes x and y in the tree, x is an ancestor of y iff $x.start < y.start$ and $x.end > y.end$. Moreover, if $x.level = y.level - 1$ then x is the parent of y . Also, for an inner node x , $x.start < x.end$ and for a leaf node x , $x.start = x.end$. Figure 3(a) shows an XML tree labeled with the range encoding scheme. This approach can identify A-D relationships but it requires re-labeling the tree after any update (such as adding or deleting nodes). To avoid re-labeling, Amagasa et al. [55] proposed a technique using float-point values for start and end but it also has limitations due to the finite word length of the computer.

2.1.2. Prime-Number Labeling Scheme

The prime-number labeling scheme was first introduced by Wu et al. [15]. They proposed bottom-up and top-down approaches to label a given XML data tree.

A) **Bottom-up Scheme:** In the bottom-up approach, the XML data tree is scanned using post-order traversal and the leaf nodes are labeled with prime numbers while the inner nodes are labeled with the products of their children's labels. The main characteristic of this approach is that it facilitates the identification of the ancestor-descendant

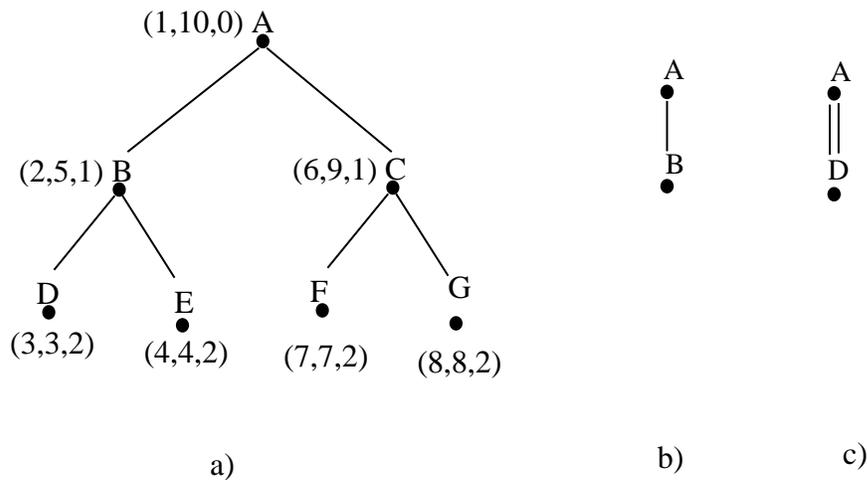


Figure 3: Example of range encoding scheme a) XML tree with the range encoding, b) Simple path parent-child query, and c) Simple path ancestor-descendant query

relationships between any two nodes with a single comparison given their labels using the divisibility property of prime numbers. In other words, for any two nodes x and y , x is an ancestor of y if and only if $label(x) \bmod label(y) = 0$ [15]. Figure 4 shows an example of a tree labeled with the bottom-up prime-number labeling scheme. The problem with this approach is that it can result in very large labels for big XML documents.

B) **Top-down Scheme:** In order to reduce the size of the labels, Wu et al. [15] also introduced the top down approach where the label of a node is the product of its self-label (the next available prime number) and its ancestors' labels in the pre-order traversal. Figure 5 shows an example of the top-down approach.

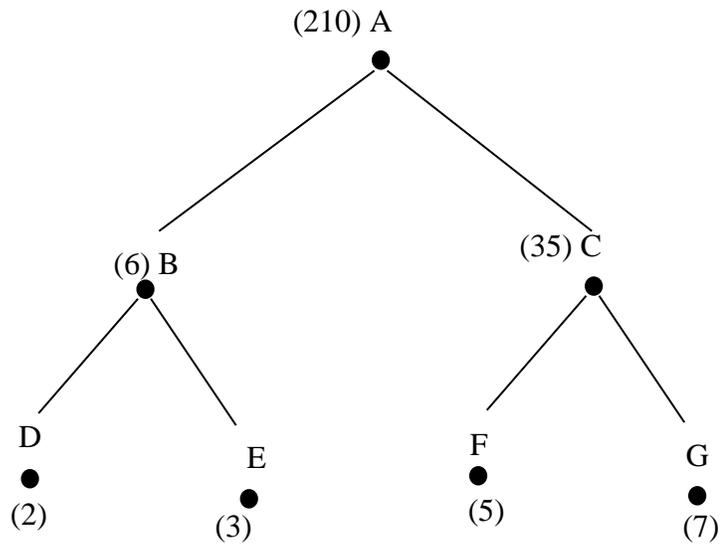


Figure 4: A tree labeled with the bottom-up prime-number labeling scheme

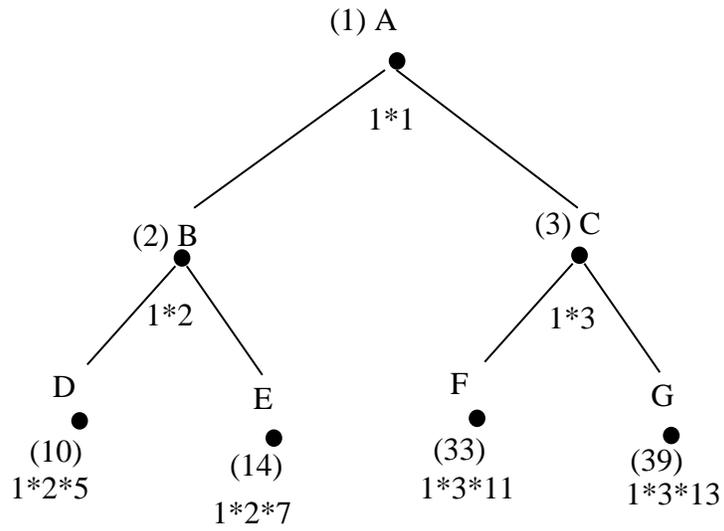


Figure 5: A tree labeled with the top-down prime-number labeling scheme

2.2. XML Query & Query Selectivity Count

The XML query is usually represented using a tree. The set of nodes in the query tree is a subset of the set of nodes in the data tree. Furthermore, the edges in the query tree are represented by either double forward slashes “//” between node types to reflect the ancestor-descendant relationship between the nodes, or a single forward slash “/” between node types to reflect the parent-child relationship between the nodes.

Several query languages exist in the literature such as XPath [16], XQuery [17], Lorel [18] and Quilt [19]. The main purpose of the query languages is to find and retrieve the matches of the query tree in the source XML data tree. Perhaps the most common query languages are the XPath and XQuery and in what follows we give a brief description of each.

2.2.1. XPath

XPath is an XML query language [16]. Its syntax is similar to the syntax used in the UNIX operating system to access files and directories. A simple XPath expression can be of the form $//n_1/n_2..n_i$ where n is a node type and the forward slashes are used to describe the structural relationships between the node types (i.e. “/” for parent-child and “//” for ancestor-descendant relationships). For instance, the simple path expression $(//book/title)$ will return all the “title” elements that have a parent of type “book”. The wildcard “*” can also be used in the XPath expressions to indicate any node type. Also, XPath allows both structural conditions and value predicates using brackets. Structural conditions can represent branching paths. For example, the path expression

//book[author]/title involves two paths, //book/author and //book/title, and will retrieve all the title elements that have an author element as their sibling and the element book as their parent.

2.2.2. XQuery

The XQuery is a functional language used to query XML documents. The XQuery expressions usually have the FLWOR (For, Let, Where, Order, Return) form [17]. The FLWOR expression allows the user to manipulate the result of the query. The “For” clause is composed of one or more path expressions where each expression is bound to a variable. Each variable iterates through the nodes returned by the expression it is bound to. The “Let” clause is used to create variables and bind them to the results from the “For” clause. The “Where” clause is used to specify conditions over the variables in the “Let” and “For” clauses in order to filter the results. The “Order” clause is used to sort the results based on a variable defined in the “For” or “Let” clauses. Finally, the “Return” clause is used to manipulate the structure of the results and is the only mandatory clause.

The following is an example of an XQuery expression:

```
For $x IN document("SSplays.xml")//PLAY
Let $title := $x/TITLE
Where count($x/ACT) > 3
Order by $title
Return $title
```

The above query returns all the titles for the plays that have more than 3 acts ordered by the title.

2.2.3. Types of XML Queries

There are several types of XML queries. In what follows, we provide a classification of XML queries similar to the one presented in [9].

- **Linear Path Queries (LP):** These are single path queries or queries that do not contain branching paths. Both queries depicted in Figure 3(b) and Figure 3(c) are linear path queries.
- **Twig queries:** These are queries that contain branching paths. The twig queries can be further broken down into two categories:
 - **Existential Twig Queries (ET):** Such queries are similar to the simple path queries in the sense that they contain a single target node but they contain some branching conditions.
 - **Regular Twig Queries (RT):** Unlike existential twig queries, regular twig queries involve finding and retrieving all combinations of multiple target nodes from the XML data tree.

Note that all of the above queries can involve parent-child (P-C) or ancestor-descendant (A-D) relationships.

2.2.4. Query Selectivity Count

Query selectivity count is the number of matches of a given query in the source XML data tree. Chen et al. [20] definition of a twig match can be slightly modified to formally define the selectivity count as follows:

Definition 1.1 (selectivity count):

Given a labeled XML data Tree $T_d = (V_d, E_d)$ and a query (twig or path) $T_q = (V_q, E_q, \{/, //\})$, the selectivity count of T_q is the approximate number of matches $c(T_q)$ that satisfy the following mapping : $f: V_q \rightarrow V_d$ such that if $f(u) = v$ for $u \in V_q$ and $v \in V_d$, then

- 1- $\text{Label}(u) = \text{Label}(v)$
- 2- If $(u, u', /) \in E_q$, then $(f(u), f(u')) \in E_d$.
- 3- If $(u, u', //) \in E_q$, then there is at least one linear path p in T_d rooted at $f(u)$ and reaches $f(u')$ where all the edges in $p \in E_d$

The selectivity count of a simple path query is simply the count of the nodes in the data tree matching the type of the query-target node (i.e. the node at the end of the expression) and satisfying the structural conditions represented by the edges in the query tree. Consider the XML queries in Figure 3(b) and Figure 3(c). Both queries have the selectivity count of 1. This is because the element A in the data tree has a single B child satisfying the query in Figure 3(b) and a single D descendant satisfying the query in Figure 3(c).

For existential twig queries, only the count of the target node contributes to the query selectivity count while the branching paths present structural conditions indicating that at least a single occurrence of each branch should exist in the source tree. Figure 6(b) shows an example of a twig query over the source tree in Figure 6(a). If this is an existential twig query and D is the single target node, then the selectivity count of the query in this case is 1 since there is only one D element whose parent has a sibling of type C .

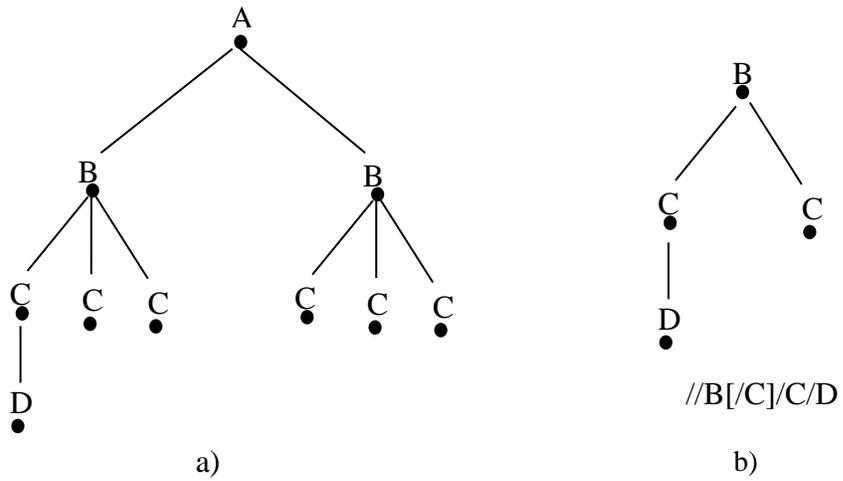


Figure 6: a) XML tree, and b) Twig query and its corresponding XPath expression

Unlike existential twig queries, the counts of all branching paths contribute to the overall query selectivity count in regular twig queries. Consider the XML query in Figure 6(b). If it is a regular twig query, then its selectivity count is 2 since there are two *C* elements in Figure 6(a) that can be matched with the single *D* element satisfying the structure described by the query.

CHAPTER 3

RELATED WORK

In the past decade, a number of selectivity count estimation techniques have been proposed in the literature by the XML database research community. The general idea of most of the techniques in the literature is to summarize the XML document in a way that preserves the structural relationships between the elements and can reflect the actual or approximate counts of those elements in the source document. Such summary structures or synopses can be in the form of graphs or trees. Examples of these techniques can be found in [21-26,57]. While graphs have the advantage of modeling more types of elements such as the IDREF/ID (i.e. attributes that refer to another element's ID value), trees can be traversed more efficiently than graphs. In contrast, histogram-based techniques depend on the use of statistical histograms to capture the structural and content distributions of the source XML documents [27 - 29, 31,58]. One important feature of histograms is that they can capture more accurate statistics when the distribution of the XML data is not uniform [31]. Moreover, they are usually simpler structures and easier to build than trees and graphs. Finally, statistics-based techniques build highly summarized statistical models to represent the structure of the source XML documents [21, 30]. Although statistical approaches are usually more storage efficient than synopses approaches, they tend to show higher error rates with non-uniform XML documents. In [31], the authors augmented a statistical model with histograms based on

an interval-based numbering scheme to reduce estimation errors when the underlying data is skewed. Table 1 summarizes the techniques in the literature.

One thing to note is that very few techniques address the selectivity count estimation of regular twig queries. This is also shown in the survey conducted by Sakr [10]. Also, several techniques are proposed for XML selectivity count estimation for queries with value predicates like [24, 32 - 36]. Since this work focuses on structural selectivity count estimation, we only discuss [24, 33] as examples for those techniques. In what follows we present some of the techniques in the literature.

3.1. Synopsis-Based Approaches

XSketch: Polyzotis et al. [26] proposed the XSketch technique which is based on a generic graph synopsis model. In this model, each node corresponds to a set of identically labeled node types in the source XML document. The synopsis graph is augmented with edge labels that represent the backward and forward stability properties. This way, the counts of paths that consist of backward stable edges will depend on the count of the last node in the path expression only. This reduces the computational complexity and also improves the estimation accuracy. In addition, a multidimensional histogram is associated with each node type to capture the distribution of the values associated with that node type. Consequently, the estimate is computed by combining the stability information of the structural part of a given twig pattern and the distribution of the values which are in the value predicates part of the same twig pattern. This technique works well with the uniformity assumption and simple path expressions but can produce a high error rate with twig queries [25]. A generalization of the XSketch called the fXSketch has been proposed

Table 1 Selectivity count estimation techniques

Reference	Approach	Category	Supported Queries	Year
Polyzotis et al. [26]	Xsketch	synopsis	Linear, existential	2002
Polyzotis et al. [25]	Twig-Xsketch	synopsis	Linear, existential, and regular	2004
Polyzotis et al.[38]	TreeSketch	synopsis	Linear, existential, and regular	2004
Abounnaga et al. [21]	Path-Tree 1	synopsis	Linear	2001
Alrammal et al. [39]	Path-Tree 2	synopsis	Linear, existential	2011
Li et al.[40]	Path-Encoding:	synopsis	linear, existential	2006
Lim et al.[59]	XPath-Learner	synopsis	linear, existential	2002
Zhang et al.[41]	Xseed	synopsis	linear, existential	2006
Luo et al. [9]	Sampling	synopsis	Linear, existential, and regular	2009
Chen et al. [56]	Correlated Subpath Tree	Synopsis	Linear, existential, and regular	2001
Wu et al. [28]	Position-Histogram	histograms	linear, existential	2002
Wang et al.[29]	Bloom-Histogram	histograms	linear	2004
Lim et al.[33]	CXHist	histograms	linear	2005
Li et al.[40]	Path-order	histograms	linear,existential	2006
Freire et al.[57]	StatiX	histograms	linear, existintial	2002
Abounnaga et al. [21]	markov Table	statistical	linear	2001
Lee et al.[30]	NodeRatio-Node-Factor	statistical	linear,existential	2004
Wang et al. [42]	Probabilistic-Decomposition	statistical	linear,existential	2004

to improve the accuracy of the estimates and handle other types of queries. fXSketch is proposed to cope with fractional stabilities by recording more details about path/branching distributions [37].

Twig-XSketch: In order to improve the accuracy of the estimate for twig queries, Polyzotis et al. in their later work [25] extended the XSketch synopsis to capture the path distribution information at a finer level of detail. The idea is to store a multidimensional histogram per synopsis node (a node in the XSketch graph) that represents the localized edge distribution. For example, if a node n in the synopsis has two outgoing edges (n, t) and (n, q) , then the histogram $H_n(c_1, c_2)$ would represent the fraction of n nodes in the data tree that have exactly c_1 children of type t and c_2 children of type q . This introduces the requirement for extra storage space to maintain the histograms along with the XSketch graph. Figure 7(b) shows a sample XSketch graph of the tree in Figure 7(a) and Figure 7(c) shows XSketch extension to include the edge distribution histograms. For example, the selectivity count of the query $C[/G]/H$ can be calculated using the graph in Figure 7(b) and the histogram in Figure 7(c) as follows:

$$\sum_{g,h} |C| H_C(g,h) g.h \quad (1)$$

One problem with the XSketch synopses is that they are complex to construct and also the edge histograms are kept for only a subset of the paths which leads to poor estimates for twig queries whose paths distribution information is not stored.

TreeSketch: Polyzotis et al. [38] proposed the TreeSketch synopsis to summarize the XML documents for the purpose of selectivity count estimation. The TreeSketch is a summary graph that clusters the elements in the XML tree based on the count stability

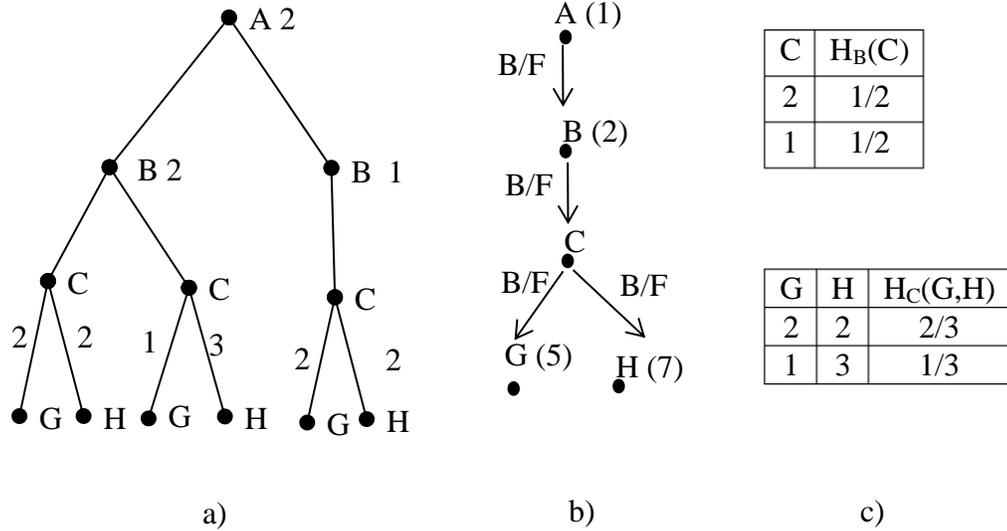


Figure 7: a) XML tree with the counts of nodes at the edges, b) XSketch graph for the tree in a, and c) edge distribution histograms

concept and the space budget. Nodes in the XML tree that have the same substructure and the same count for each child and descendant belong to the same graph node. Figure 8 shows the TreeSketch graph for the XML tree in Figure 7. The main advantage of the TreeSketch approach is that in addition to estimating the selectivities, it facilitates the generation of the query results. One issue with this approach is that it can be very time consuming to generate the Treesketch synopsis according to a certain space budget. Also, the synopsis size can become large if the XML source tree exhibits any type of irregularity in terms of elements' structures or counts.

Path-Tree-1: Aboulnaga et al. [21] proposed two techniques to estimate the selectivity counts of path expressions. Their first technique is based on capturing the structure of the XML data on a path-tree. The aim is to represent the structure of the source XML tree in a more succinct manner using the path tree. This tree contains the nodes frequencies

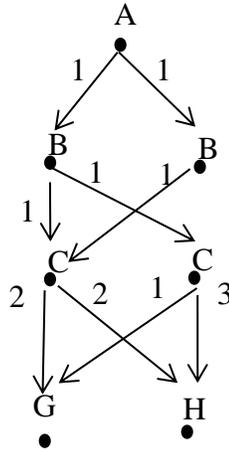


Figure 8: Count stable TreeSketch graph for the XML tree in Figure 7(a)

through all paths. A node in the path tree represents a path string from the root, and there is a child node for every distinct element reachable by that path. Each node is labeled with the type of the element reachable by that path along with the frequency of the node. Figure 9(a) is an example of a path tree representation of an XML document. The authors also suggested four techniques to summarize the path-tree itself in case it was larger than the available memory. For instance, one of the methods they used to summarize the path-tree was called the Sibling-*. The basic idea of this method is to repeatedly merge siblings with the lowest frequencies into one node called the * node until the tree can fit in the memory. The *-node then has the average frequencies of the merged nodes. Moreover, all children of the merged nodes become the children of the newly created *-node and the children with the same tag name are merged and their frequencies are added rather than averaged as shown in Figure 9(b). In this work the authors did not address the selectivity count estimation for twig queries.

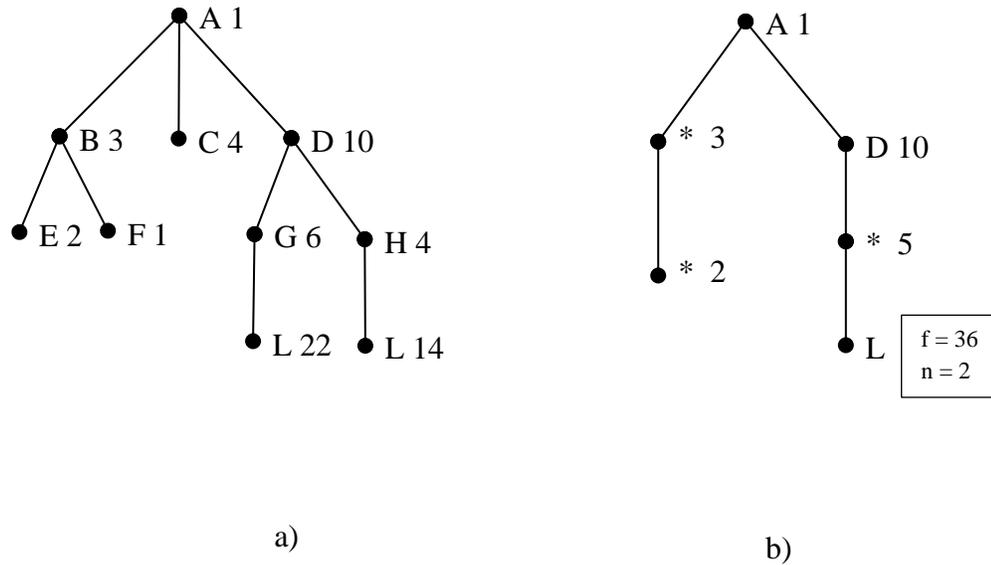


Figure 9: Path tree example: a) Sample path tree, and b) Sibling* summarization.

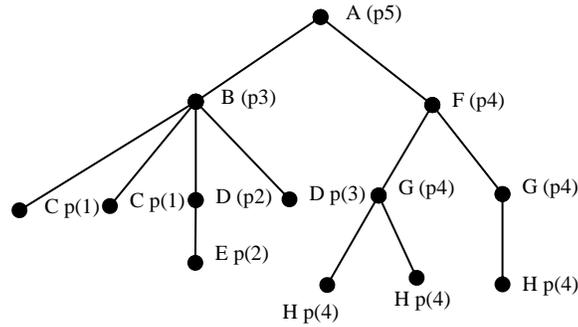
Path-Tree-2: Alrammal et al. [39] also used the path-tree model, Figure 9(a), as the basis for selectivity count estimation. Furthermore, in their estimation system the path-trees are generated incrementally from the source document therefore partial query selectivity count estimates can be retrieved using the partial path-trees. Like the path-tree technique proposed in [21], their work does not address the selectivity count estimation for regular twig queries.

XPath-Learner: Alrammal et al. [59] proposed the XPath-learner system to estimate the selectivity counts of XPath expressions. The XPath-Learner is similar to their earlier work in [39] with a few modifications. The XPath-Learner is an on-line system and it does not scan the XML document directly to collect the required statistical information but instead it uses query feedbacks and therefore the collected statistics about the source XML document depend on the query workload. This allows the XPath-Learner to efficiently allocate more storage for the statistical information about more frequent

queries in the workload to achieve a higher estimation accuracy. Although, XPath-Learner can support value predicates in path expressions, it does not support regular twig queries.

Path-Encoding: Li et al. [40] proposed a path-based encoding model for XML selectivity count estimation. Here every distinct root-leaf path is represented by a binary integer with length k bits where k is the total number of distinct root-leaf paths. The inner nodes are labeled by performing the binary “or” operator on their children and all IDs are saved in an encoding table. The frequency of each path is then saved in a PathID-frequency table which is used to estimate the selectivity count of a given query. Figure 10 shows an XML tree and its corresponding encoding table and PathID-frequency table. They also build another table for each element tag to capture the element order information. This table is used to estimate the selectivity counts of queries with order axes. Moreover, they generate p-histograms and o-histograms to summarize the PathID-frequency and element order tables respectively. To the best of our knowledge, this technique is the first to address the selectivity counts of order-sensitive queries but the selectivity count estimation problem for regular twig queries is not addressed in their work.

XSeed: Zhang et al. [41] proposed the XSeed selectivity count estimation system. The XSeed synopsis is a label-split graph called the kernel. The XSeed kernel edges are labeled with vectors of integer pairs $(p_0:c_0, p_1:c_1 \dots p_n:c_n)$ where the i -th pair $p_i:c_i$ shows that at recursion level i there are p_i elements in the XML tree that have c_i children. Figure 11 shows a sample XML tree and its corresponding XSeed kernel. One of the main contributions of their work is that they explicitly address the recursion (i.e. elements that



a)

Root-	Encod
A/B/C	1
A/B/D/	2
A/B/D	3
A/F/G/	4

b)

Bit-Seq	ID
0001	p1
0010	p2
0011	p3
0100	p4
0111	p5

c)

Ele	PathID-Frequency
B	(p3,1)
F	(p4,1)
C	(p1,2)
D	(p2,1),(p3,1)
G	(p4,2)
H	(p4,3)
A	(p5,1)

d)

Figure 10: a) XML tree, b) Encoding table, c) Bit-Seq table, and d) PathID-freq table

share the same type with one or more child or descendant nodes) in the XML document. They also complement the XSeed kernel with a hyper-edge table (HET) which contains the cardinalities of some queries that are known to produce a high error rate under the path independence assumption. Unfortunately, their work does not address the regular twig queries.

Sampling: Luo et al. [9] proposed a sub-tree sampling approach to estimate the selectivity counts of regular twig queries. The idea is to examine the number of nodes of each element type starting from the first level. If the number of nodes is sufficiently

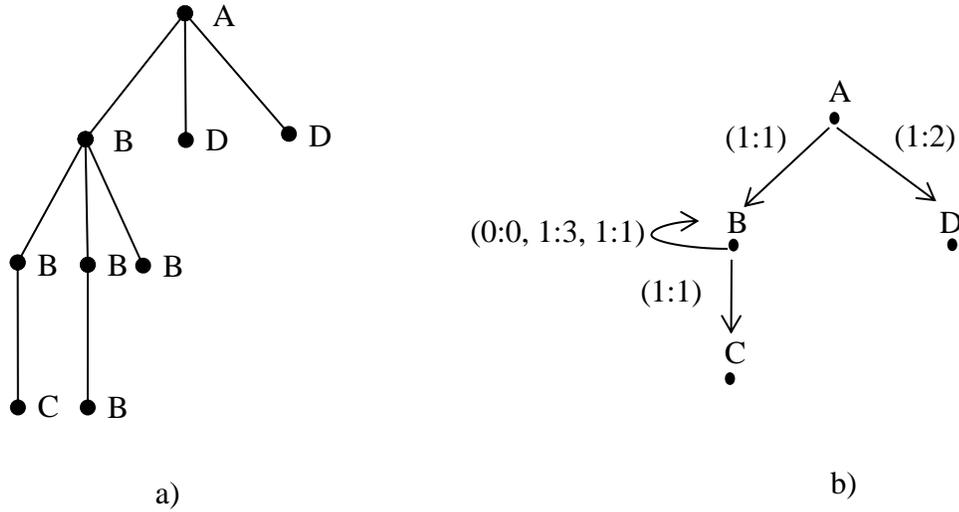


Figure 11: XSeed example a) XML tree, and b) XSeed kernel

large, they randomly sample a certain fraction of the nodes and their sub-trees. They also save the root-to-sub-tree paths in the sample. On the other hand, If the number of nodes is not large then the next level is examined for sampling. The generated sample XML tree is then used to retrieve the selectivity count of twig queries. The main advantage of this technique is its simplicity. In other words, samples can be generated quickly for a given storage budget. Unfortunately, the sample tree is not always a good representation of the source XML document especially for irregular XML data, where elements of the same type tend to have different structures (sub-trees), and when the memory budget is limited.

Correlated Subpath Tree: Chen et al. [56] proposed a selectivity count estimation technique for XML queries. In their work, they maintained the count statistics of the subpaths in the XML document up to a certain length in a tree structure. Then they captured the correlations between the subpaths with the same root using a set hashing signature. In order to estimate the selectivity counts of twig queries, they decomposed the

query into a set of paths stored in the CST and then combined the retrieved paths and used their stored statistical counts to estimate the overall query selectivity count. The CST was among the first techniques for XML query selectivity count estimation especially for twig queries. Also, the CST can handle substring queries at the leaves. One issue is that the CST normally consumes a large storage to achieve a reasonable estimation accuracy. However, it is shown in [38] that the TreeSketch outperformed this technique in terms of estimation accuracy.

3.2. Histogram-Based Approaches

Position-Histogram: Wu et al. [28] used two dimensional position-histograms to capture the structural information of XML documents. First, they merged all XML documents into a single root document and then they labeled the nodes using the range encoding labeling scheme [14]. After that, they generated a 2-dimensional histogram for the nodes that satisfy a certain predicate P of the form “element = $\langle element\ name \rangle$ ”. In other words, they generated a histogram for every distinct element. The x -axis of the histogram represents the *start* position of the node while the y -axis represents the *end* position according to the range encoding scheme. Consequently, every cell in this histogram represents a range of start and end positions and it contains the count of the nodes satisfying the predicate P and falling in the range $[start, end]$. This histogram is called the position histogram. The ancestor descendant relationship between any two nodes can be then identified by combining their histograms and using the fact that the *start* and *end* ranges of any two nodes can either have no overlaps or one range is totally contained in the other. The main issue with this approach is that it only addresses the ancestor-

descendant axis.

Bloom-Histogram: Wang et al. [29] proposed a selectivity count estimation technique based on bloom-histograms. The bloom histogram is generated using a path-table that contains all paths in the XML document and their frequencies. Unlike the path-table the bloom histogram contains a fixed number of buckets and each bucket contains paths with similar counts or frequencies. Thus, the bloom histogram has two columns, namely the bloom filter which represents the set of paths in each bucket, and the count which reflects the frequencies of the paths represented by the bloom filter. More specifically, the bloom filter is a bit array of a fixed length m with k hash functions $h_1 h_2 \dots h_k$. To add a new element x , all k hash functions are applied and every bit $h_i(x)$ is turned to 1 where $1 \leq i \leq k$. To query an element q , all $h_i(q)$ of a bloom filter must be 1 if q belongs to the set represented by the filter. In this work, the authors sort the path table based on the frequencies and then group paths with similar frequencies into buckets and the selectivity count of a given path can be retrieved by testing the membership of the path using the bloom filter at each bucket. Figure 12 shows an example of a path table and its corresponding bloom-histogram. One thing to note here is that the accuracy of the estimation is highly dependent on the selection of parameters, namely k , the number of hash functions, and also m which is the size of the bit array. Moreover, the number of buckets has to be chosen carefully because if the number of buckets is too small then each bucket will contain a wider range of frequencies and the error rate will increase. On the other hand, a large number of buckets will increase the storage requirement. Another thing that needs careful attention is the selection of the hash functions which might

Path	Count
/A	9
/A/B	11
/A/C	39
/A/D	41
/A/E	79
/A/F	81

Bloom Filter	Count
BF(/A,/A/B)	10
BF(/A/C,/A/D)	40
BF(/A/E,/A/F)	80

Figure 12: Example of a path count table and its corresponding bloom histogram

produce false positives (i.e. more than one bloom filter can test positive for the membership of a given path). Also, the authors address only the linear path expressions in their work.

CXHist: Lim et al. [33] proposed a machine learning technique for XML selectivity count estimation. Their technique is based on building a query model using the PathID and n-grams for string and substring predicates as features. Each $\langle path, string \rangle$ query is mapped to a bucket representing the query selectivity count using the Bayesian classifier. This is an online technique and the histogram is tuned depending on the workload and query feedbacks. One problem with online techniques is that the estimation accuracy can be very low for new queries. This is because they use the results (feedback) after executing the query to tune the histograms and therefore the accuracy is improved only for the queries that are seen (executed) earlier.

StatiX: Freire et al. [57] proposed the StatiX system for XML selectivity count estimation. The StatiX system uses the XML Schema to capture statistics about the source XML structure and values and then stores these statistics on histograms. This system is composed of two main components: the XML Schema Validator and the XML Schema Transformer. The XML Schema validator validates the XML documents against

the schema and simultaneously collects elements statistics for the given schema. The XML Schema Transformer ,the second component of StatiX, collects more detailed statistics about the elements distribution to capture skewedness in the data. StatiX system is used in the context of LegoDB system [59] and their experiments show highly accurate results but for limited types of queries. For instance, StatiX does not support query estimation for regular XML queries. Also, StatiX cannot be applied on Schema-less XML documents.

3.3. Statistical Approaches

Markov Table: Aboulnaga et al. [21] presented a technique that is based on storing all paths in the XML data tree on a Markov table. The purpose of this table is to summarize the structure of the XML tree. This is done by saving all distinct paths in the data tree up to a fixed number $m \geq 2$ and their frequencies in a table structure. The selectivity counts of paths of length ≤ 2 is directly retrieved from the table while the selectivity counts of paths of length > 2 is estimated using the following formula:

$$f(t_1/t_2 \dots/t_n) = f(t_1/t_2 \dots/t_m) \prod_{i=1}^{n-m} \frac{f(t_{1+i}/t_{2+i} \dots/t_{m+i})}{f(t_{1+i}/t_{2+i} \dots/t_{m+i-1})} \quad (2)$$

The fraction $\frac{f(t_{1+i}/t_{2+i} \dots/t_{m+i})}{f(t_{1+i}/t_{2+i} \dots/t_{m+i-1})}$ is interpreted as the average number of t_{m+i} elements contained in all $t_{1+i}/t_{2+i} \dots/t_{m+i-1}$ elements. They also present several techniques to summarize the Markov tables. This approach provides accurate estimates for linear path queries but it cannot estimate the selectivity counts of twig queries.

NR-NF: Lee et al. [30] proposed a statistical approach for selectivity count estimation. Their work is based on collecting all parent-child paths in the XML data tree along with

the node counts and then generating two types of statistics about the nodes, namely the node ratio NR and the node factor NF . For every parent/child path $PC = P/C$ the node ratio is defined as the ratio of the frequency of P in PC to the frequency of P in the data tree (i.e. P node count). Moreover, the node factor is defined as the ratio of the frequency of C in PC to the frequency of P in PC . The selectivity count of node C in PC becomes:

$$S(C) = frequency(P) * NR(P|C) * NF(C|P) \quad (3)$$

Figure 13 shows an XML tree with the corresponding node statistics. For existential twig and path queries with more than one parent-child path, they recursively decompose the given query into multiple simple parent-child queries and then aggregate the NR and NF statistics to estimate the size of the original query. Also, the ancestor-descendant queries are converted into twig queries with parent-child axes. Every branch in the generated twig query is a possible query-root to query-target node full path in the data tree. While their approach is very efficient in terms of storage utilization, it performs poorly on skewed (non-uniform) XML data trees. In their later work [31] they proposed the use of histograms based on the range encoding scheme on selected basic parent/child paths in order to improve the estimation accuracy. Both approaches do not address the regular twig queries.

Probabilistic-Decomposition: Wang et al. [42] proposed a statistical approach for XML selectivity count estimation. Their approach is based on saving the counts of the twigs in the XML tree of a selected size k . The selectivity count of a given query is then calculated by decomposing the query into smaller twigs of size k then estimating the

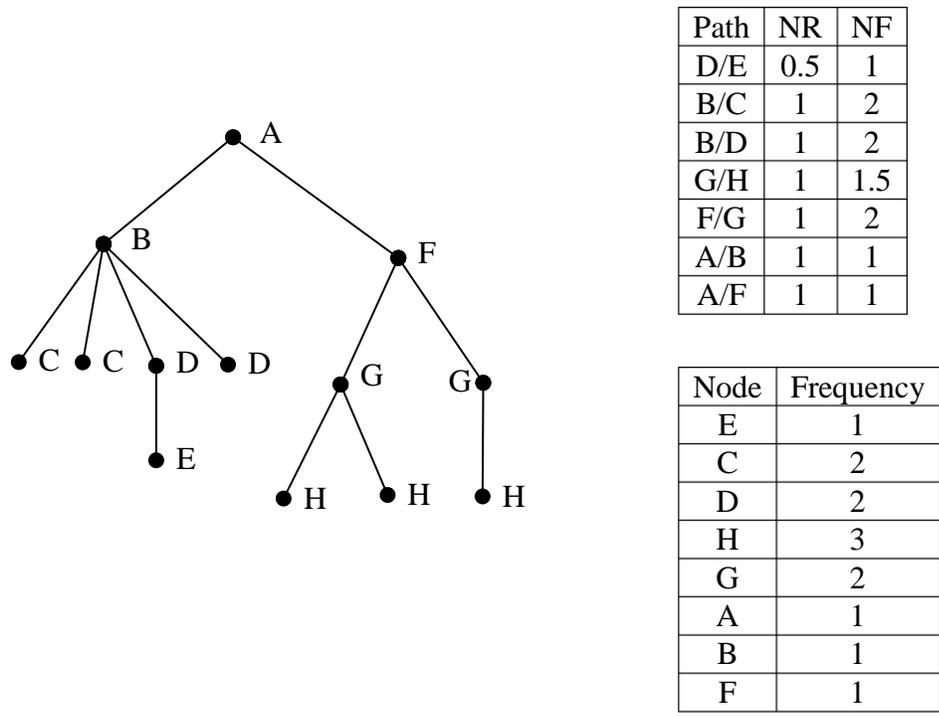


Figure 13: XML tree with the corresponding statistics

count of the query using the counts of the smaller twigs. For example, given two twigs T_1 with selectivity count $\sigma(T_1)$ and T_2 with selectivity count $\sigma(T_2)$ that differ only by one edge, the selectivity count of $T_1 \cup T_2$ is given by

$$\frac{\sigma(T_1) * \sigma(T_2)}{\sigma(T_1 \cap T_2)} \quad (4)$$

One issue with their approach is that it does not support ancestor-descendant queries. Also, generating all twigs of a certain size can be time consuming.

CHAPTER 4

XML STRUCTURE-BASED SUMMERIZATION

Our proposed approach, which we called *SynopTech*, consists of two main modules: 1) the tree summarizer, and 2) the selectivity count estimator. In this chapter, we discuss the first module (tree summarizer) and propose two different algorithms to implement this module, namely *SynopGenPrime* and *SynopGen*. *SynopGenPrime* is based on the prime-number labeling scheme [15] while *SynopGen* is based on a fingerprint hash function used in the string pattern matching domain. The second module of *SynopTech* is called *SynopCalc* and is discussed in the next chapter.

4.1. Summarization Based on Prime-Number Labeling

In what follows, we present some useful definitions before we discuss the details of *SynopGenPrime*.

4.1.1. SynopGenPrime Preliminaries

Definition 4.1: (*Leaf node label*)

Every distinct path from root to leaf is assigned a prime number which is used to label the leaf in that path. Identical paths share the same prime number.

Definition 4.2: (Inner node label)

Given a node n with a set of unique children $C\{c_1, c_2, \dots, c_m\}$; $Label(n)$ is defined as follows:

$$Label(n) = \begin{cases} \prod_{i=1}^m Label(c_i), & \text{if } |C| > 1 \\ Label(c_1^2), & \text{if } |C| = 1 \end{cases} \quad (5)$$

If n is the root node, $Label(n) = 0$.

Definition 4.3 (Summary node)

Given an XML tree $T(V,E)$ where V is the set of edges and E is the set of nodes (vertices), a summary node N represents a set of nodes $\{n_1, n_2, \dots, n_m\}$, which is a subset of V , such that $Label(n_i) = Label(n_{i+1})$ and sub-tree $S(n_i)$ is equivalent to sub-tree $S(n_{i+1})$ for all $1 \leq i < m$.

Observation 4.1:

Given an XML tree $T(V,E)$, for any two nodes n_1 and $n_2 \in V$, the following holds:

If the sub-trees rooted at n_1 and n_2 are identical, i.e. $S(n_1) = S(n_2)$, then $Label(n_1) = Label(n_2)$.

This is because if $S(n_1) = S(n_2)$ then $S(n_1)$ and $S(n_2)$ have the same set of root-leaf paths and since every distinct root-leaf path is labeled with a prime number, the product of all root-leaf path labels in $S(n_1)$ will produce the same composite number as the product of all root-leaf path labels in $S(n_2)$ and thus $Label(n_1) = Label(n_2)$.

Example: Figure 14 shows an XML data tree. Figure 15 shows the tree after summarization with a tuple $(ID, count)$ on each node. The table in Figure 15(b) shows the

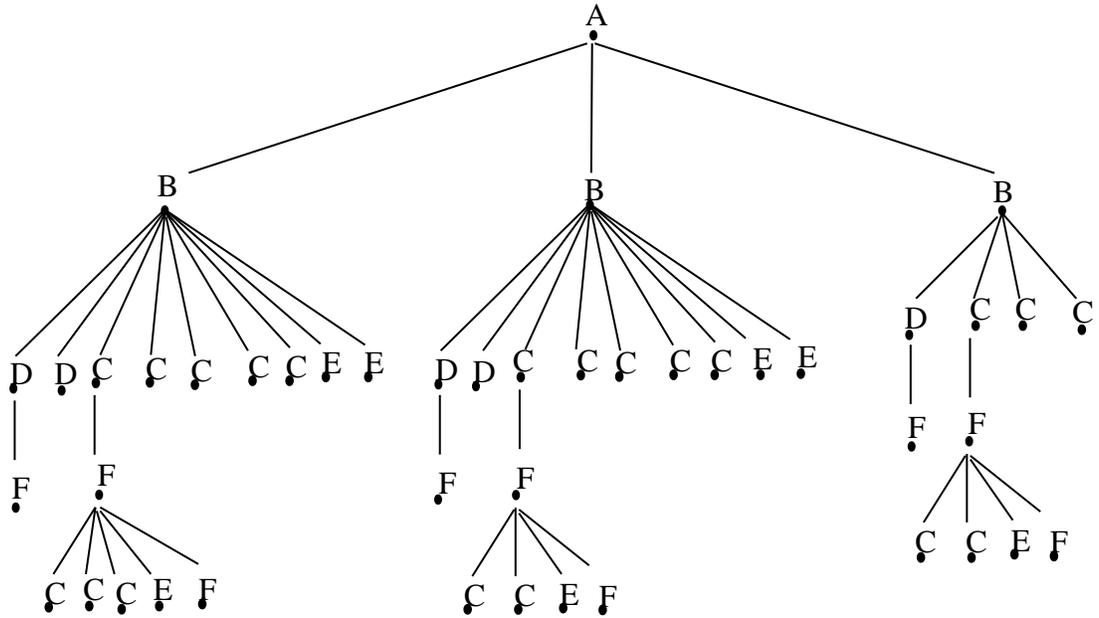


Figure 14: A sample data tree for an XML document

distinct root-leaf paths and their labels while the table in Figure 15(c) shows all the inner nodes and their labels. Note that the tables in Figure 15(b) and Figure 15(c) are only maintained during the construction of the summary tree and they are discarded after the summary is generated.

4.1.2. Construction

The idea of *SynopGenPrime* is to use a node labeling scheme and merge nodes with identical labels to generate a summary tree. The main steps of *SynopGenPrime* are outlined in Algorithm 1. It takes as input the XML data tree, T_d , and returns a summary tree, T_s , as output. *SynopGenPrime* traverses the data tree in post-order style starting at the root node (level 0), line 2, and tags each unique root-to-leaf path of the XML data tree with a unique prime number. Leaf nodes are labeled by their corresponding root-to-leaf

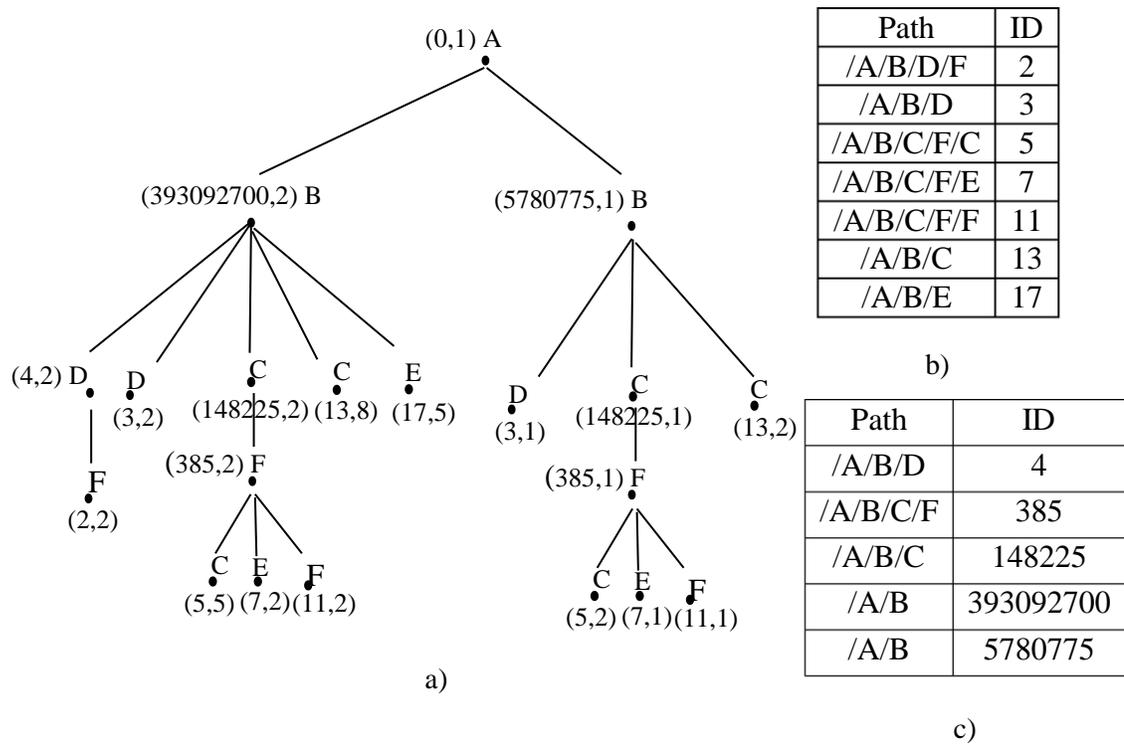


Figure 15: a) XML summary tree of the document, b) Leaf nodes and their prime-number labels, and c) Inner nodes and their labels

path tags, line 5, whereas the inner nodes are labeled using Definition 4.2 by calling the *computeLabel()* function, line 17.

SynopGenPrime merges sibling nodes with identical labels by the function *mergeSiblings()*, line 16 (defined in Algorithm 2). This function associates with each node a count, which represents the number of nodes merged. When two nodes are merged, their sub-trees will be also merged. This is because when two nodes, say x and y , are merged, they become one node, say w . Consequently, the child nodes of x and the child nodes of y become child nodes of w . In other words, they become siblings. Then,

Algorithm 1: SynopGenPrime

Input: T_d **Output:** T_s

```
1:  init(Stack S, Ts, dataGuide, level)
2:  while ( $v \leftarrow T_d.nextPreorderNode()$ )  $\neq$  NULL do
3:     $T_s.add(v)$ ;  $dataGuide.add(v)$ ;
4:    if  $isLeaf(v)$  then
5:       $v.label \leftarrow dataGuide.getLabel(v)$ ;  $S.push(v)$ ;
6:    else if  $v.level \geq level$  then
7:       $v.label \leftarrow -1$ ;  $S.push(v)$ ;
8:    else
9:       $i \leftarrow 0$ ;  $done \leftarrow FALSE$ 
10:   end if
11:   while  $!done \ \&\& \ !S.empty()$  do
12:      $u \leftarrow S.pop()$ 
13:     if  $u.label \neq -1$  then
14:        $sibling[i] \leftarrow u$ ;  $i++$ 
15:     else
16:        $i \leftarrow mergeSiblings(sibling, i)$ 
17:        $u.label \leftarrow computeLabel(sibling, i)$ 
18:       if  $u.level == v.level$  then
19:          $v.label \leftarrow -1$ ;  $S.push(u)$ ;  $S.push(v)$ ;  $done \leftarrow TRUE$ ;
20:       else
21:          $sibling[0] \leftarrow u$ ;  $i \leftarrow 1$ 
22:       end if
23:     end if
24:   end while
25:    $level \leftarrow v.level$ 
26: end while
27:  $i \leftarrow 0$ 
28: while  $!S.empty()$  do
29:    $u \leftarrow S.pop()$ 
30:   if  $u.label \neq -1$  then
31:      $sibling[i] \leftarrow u$ ;  $i++$ 
32:   else
33:      $i \leftarrow mergeSiblings(sibling, i)$ 
34:      $u.label \leftarrow computeLabel(sibling, i)$ 
35:      $sibling[0] \leftarrow u$ ;  $i \leftarrow 1$ 
36:   end if
37: end while
38: return  $T_s$ 
```

siblings with identical labels are merged. The process of creating new siblings and then merging is recursively repeated until all the siblings in the sub-trees are merged.

Note that the time complexity of Algorithm 1 mainly depends on the complexity of the merge function in line 16 whose complexity depends on the height and the maximum number of children per node in the data tree. For instance, if the input data tree is a fully populated K -ary tree, where K is the maximum number of children per parent, then at the first level there are K^1 nodes and therefore the number of merges cannot exceed $(K - 1)$. Each node in level one has exactly K children and at most $(K-1)$ possible child node merges therefore the number of node merges in level 2 is at most $(K - 1) \times K^{2-1}$. Thus the number of node merges at the leaf level cannot exceed $(K - 1) \times K^{h-1}$, where h is the height of the tree, and the summation of the number of node-merges in all levels will yield $K^h - 1$ which indicates that the number of node merges cannot exceed $O(K^h)$. Moreover, every node-merge operation involves merging the sub-trees of the nodes being merged. For example, every node in the first level is the root for $K^{2-1} + K^{3-1} \dots + K^{h-1} = \sum_{i=1}^{h-1} K^i$ sub-tree-nodes which represents the maximum number of sub-tree-node merges involved in every node-merge in level one. Similarly, in level 2 we have $K^{3-2} + K^{4-2} \dots + K^{h-2} = \sum_{i=1}^{h-2} K^i$ sub-tree-node merges per node-merge and so on. As a result summing up the number of sub-tree-node merges per node-merge in all levels yields $\sum_{i=1}^h i K^{i-1}$ which is in the order of $O(hK^{h-1})$. Consequently, the overall complexity of Algorithm 1 is in the order of $O(hK^h)$.

Algorithm 2: MergeSiblings

Input: *siblings, siblingCount*

Output: *distinctSiblingCount, updated sibling*

```
1:  $i \leftarrow 1$ ;
2: while  $i < siblingCount - 1 \ \&\& \ sibling[i].label > 0$  do
3:    $j \leftarrow i + 1$ ;
4:   while  $j \leq siblingCount \ \&\& \ sibling[i].label > 0$  do
5:     if  $sibling[i].label == sibling[j].label \ \&\& \ identicalSub-$   

     trees(sibling[i], sibling[j]) then
6:       mergeSub-trees(sibling[i], sibling[j]);
7:        $sibling[j].label \leftarrow -1$  ;
8:     end if
9:      $j++$ 
10:  end while
11:   $i++$ ;
12: end
13:  $distinctSiblingCount \leftarrow removeDuplicateSiblings(sibling, siblingCount)$ 
14: return  $distinctSiblingCount$ 
```

Getting Next Prime: in order to get a new prime label for a unique root-leaf path, line 5 of Algorithm 1, we use the deterministic variant of Miller-Rabin algorithm [43] to determine the next prime number which is denoted as *PrimeIncrement*. This algorithm is based on the following conditions:

Let a be a positive integer and n be a prime. If $n-1 = 2^q m$ ($q \geq 1$ and m is odd), then at least one of following statements is true:

$$a^m \equiv 1 \pmod{n}$$

or

$$a^{2^j m} \equiv 1 \pmod{n} \text{ for some } 0 \leq j \leq q - 1$$

Although there exists many composite numbers that satisfy the above conditions, fortunately Pomerance et al. [44] have verified that if $n \leq 1373653$ it is sufficient to test

Algorithm 3: primeIncrement

Input: positive integer n

Output: $prime$

```
1.   $a[] = \{2,3\}; prime = n; q = 0; m = n; isPrime = False;$ 
2.  while !isPrime do
3.       $prime++$ 
4.      foreach element in  $a[]$  do
5.           $isPrime = False$ 
6.          while( $m \bmod 2 == 0$ ) do
7.               $m = m / 2;$ 
8.               $q++;$ 
9.          end while
10.          $x = a^m \bmod prime$ 
11.         if  $x == 1$  then
12.              $isPrime = True$ 
13.         end if
14.         if  $isPrime == False$  then
15.             for  $j = 0; j \leq q-1$ 
16.                 if  $m == -1 \bmod prime$  then
17.                      $isPrime = True$ 
18.                 exit for
19.             end if
20.              $x = x^2 \bmod prime$ 
21.         end for
22.         end if
23.     end foreach
24.     return  $prime$ 
```

with $a = 2$ and 3 to ensure that the number being tested is definitely prime. Also, Jaeschke [45] provided similar verifications for greater n . For instance, if $n \leq 341550071728321$ it is sufficient to test with $a = 2, 3, 5, 7, 11, 13,$ and 17 . This leads to the *PrimeIncrement* algorithm presented in Algorithm 3.

Note that for efficiency purposes the array $a[]$, line 1, only contains the basis 2 and 3. This can be easily modified to cater for n greater than 1373653. Also, if the data tree has N distinct root-to-leaf paths, then N prime numbers, $P_1, P_2, \dots, P_{N-1}, P_N$, are required to label these paths. In the first call to Algorithm 3 the input integer n is 0 and P_1 is given the

label 1 which is the first prime after 0. The next prime after 1 is 3 and so in the second call Algorithm 3 will require $P_2 - P_1$, 3-1, comparisons to find the next prime. Similarly, to label the last root-to-leaf path Algorithm 3 needs to perform at least $P_N - P_{N-1}$ comparisons and consequently the total number of comparisons required to label all root-to-leaf paths is linear and cannot exceed $P_N - 1$ which is of $O(P_N)$, where P_N is the highest prime label.

4.2. Summarization Using Fingerprinting (*SynopGen*)

The problem with the prime-based summarization is that the node IDs can become extremely large for big non-uniform data trees. For example, the XMark data tree requires *prime*IDs that are larger than 64 bits and sometimes larger than 128 bits. Moreover, the reverse of Observation 4.1, i.e., if two nodes share the same labels they share the same structures, is not necessarily true which indicates that two summary nodes can share the same ID while having different sub-tree structures. This can be proven using the simple example depicted in Figure 16. The Figure shows a part of an XML data tree labeled using *SynopGenPrime* algorithm right before the merge operation. Note that both “A” nodes have the same ID while having different structures. Therefore, we propose an alternative tree summarization technique that is based on the fingerprint function used in Karp-Rabin string search function [46]. This will drastically reduce the size of node IDs during the construction of the summary tree.

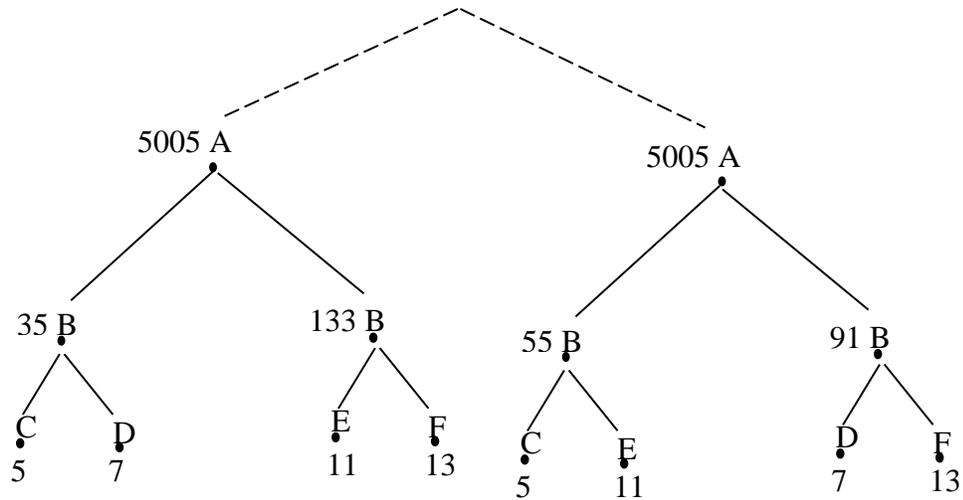


Figure 16: Two siblings with the same IDs and different structures

4.2.1. Construction

Like *SynopGenPrime*, *SynopGen* uses a node labeling scheme and merges nodes with identical labels to generate a summary tree. The main difference between the two algorithms is that the inner nodes are labeled using a fingerprint hash function and the root-leaf paths are labeled with a unique integer that is not necessarily prime. The main steps of *SynopGen* are outlined in Algorithm 4. It takes as input the XML data tree, T_d and returns a summary tree T_s as output. *SynopGen* traverses the data tree in post-order style starting at the root node (level 0), line 2, and tags each unique root-to-leaf path of the XML data tree with a unique number using the *dataGuide* object initialized by the *init()* function at line 1. Leaf nodes are labeled by their corresponding root-to-leaf path tags, line 5, whereas the inner nodes are labeled using a fingerprint hash function by calling the *ComputeLabel()* function at line 17. The fingerprint function needs the parameters K , B , and M to compute the inner node labels. These three parameters are generated by the function *getKBM()* at Line 1. The fingerprint hash function and the generation of K , B , and M will be explained in subsequent subsections.

SynopGen merges sibling nodes with identical labels by the function *mergeSiblings()*, line 16, presented in Algorithm 2 as explained in the previous subsection. The final tree generated after considering all nodes in the data tree is the summary tree, T_s . Since the fingerprint node labeling scheme cannot handle queries with ancestor-descendent (A-D) axis, a range node labeling scheme is executed by the function *rangeLabel()*, line 38. In this node labeling scheme, each node n is labeled by a tuple consisting of two numbers, namely, $n.start$ and $n.end$. These numbers are constrained such that if a node x is an ancestor of a node y , then $x.start < y.start$ and $x.end > y.end$. This can be performed by traversing T_s in a preorder and labeling all its nodes in an increasing order starting from any number. The first time a node n is visited, it is assigned a unique $n.start$ label, then when all its children are labeled, it is assigned a unique $n.end$ label in a post-order fashion.

Note that Algorithm 4 is similar to Algorithm 1 in terms of complexity which is $O(hK^h)$ as explained in section 4.1.2. The difference between the two algorithms is the inner node labeling mechanism. Also Algorithm 4 performs an additional scan of the tree in line 1 by invoking *getKBM()* to compute the labeling parameters. In the next chapter we will show how we can eliminate this scan by randomizing the selection of K .

4.2.2. Inner Node Labeling Scheme

Let $F(C)$ denote the label of a node C . If C is a leaf node, then $F(C)$ will be the label of the corresponding root-to-leaf path. Otherwise, C is an inner node and it will be assigned a label using a fingerprint function. Assume C is an inner node which has $|C|$ distinct children denoted as $c_0, c_1, \dots, c_{|C|-1}$. Then, $F(C)$ is calculated by the following equation:

$$F(C) = \sum_{i=0}^{K-1} F(c_{(i \bmod |C|)}) * B^{K-i-1} \quad (6)$$

where K represents the maximum number of distinct siblings in the data tree, and B is the size of the adopted alphabet. The computation of the values of K and B will be discussed

Algorithm 4: SynopGen

Input: T_d **Output:** T_s

```
1:  getKBM(), init(Stack S, Ts, dataGuide, level)
2:  while ( $v \leftarrow T_d.nextPreorderNode()$ )  $\neq$  NULL do
3:     $T_s.add(v)$ ;  $dataGuide.add(v)$ ;
4:    if  $isLeaf(v)$  then
5:       $v.label \leftarrow dataGuide.getLabel(v)$ ;  $S.push(v)$ ;
6:    else if  $v.level \geq level$  then
7:       $v.label \leftarrow -1$ ;  $S.push(v)$ ;
8:    else
9:       $i \leftarrow 0$ ;  $done \leftarrow FALSE$ 
10:   end if
11:   while  $!done \ \&\& \ !S.empty()$  do
12:      $u \leftarrow S.pop()$ 
13:     if  $u.label \neq -1$  then
14:        $sibling[i] \leftarrow u$ ;  $i++$ 
15:     else
16:        $i \leftarrow mergeSiblings(sibling, i)$ 
17:        $u.label \leftarrow computeLabel(sibling, i, K, B, M)$ 
18:       if  $u.level == v.level$  then
19:          $v.label \leftarrow -1$ ;  $S.push(u)$ ;  $S.push(v)$ ;  $done \leftarrow TRUE$ ;
20:       else
21:          $sibling[0] \leftarrow u$ ;  $i \leftarrow 1$ 
22:       end if
23:     end if
24:   end while
25:    $level \leftarrow v.level$ 
26: end while
27:  $i \leftarrow 0$ 
28: while  $!S.empty()$  do
29:    $u \leftarrow S.pop()$ 
30:   if  $u.label \neq -1$  then
31:      $sibling[i] \leftarrow u$ ;  $i++$ 
32:   else
33:      $i \leftarrow mergeSiblings(sibling, i)$ 
34:      $u.label \leftarrow computeLabel(sibling, i)$ 
35:      $sibling[0] \leftarrow u$ ;  $i \leftarrow 1$ 
36:   end if
37: end while
38:  $rangeLabel(T_s)$ 
39: return  $T_s$ 
```

shortly. If $F(C)$ is very large, Eq. 6 can be modified to limit $F(C)$ and B^{K-i-1} by using the modulo operator. Thus, the new equation will be as follows:

$$F_M(C) = \left(\sum_{i=0}^{K-1} F_M(c_{(i \bmod |C|)}) * (B^{K-i-1} \bmod M) \right) \bmod M \quad (7)$$

where M is a prime number. Because of the modulo operator, Eq. 7 can result in collisions (i.e. dissimilar nodes can have identical labels). Colliding nodes must not be merged if their sub-trees are different which requires additional processing time. Hence, it is required to minimize collision by choosing appropriate values for K , B , and M .

4.2.3. Selection of the Parameters K , B and M

Algorithm 5 shows how these parameters are determined. But since the appropriate value of M depends on the values of B and K , and the appropriate value of B depends on the value of K ; let us first discuss how to find K followed by the determination of B and M .

A) Finding K : Earlier we defined K to be the maximum number of distinct siblings in the data-tree. Having K less than the maximum number of siblings results in considering two inner nodes with different sub-tree structures to be identical. Hence, the probability of collision will increase and so will the CPU cost of *SynopGen*. As shown in Algorithm 5, K is generated by first initializing K , line 1. It then traverses the data tree in post-order, line 2. Each time it encounters a unique root-to-leaf labeled path it assigns it a unique prime number as its label, line 8. It also labels each leaf node with the label of its corresponding path, line 5. After labeling all the siblings of a node it finds the number of distinct siblings, line 16. It labels each inner node using definition 4.1, line 17. If the number of distinct children is higher than the current value of K , it changes K to the number of the current siblings, line 16. At the

end of the algorithm K will be equal to the highest number of distinct siblings in the data-tree.

B) Computing B : B in the fingerprint hash function represents the size of the alphabet. For example, if the function were matching bit sequences then B would be 2 and if it were matching English characters then B would be 26. In case of *SynopGen*, B represents the approximate number of distinct node labels in the data tree. When computing B , we assume that each node in the data-tree has K distinct children. In a full K -ary tree of height h , there are at most K^{h-1} distinct labels at level $h - 1$ and K^{h-2} at level $h - 2$ and so on. Thus B is computed from:

$$B = \sum_{i=0}^{h-1} K^i = \frac{K^h - 1}{K - 1} \quad (8)$$

C) Computing M : Consider the problem of matching bit sequences where it is required to find a match for a given pattern V of length $|V|$ bits in a sequence $S = s_0, s_2, \dots, s_{|S|-1}$ of length $|S|$ bits with $|S| \geq |V|$. This can be achieved by comparing the fingerprint $F_M(V)$ with every $F_M(S(j))$, where $S(j) = s_j, s_{j+1}, \dots, s_{j+|V|-1}$ and $j = 0, 1, \dots, |S|-|V|$. If $F_M(V) \neq F_M(S(j))$ then $V \neq S(j)$. However, if $F_M(V) = F_M(S(j))$, there is no guarantee that $V = S(j)$. A collision occurs if there exists j such that $F_M(V) = F_M(S(j))$ and $V \neq S(j)$, which means, $|F(V) - F(S(j))|$ is a multiple of M . In other words, to have a collision, $\prod_{\{j|V \neq S(j)\}} |F(V) - F(S(j))|$ must be a multiple of M . It has been proven by Rabin and Karp in [46] that the probability of collision, $Prob(collision)$, is $\frac{1}{|S|}$ if M is the highest prime less than $2|V||S|^2$. To apply this to our problem, let us assume a full K -ary tree. Since we start labeling from the leaf level upwards towards the root.

Algorithm 5: getKBM

Input: T_d **Output:** $K, B,$ and M

```
1:  init(Stack S, dataGuide, level, K);
2:  while ( $v \leftarrow T_d.nextPreorderNode()$ )  $\neq NULL$  do
3:    dataGuide.add(v)
4:    if isLeaf(v) then
5:       $v.label \leftarrow dataGuide.getLabel(v); S.push(v);$ 
6:    else if  $v.level \geq level$  then
7:       $v.label \leftarrow -1; S.push(v);$ 
8:    else
9:       $i \leftarrow 0; done \leftarrow FALSE;$ 
10:   end if
11:   while  $!done \ \&\& \ !S.empty()$  do
12:      $u \leftarrow S.pop();$ 
13:     if  $u.label \neq -1$  then
14:        $sibling[i] \leftarrow u; i++;$ 
15:     else
16:        $i \leftarrow distinctSiblings(sibling, i); K \leftarrow \max(K, i);$ 
17:        $u.label = computeLabel(sibling, i);$ 
18:       if  $u.level == v.level$  then
19:          $v.label \leftarrow -1; S.push(u); S.push(v); done \leftarrow TRUE;$ 
20:       else
21:          $sibling[0] \leftarrow u; i \leftarrow 1;$ 
22:       end if
23:     end if
24:   end while
25:    $level \leftarrow n.level;$ 
26: end while
27:  $i \leftarrow 0$ 
28: while  $!S.empty()$  do
29:    $u \leftarrow S.pop()$ 
30:   if  $u.label \neq -1$  then
31:      $sibling[i] \leftarrow u; i++$ 
32:   else
33:      $i \leftarrow distinctSiblings(sibling, i); K \leftarrow \max(K, i);$ 
34:      $u.label \leftarrow computeLabel(sibling, i);$ 
35:      $sibling[0] \leftarrow u; i \leftarrow 1$ 
36:   end if
37: end while
38:  $B \leftarrow \frac{K^{T_d.height-1}}{K-1};$ 
39:  $m \leftarrow \log_2 \left( K^{T_d.height-1} \left( \frac{B^{K-1}}{B-1} \right)^{T_d.height-1} \right)$ 
40:  $n \leftarrow m * K$ 
41:  $M \leftarrow HighestPrime(2 * m * n^2);$ 
42: return  $[K, B, M];$ 
```

the highest value for the fingerprint will be at the root. The fingerprint label of a node corresponds to the V pattern and the space of search, S , corresponds to all K fingerprints of the node and its siblings. To find these values roughly, we should find the highest value at the leaf level (level $h-1$) and from which we find the highest value at level $h-2$ and so on until we reach the root. The number of nodes at the leaf level, level $h-1$, is K^{h-1}

$$\begin{aligned}
F_{max}^{h-1} &= K^{h-1} \\
F_{max}^{h-2} &< \sum_{i=1}^K K^{h-1} * B^{K-i} \\
&= K^{h-1} \sum_{i=1}^K B^{K-i} \\
&= K^{h-1} \sum_{i=0}^{K-1} B^i \\
&= K^{h-1} \left(\frac{B^K - 1}{B - 1} \right) \\
F_{max}^{h-3} &= K^{h-1} \left(\frac{B^K - 1}{B - 1} \right)^2 \\
&\cdot \\
&\cdot \\
&\cdot \\
F_{max}^0 &= K^{h-1} \left(\frac{B^K - 1}{B - 1} \right)^{h-1}
\end{aligned}$$

Let m be the minimum number of bits in the highest node label and n be the number of bits in K labels; thus, $m = \log_2 F_{max}^0$ and $n \leq m * K$. After finding n and m , we choose M to be the highest prime number less than $2mn^2$ to get a $Prob(collison) \leq \frac{1}{n}$.

Note that Algorithm 5 does not actually perform any merge operation and every node is visited only once, therefore the post-order scan is done in $O(n)$ time where n is the number of nodes in the data tree.

4.2.4. Implementation of the Summary-Tree

SynopGen stores the summary-tree in a set of summary-tables $T = \{T_{t_1}, T_{t_2}, \dots, T_{t_{|T|}}\}$, where $|T|$ is the number of distinct types in the XML data tree. A summary-table named T_{t_i} corresponds to the type t_i . Each summary-table T_{t_i} has 4 columns, namely, *Start*, *End*, *Level*, and *count*; and is populated with the *Start*, *End*, *Level*, and *count* values of type t_i nodes in the XML data tree. The records in each summary-table are sorted in ascending order of the *Start* column. The maximum size of the *count* field is less than or equal to $\log N$ bits where N is the number of elements in the original data tree. The *Level* field is fixed at 4-bits as the depth of XML documents rarely exceeds 64. Also we allocate 8-bytes for each distinct element name. The total storage requirement cannot exceed the following:

$$storage = (\# \text{ distinct elements in data tree} * 64) + (2n * \log_2 n) + 4n + n \log_2 N \quad (9)$$

Example: Figure 17(b) shows the pathIDs and Figure 17(a) shows the summary tree of the sample XML tree shown in Figure 14. Table 2 shows the summary tables of the summary-tree shown in Figure 17. As can be seen in the table, each summary table corresponds to a type in the summary-tree. The records in each summary table are sorted in ascending order of *Start*.

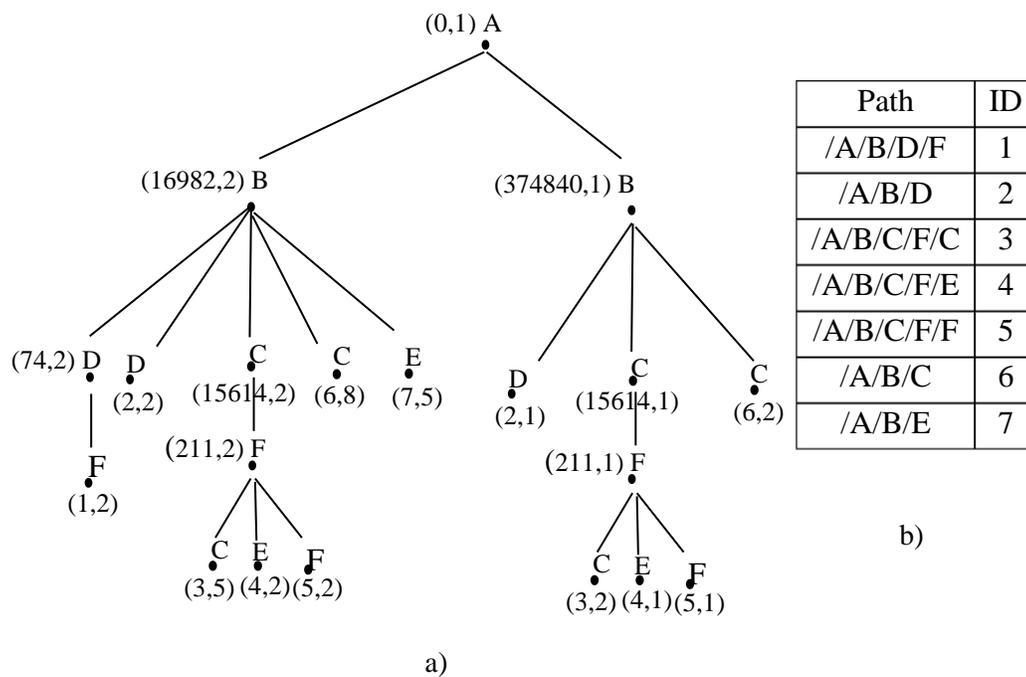


Figure 17: The summary tree of the XML data tree in Figure 14

Table 2: Summary-tables implementation of the summary-tree in Figure 17

Table	Start	End	Level	Count
T_A	0	27	0	1
T_C	6	12	2	2
	8	8	4	5
	13	13	2	8
	18	24	2	1
	20	20	4	2
	25	25	2	2
T_E	9	9	4	2
	14	14	2	1
	21	21	4	1

Table	Start	End	Level	Count
T_B	1	15	1	2
	16	26	1	1
T_D	2	4	2	2
	5	5	2	2
	17	17	2	1
T_F	3	3	3	2
	7	11	3	2
	10	10	4	2
	19	23	3	1
	22	22	4	1

CHAPTER 5

SELECTIVITY COUNT ESTIMATION

In this chapter we present the proposed selectivity count estimation algorithm, *SynopCalc* which takes as input a query and a summary-tree and returns as output the selectivity count estimate of the query. *SynopCalc* can find the selectivity count estimation for all types of queries, namely, linear, existential, and regular. But before we explain how *SynopCalc* computes the selectivity count estimation of the three types of queries, let us define some notations used in the rest of the chapter.

Let:

- $T_s(V_s, E_s)$ be a summary-tree where V_s and E_s are the set of nodes and the set of edges in the summary-tree respectively.
- $T_q(V_q, E_q)$ denotes a query where V_q and E_q are the set of nodes and the set of edges in the query respectively.
- $S = \{S_i(V_i, E_i) : 1 \leq i \leq |S|\}$ represents the set of sub-trees in T_s that match T_q where $|S|$ is the number of all sub-trees in S .
- $Target(S_i) \in V_i$ refers to the target node in S_i which contains the count of matches in S_i .

The pseudo code of *SynopCalc* is shown in Algorithm 6. It takes as input T_s and T_q and computes the selectivity count of T_q from T_s . It first checks what type of query T_q is and calls either *FindPathMatches* function or *FindTwigMatches* function accordingly to find matches in T_s and computes the selectivity count estimates, as explained in the following

subsections. Note that the complexity of Algorithm 6 depends on the complexity of the selected query processing algorithm that is used to retrieve the matches from the summary tree in lines 2 and 5. After that every match is scanned once to calculate its selectivity count before summing up the selectivity counts of all matches to estimate the overall query selectivity count as we will show in the following subsections. Therefore, if the number of matches is c and the number of nodes in the query tree is M , then there are at most cM scans which indicates that the complexity of the selectivity count estimation part, lines 7 to 15, of Algorithm 6 is $O(cM)$.

5.1. Selectivity Count of Linear Queries

To find the paths that match a linear query T_q in T_s , *SynopCalc* uses the *FindPathMatches* function, line 3 in Algorithm 6. The *FindPathMatches* function can use any of the existing tag-based query evaluation algorithms, such as *PathStack* [47,49]. For each query match, S_i , *SynopCalc* adds the Count of the $Target(S_i)$, to the selectivity count of T_q , which is denoted as $\|\hat{T}_q\|$. Eq. 10 shows the equation that is used by *SynopCalc*, line 9 in Algorithm 6, to compute the selectivity counts of linear queries.

$$\|\hat{T}_q\| = \sum_{i=1}^{|S|} Count(Target(S_i)) \quad (10)$$

where $Count(Target(S_i))$ is the *Count* of the target node in S_i .

Example: Consider the summary tables shown in Table 2 and a linear query $/A/B//C$ shown in Figure 18(a) where the target (output) node is C . To find the paths that match the query the *FindPathMatches()* function in *SynopCalc* searches the summary tables A , B , and C and returns the two paths that match the query. Figure 18(b) shows the matching paths. *SynopCalc* then computes the selectivity count of the query from these two paths

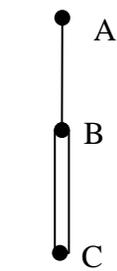
Algorithm 6: SynopCalc

Input: T_s, T_q
Output: *SelectivityCount*

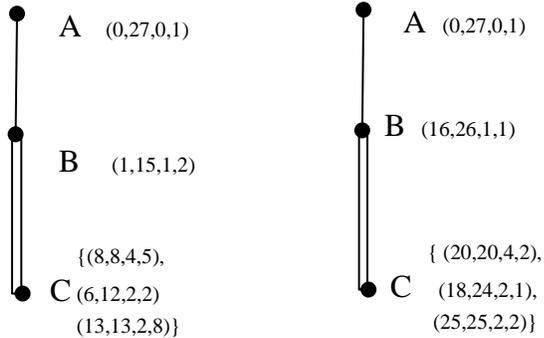
```

1: SelectivityCount  $\leftarrow$  0;
2: if  $T_q == \text{"Linear"}$  then
3:    $S \leftarrow \text{FindPathMatches}(T_q, T_s)$ 
4: else
5:    $S \leftarrow \text{FindTwigMatches}(T_q, T_s)$ 
6: end if
7: if  $T_q == \text{"Linear"}$   $\parallel T_q == \text{"Existential"}$  then
8:   for all  $S_i \in S$  do
9:     SelectivityCount  $\leftarrow$  SelectivityCount + Count(Target( $S_i$ ));
10:  end for
11: else
12:  for all  $S_i \in S$  do
13:    SelectivityCount  $\leftarrow$  SelectivityCount + twigCount(Target( $S_i$ ));
14:  end for
15: end
16: return SelectivityCount;

```


 $//A/B//C$

a) A linear query



b) Selectivity count

Figure 18: Selectivity count of a linear query

using Eq. 10 as follows: $ount(Target(S_1)) = 5 + 2 + 8 = 15$; $Count(Target(S_2)) = 2 + 1 + 2 = 5$, $||\hat{T}_q|| = 15 + 5 = 20$

5.2. Selectivity Count of Existential Twig Queries

To estimate the number of matches of an existential twig query, T_q , in T_s , *SynopCalc* uses the *FindTwigMatches* function, line 5 in Algorithm 6, which can use any of the existing tag-based twig pattern matching algorithms such as the TwigStack algorithm [48]. For each query match in T_s , *SynopCalc* adds the *Count* of the target node, $Target(S_i)$, to $||\hat{T}_q||$ similar to Eq. 10.

Example: Consider the summary tables shown in Table 2 and an existential twig query $/A/B[/C//E]/E$ shown in Figure 19(a). Records from the summary tables A , B , C , and E that satisfy the query are returned by the selected twig matching algorithm, line 5, and are depicted in Figure 19(b). The selectivity count is computed using Eq. 10 and is the sum of the *Count* of the target node, E , which is 5.

5.3. Selectivity Count of Regular Twig Queries

To estimate the number of matches of a regular twig query in T_s , *SynopCalc* uses the *FindTwigMatches* function, line 5 in Algorithm 6, which can use any of the existing tag-based twig pattern matching algorithms such as the TwigStack algorithm [48]. For each query match in T_s , denoted as S_i , *SynopCalc* estimates the number of matches in T_d , from:

$$twigCount(S_i) = Count(S_i.Root) \prod_{n \in \{V_i - S_i.Root\}} \frac{Count(n)}{Count(\Phi(n))} \quad (11)$$

where $\Phi(n)$ is the node directly connected to n as its parent or ancestor.

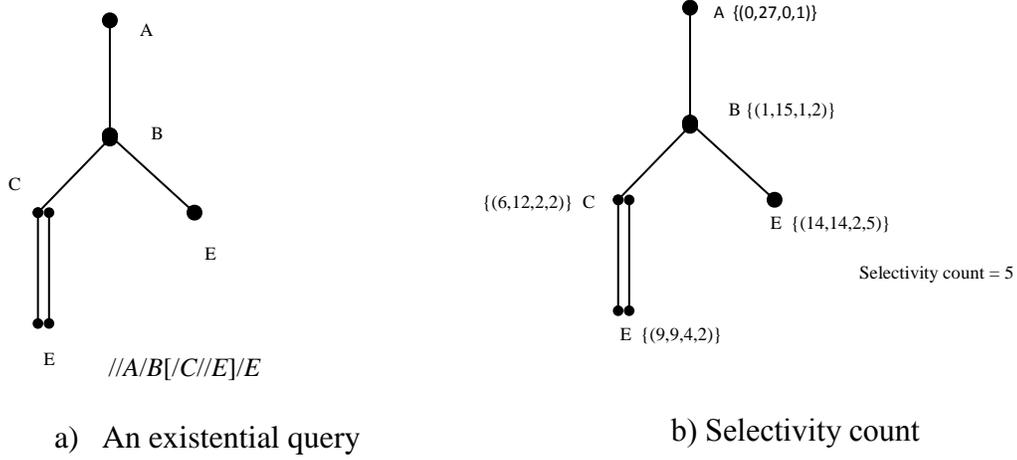


Figure 19: Selectivity count of an existential twig query

Subsequently, *SynopCalc* estimates the overall selectivity count of T_q as:

$$||\hat{T}_q|| = \sum_{i=1}^{|S|} \text{twigCount}(S_i) \quad (12)$$

Example: Consider the summary tables shown in Table 2 and a regular twig query $/A/B[/D]/C//E$ shown in Figure 20(a). To compute the selectivity count of the query, the *FindTwigMatches* function in *SynopCalc* will use a twig matching algorithm to search the summary tables T_A , T_B , T_D , and T_E to return the sub-trees that match the query. These sub-trees are depicted in Figure 20(b) and are denoted as S_1 , S_2 , and S_3 . Then it will compute the selectivity count of the query from these sub-trees using Eq. 12 as follows:

$$\text{twigCount}(S_1) = \binom{2}{2} * \binom{2}{1} * (1) * \binom{2}{2} * \binom{2}{2} = 2 ; \text{twigCount}(S_2) = \binom{2}{2} * \binom{2}{1} * (1) * \binom{2}{2} * \binom{2}{2} = 2; \text{twigCount}(S_3) \binom{1}{1} * \binom{1}{1} * (1) * \binom{1}{1} * \binom{1}{1} = 1; \text{ and } ||\hat{T}_q|| = \text{twigCount}(S_1) + \text{twigCount}(S_2) + \text{twigCount}(S_3) = 2 + 2 + 1 = 5.$$

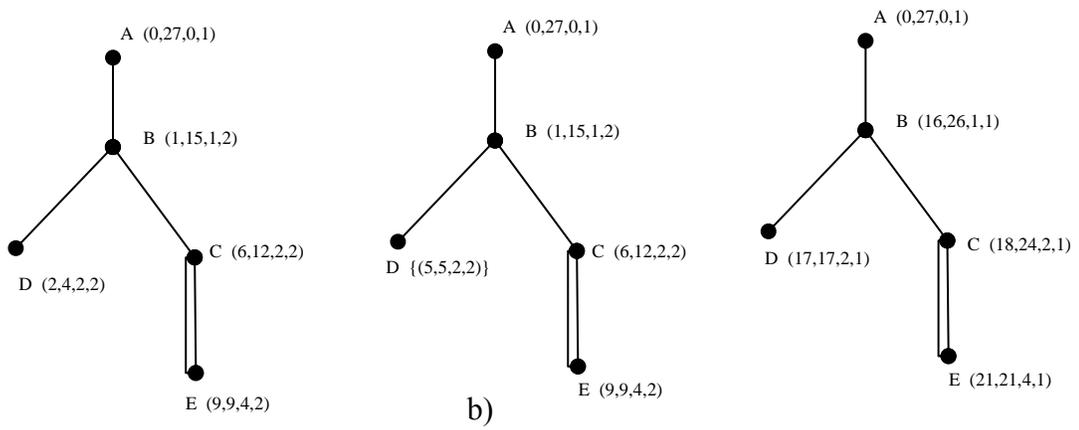
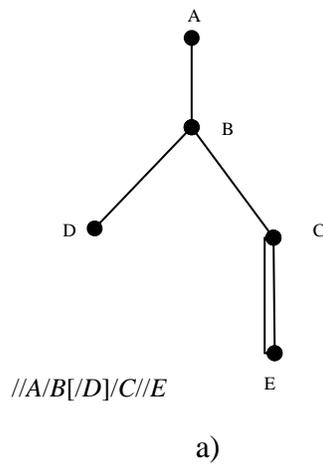


Figure 20: a) A twig query, and b) Sub-trees that match the twig query

CHAPTER 6

PERFORMANCE STUDY

In this chapter, we describe the conducted experiments to evaluate the performance of the proposed technique, *SynopTech*. We used the three common evaluation measures, namely, the summary-tree generation time, the estimation error rate, and the size of the summary tree. We also compared its performance with that of state-of-the-art techniques from the literature, namely, the Sampling algorithm; using the code provided to us by its proposers [9], and the TreeSketch using our implementation of that technique as presented in [38].

6.1. Experimental Settings

All the experiments were conducted on an Intel 2.8 GHz machine with 2GB RAM running Windows 7 operating system and all our programs were written in C#. In our experiments, we adopted four datasets, namely, DBLP [50], XMark [51], Ssplays [52], and Uniprot [56]. Table 3 shows some characteristics of these datasets. These datasets were chosen because they span a different range of structural characteristics and they are commonly used in related work in the literature. The DBLP dataset is an example of a real-world XML database containing publication information in the Digital Bibliography and Library Project (DBLP) website. In this dataset, the structural difference between elements with identical types at the same level is small. The XMark dataset is an example of an XML synthetic database where most of its elements of the same type and which are

at the same level have different sets of descendent elements and consequently different sub-tree structures. The Ssplays is another example of a real-world database containing Shakespeare’s plays. In this dataset, most of the elements of the same type and which are at the same level have the same set of descendent elements but with different counts. The Uniprot dataset is also a real world database that acts as a comprehensive and freely accessible knowledgebase of protein sequence and functional information. This dataset shows more irregularities in elements structures than the DBLP and the Ssplays. The queries were generated randomly by a random query generator that we implemented. For each dataset, we generated 100 random queries of each query type as follows: 100 twig queries consisting of only parent-child (P-C) axis, 100 twig queries consisting of only ancestor-descendent (A-D) axis, 100 P-C linear queries, and 100 A-D linear queries.

Table 3: Some characteristics of the adopted datasets

Dataset	Size (MB)	Total Elements	Unique Elements	Max Depth
DBLP	153	3567298	33	6
XMark	112	1666315	74	12
Ssplays	7.52	179690	22	7
Uniprot	136	2541733	70	6

6.2. Summary-Tree Generation Time

The summary-tree generation time is affected by the selected value for the parameter K . As it can be seen from Eq. 7, the number of multiplication and addition operations in the fingerprint function is proportional to K . In our experiments, the maximum value for K was found to be 11 for both the DBLP and Ssplays datasets; and the corresponding elapsed times for the generation of the summary trees were 50 seconds and 1.6 seconds,

respectively. The Uniprot on the other hand, had a maximum K of 28 and the elapsed time for summary generation was 31 seconds. For the XMark dataset, the maximum value for K was found to be 3420, which is very large, and the elapsed time for the generation of the summary tree was around 960 seconds. In order to reduce the elapsed time for the XMark dataset, we should use a smaller value for K ; but a very small K also increases the computation of the proposed approach because it results in more collisions, and consequently more time will be needed to check the sub-trees of colliding nodes to see if they can be merged or not. Therefore, we started by examining the number of nodes with K distinct siblings for various values of K in the range 1 to 3420 (which is the maximum). The results are as shown in Figure 21. This figure shows that there is a very small number of nodes having 12 or more distinct siblings. So, by ignoring these nodes and setting K to the maximum value for the remaining nodes, the generation time is reduced significantly to less than 46 sec. We also tried other approaches for setting the value of K , e.g., taking the average value and taking the average value after ignoring nodes that have low counts. Although the generation time was better than the maximum but still it was much higher than the maximum after ignoring the low-count nodes. Table 4 shows the summary-tree generation times for different values of K for various datasets. In this table, Max and Avg represent the highest and average values for K whereas Max* and Avg* refer to the cases with the highest and average values after ignoring the nodes with the low counts. Table 4 also shows the number of collisions in each case. For very small values of K , such as in Avg and Avg*, the numbers of collisions were 2172 and 14314 respectively. The number of collisions was drastically reduced when K was set to Max, and slightly increased when K was set to Max*.

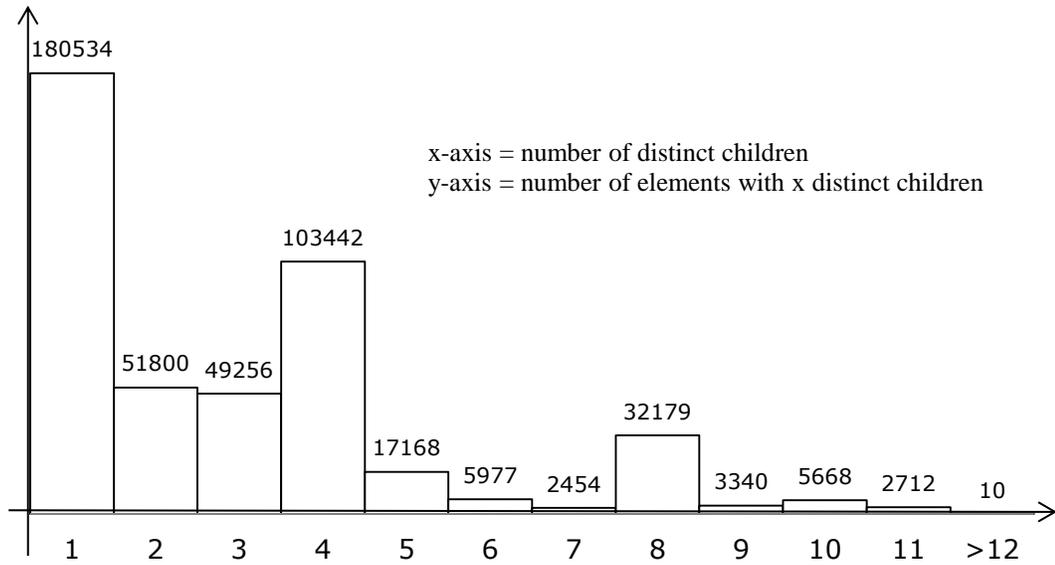


Figure 21: Count of elements having a given number of distinct children

Table 4: Summary-tree generation times and number of collisions for different datasets and various approaches of selecting K

Dataset	K	Type	collisions	Elapsed Time (sec)
XMark	3420	Max	84	960
XMark	11	Max*	86	21
XMark	3	Avg	21712	359
XMark	6	Avg*	14314	322
DBLP	11	Max	0	50
Ssplays	11	Max	0	1.6
Uniprot	28	Max	111	31

To avoid the cost incurred to find the value of K , we removed the call of the *getKBM* function from *SynopGen* and we set K to a random number between 3 and 50; and we set the height of the data-tree to 8. Mlynkova et al. [53] in their work analyzed more than 200,000 XML documents and found that the average depth in more than 99% of them is 8. They also found that the average fan-out (children) of an element is 9. Therefore, we believe that the numbers we chose for K and the height are reasonable according to their findings. Figure 22 shows the elapsed time of *SynopGen* for different values of K

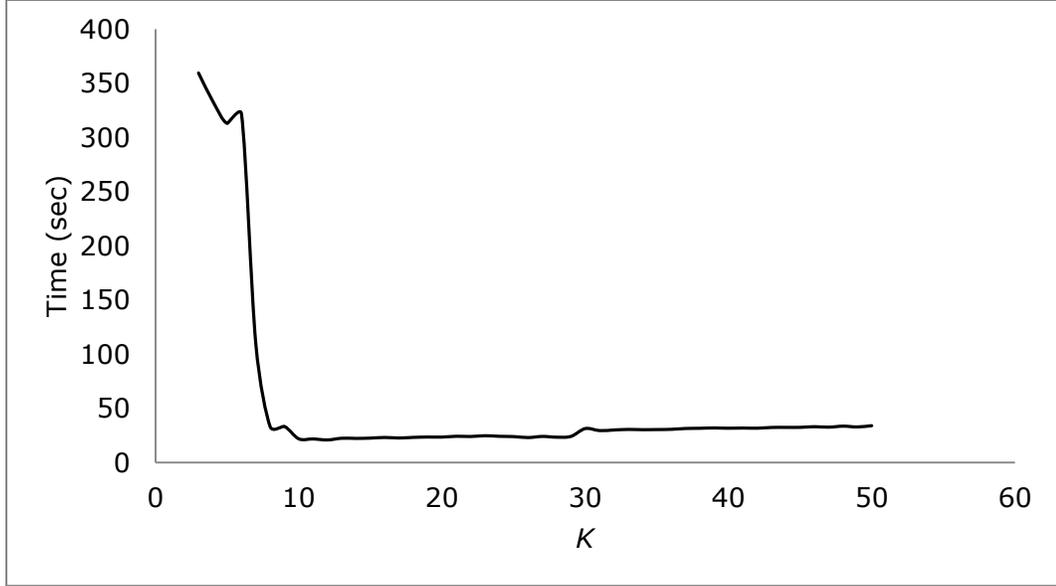


Figure 22: Impact of randomly selecting K in range 3 to 50 on the summary-tree generation time for the proposed approach

between 3 and 50. As shown in the graph, when K is less than 11 the generation time increases. This is because the number of collisions increases when K is less than the Max or Max*, as shown in Table 34, and consequently the number of merge operations that *SynopGen* needs to perform increases as well. On the other hand, as the value of K reaches 11 (Max*) the number of collisions decreases and so does the generation time. Moreover, as K grows larger than the Max* the generation time slightly increases. This is because the computation time of Eq. 7 increases with the value of K .

6.3. Estimation Error Rate and Storage Size

For each of the above-mentioned datasets and the generated queries, we compared the performance of our proposed approach with that of the Sampling and TreeSketch approaches in terms of the percentage estimation error rate (*ER*) and the storage size ratio (*SSR*). The (*ER*) measure is defined by the mean absolute relative error as:

$$ER = \left(\frac{1}{N_q} \sum_{i=1}^{N_q} \frac{||T_q||_i - ||\hat{T}_q||_i}{||T_q||_i} \right) * 100 \quad (13)$$

where N_q is the number of queries, and $||T_q||_i$ and $||\hat{T}_q||_i$ are the actual and estimated numbers of matches for the i^{th} query. On the other hand, *SSR* is given by:

$$SSR = \frac{size(T_s)}{size(T_d)} * 100 \quad (14)$$

where $size(T_d)$ and $size(T_s)$ are the sizes of the XML data tree and the corresponding summary tree, respectively.

The ER values of the proposed technique, the Sampling algorithm, and the TreeSketch technique are depicted in Figure 23 to Figure 25 for linear, existential, and regular twig queries, respectively. Figure 26(a) shows the *SSRs* used in the experiments by the three algorithms. *SynopTech* algorithm achieved smaller *SSR* values than the TreeSketch except for the Uniprot dataset where the TreeSketch has a slightly smaller *SSR*. This is because in the other datasets elements with identical tags and sub-tree-structures tend to have different sub-tree counts. Therefore, the TreeSketch would consider these elements as non-identical and add additional nodes to the summary. On the other hand, elements with identical tags and sub-tree-structures in the Uniprot dataset tend to have identical sub-tree counts as well, and since the TreeSketch uses a graph to store the summary, which is normally more succinct than a tree structure, it requires less number of nodes to store the summary and consequently less storage for this dataset. In our experiments we kept the *SSR* of all algorithms for each dataset the same except for the Uniprot dataset where we set the *SSR* value for the sampling algorithm to that of the proposed algorithm (0.07%) while keeping the TreeSketch at (0.05%) as shown in Figure 26(a).

As can be seen from the figures, the *SynopTech* approach outperformed the Sampling approach in all types of queries and datasets. This is because at low *SSR* values, the size of the sample tree was too small to capture enough structural information of the original data tree. Also, since samples are collected randomly, the error rate on a specific query set depends on whether the randomly selected sample tree contains the structural information needed to satisfy the queries on that specific query set. This indicates that the error rate on a specific query set might change every time a new sample tree is generated. On the other hand, our approach consistently showed a very low error rate on all types of queries and datasets especially with the DBLP and Ssplays datasets (uniform datasets), the proposed algorithm used a memory budget of 0.005 *SSR* and 0.3 respectively. These values of memory budget were too small for the Sampling approach to capture considerable amount of representative samples from the data tree. That is why its error rate reached around 98% whereas the worst error rate of the proposed approach was only 0.8% on the Ssplays dataset. For instance, Table 5 and Table 6 show some sample twig queries and their regular estimates. As can be seen in Table 6 the Sampling algorithm could not produce any estimates for Q1, Q2, and Q3 resulting in an error rate of 100% for these queries. Also, the error rates for Q1 by the *SynopTech* and TreeSketch were greater than 4%. This is because the counts of the leaf nodes “LINE” and “STAGEDIR” in the source data tree have irregular distributions under the parent element “SPEECH” compared to the other elements in the source data tree (e.g. “TITLE” elements under “SCENE”). Therefore, during the summary generation both approaches could not accurately capture this irregularity resulting in relatively high error rates for this query compared to the overall low error rates both approaches achieved on the Ssplays dataset.

For Q2 on the other hand, *SynopTech* managed to capture the complete distribution of the elements in the query achieving a 0% error rate while the TreeSketch showed an error rate of 40%. This is because for the DBLP dataset the TreeSketch needed an *SSR* value of 0.03 to capture the accurate distribution of elements and in our experiments this value was reduced 6 times to match the *SynopTech SSR* , 0.005, causing the TreeSketch to lose a great deal of the distribution information. Moreover, although the TreeSketch outperformed the sampling algorithm in some XMark queries, such as Q5 in Table 6, the Sampling algorithm showed a better overall accuracy on the XMark at *SSR* 1.7. This is also evident in Q4 in Table 6 where the TreeSketch error rate was 19.7% while the error rate for the Sampling approach was only 2.8%. This is because at the *SSR* value of 1.7 the sampling algorithm had enough storage space to store representative samples of the source data tree resulting in an improved estimation accuracy. For the Uniprot dataset, *SynopTech* showed a slightly higher error rate than the TreeSketch. This is because the TreeSketch managed to capture the complete distribution of elements at a lower *SSR* value and therefore achieved an error rate of 0 % on all types of queries on that dataset. This can be seen also in Q6 and Q7 in Table 6 where the TreeSketch had a 0% error rate while *SynopTech* had a slightly higher error rate for these queries. Even though the TreeSketch had a perfect estimation for the Uniprot dataset, *SynopTech* error rate remained below 1%. Moreover, *SynopTech* managed to generate the Uniprot summary tree more than 5 times faster than the TreeSketch and therefore this slight improvement in the accuracy came at the expense of the summary generation time as shown in Table 7.

Table 5: Sample twig queries

ID	Query	Dataset
Q1	//SCENE[/TITLE]/SPEECH[/LINE]/STAGEDIR	Ssplays
Q2	//proceedings[/volume]/series	DBLP
Q3	//PLAY[/SUBHEAD]/LINE	Ssplays
Q4	//text[/emph]/bold	XMark
Q5	//person[/homepage]/profile[/business]/gender	XMark
Q6	//reference/source[/tissue]/strain	Uniprot
Q7	//comment/subcellularLocation[/location]/topology	Uniprot

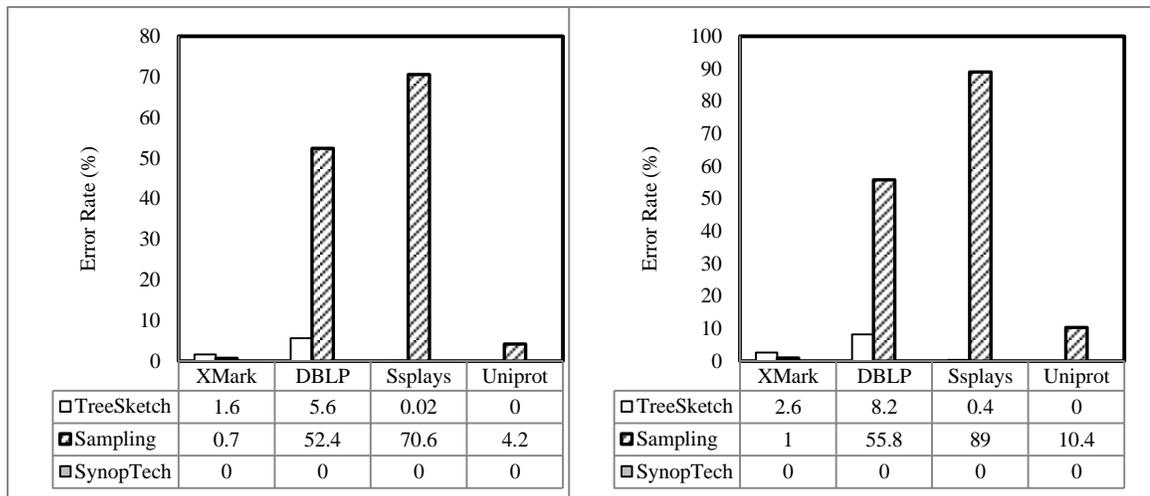
Table 6: Regular twig estimates for the queries in Table 5

ID	actual regular count	SynopTech estimate	SynopTech error(%)	TreeSketch estimate	TreeSketch error(%)	Sampling estimate	Sampling error(%)
Q1	14273	13374.6	6.3	13656	4.3	0	100
Q2	1478	1478	0	874.2	40.9	0	100
Q3	47295	47295	0	48606.7	2.8	0	100
Q4	66668	65906.2	1.1	53516.9	19.7	68509.4	2.8
Q5	3189	3189	0	3212.3	0.7	3438	7.9
Q6	1443	1438.7	0.3	1443	0	1253.8	13.1
Q7	2985	2985.05	0.002	2985	0	2914	2.4

In summary, we showed that the *SynopeTech* algorithm had a high estimation accuracy for all datasets and its error rate did not exceed 1% in any of the datasets while the TreeSketch error rate exceeded 15% for the twig queries on the DPLP since it could not capture the accurate distribution of elements at the *SSR* value of 0.005%. Moreover, *SynopeTech* outperformed the other two approaches on all datasets and for all types of queries in terms of accuracy and storage requirement. The only exception was the Uniprot dataset where the *SynopTech* had a slightly higher error rate than the TreeSketch but a faster generation time. Also, even in generally uniform datasets such as the Ssplays and DBLP, some elements can exhibit a slight irregularity in terms of their count distribution which can deteriorate the estimation accuracy for regular twig queries involving such elements as shown in Table 6. One way to improve the estimation accuracy for regular twig queries is to complement the structural synopsis with histograms to capture irregularities in the source data trees without greatly impacting the storage which is something we aim to study in our future work. Also, note that *SynopeTech* needed more memory budget to summarize the XMark dataset than it required with the other datasets because of its irregular structure. However, as the size of the XMark source data tree increased the *SSR* needed was decreasing as depicted in Figure 26(b). In the following chapter, we propose several techniques to handle selectivity count estimation when the memory budget is small or limited.

Table 7: Summary generation time

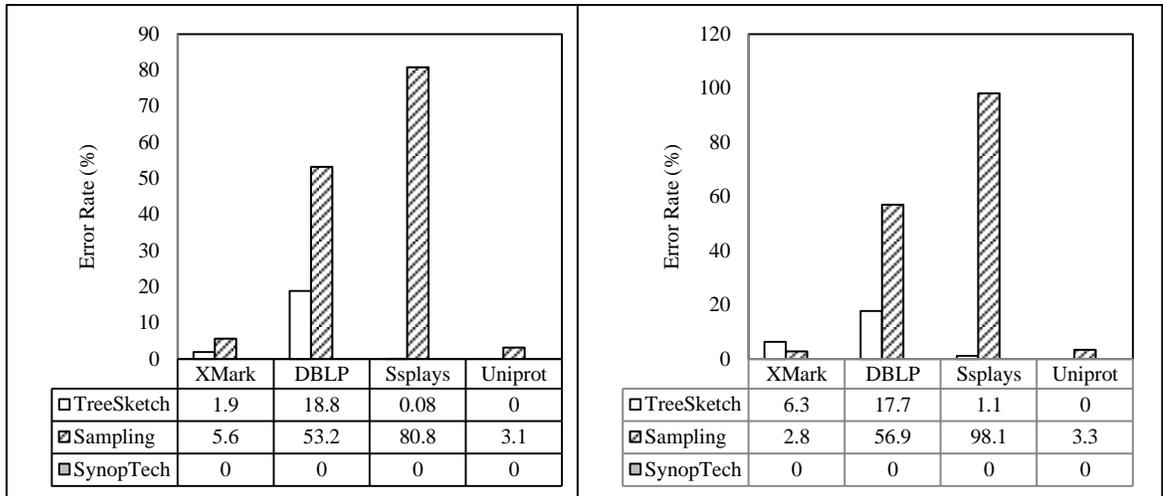
Dataset	SynopTech gen time (sec)	TreeSketch gen time (sec)	Sampling gen time (sec)
Ssplays	1.7	1.1	1
DBLP	53	30.8	11.3
XMark	53.8	455	7.2
Uniprot	30.2	163	9.7



(a) P-C Queries

A-D Queries

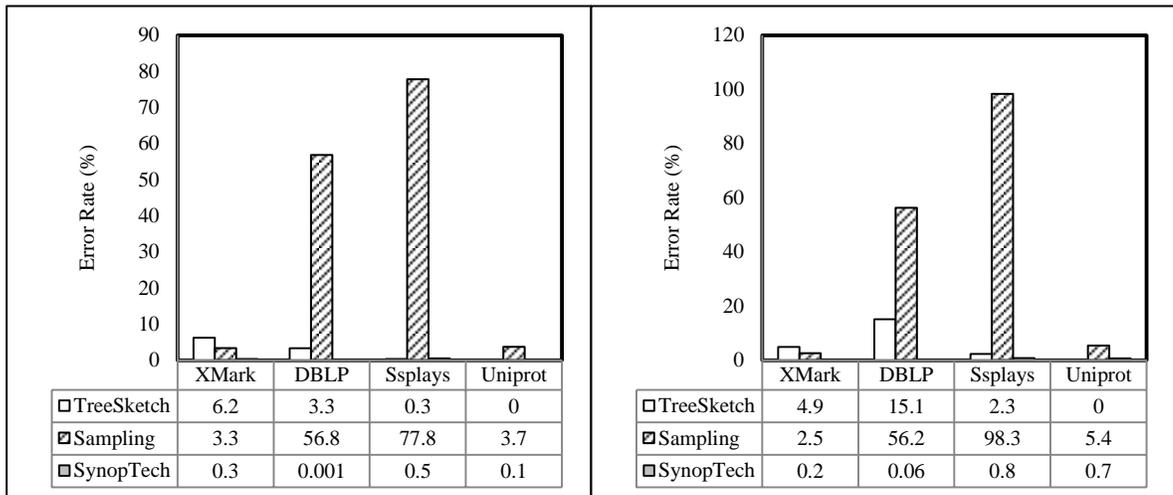
Figure 23: ER for proposed, Sampling, & TreeSketch on linear queries



(a) P-C Queries

A-D Queries

Figure 24: ER for proposed, Sampling, & TreeSketch on existential twig queries



(a) P-C Queries

A-D Queries

Figure 25: ER for proposed, Sampling, & Treesketch on regular twig queries

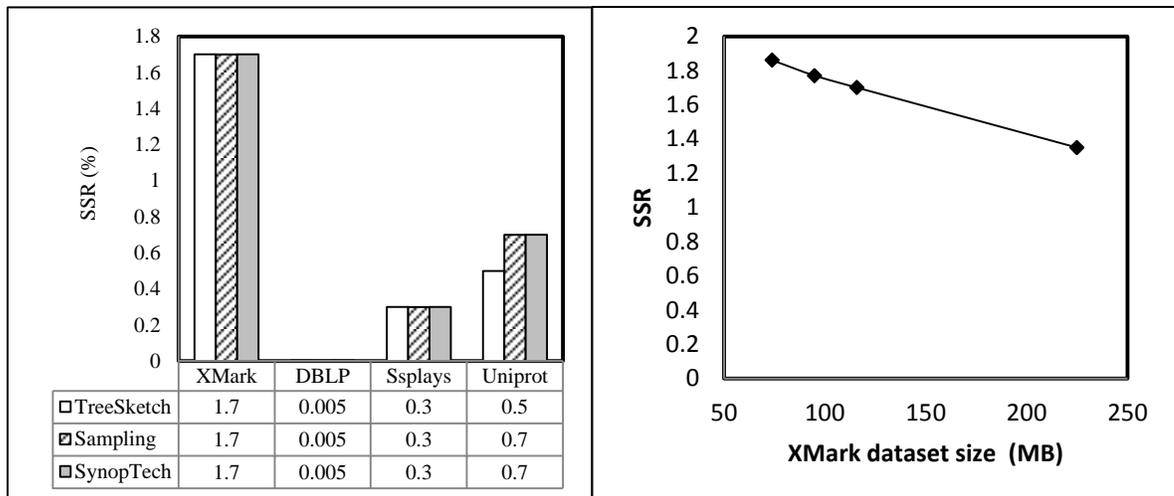


Figure 26: Required storage: a) Comparison of *SSR* for *SynopTech*, *Sampling*, and *TreeSketch* algorithms, and b) Effect of XMark dataset size on the *SSR* for *SynopTech* algorithm

CHAPTER 7

HANDELING STORAGE LIMITATION

In this chapter we propose several techniques to handle selectivity count estimation when the storage budget is limited. First, we explain how the summary tree generated by the *SynopGen* algorithm can be reduced or pruned to fit the available storage budget. After that, we explain our first proposed technique, node-count ratio, to estimate the selectivity count using the pruned summary tree. Then, we explain how the statistical approach Node-Ratio-Node-Factor (NR-NF) [30] can be extended to support regular twig queries. Next, we present a hybrid approach for selectivity count estimation using the extended statistical approach and our proposed structural approach, *SynopTech*. We also provide the results of our experiments after the discussion of each approach and compare our hybrid approach with the Sampling approach [9] and the TreeSketch approach [38] in terms of scalability (storage vs. accuracy).

7.1. Pruning the Summary-Tree

The main idea of the pruning method is to keep removing the elements with the lowest counts until the summary tree fits in the allocated storage. This is shown in Algorithm 7. It takes a summary tree and the desired size as inputs and it outputs a reduced summary tree. To reduce the summary tree, Algorithm 7 first generates a sorted list of all distinct counts in the summary tree, lines 1-3. After that, it visits the nodes at each level, starting at the lowest level moving upwards, deleting the nodes with the lowest counts until the summary tree fits in the available storage budget, lines 4-17. Also, for each node type it

stores the ratio of the node count in the original summary tree to the node count in the new summary tree, lines 18-23. For instance, if the original summary tree T_s has 50 elements of type “ b ” and the reduced summary tree T_{s_r} has 25 elements of type “ b ” then the node-count ratio for element type “ b ” is $r_b = 2$. Note that this algorithm takes a linear time to visit the nodes in every level therefore its complexity cannot exceed $O(n \log n)$ for the sort operation, line 3, where n is the number of distinct node frequency in the summary tree which is at most the number nodes in the summary tree.

Definition 7.1 (*Node-count ratio*)

Given two summary trees T_s and T_{s_r} , where T_{s_r} is generated by reducing the element counts in T_s , the count ratio for element type n is given by:

$$r_n = \frac{|n|_s}{|n|_{s_r}} \tag{15}$$

where $|n|_s$ refers to the count of n nodes in T_s and $|n|_{s_r}$ refers to the count of n node in T_{s_r} .

7.2. Selectivity Count Estimation With Node-Count Ratio

To estimate the selectivity count $||\hat{T}_q||$ of a given query T_q rooted at element type n , we first apply the *SynopCalc* algorithm explained in Chapter 5 using the reduced summary tree T_{s_r} to generate the initial estimate of the count of T_q in the data tree T_d . We refer to the estimate generated using the reduced tree as $||\hat{T}_q||_{s_r}$. After that we multiply the answer with the node ratio of the query root r_n . Therefore, the estimated selectivity count of T_q in T_d is given by:

$$||\hat{T}_q|| = ||\hat{T}_q||_{s_r} * r_n \tag{16}$$

Algorithm 7: Pruning

Input: T_s , $size$ **Output:** T_{sr} , $countRatios$

```
1:  $frequencies \leftarrow T_s.getUniqueFrequencies()$ ,  $countRatios[,] \leftarrow null$ 
2:  $TreeSize = T_s.size()$ 
3:  $frequencies.sort$ 
4: for all  $fr \in frequencies$ 
5:   for  $l = T_s.height$ ;  $l < 0$ ;  $l--$ 
6:     for all  $n_i \in T_s[l]$ 
7:       if  $n_i.count == fr$ 
8:          $TreeSize = TreeSize - n_i.size()$ 
9:          $T_s.Remove(n_i)$ 
10:      end if
11:      if  $TreeSize \leq size$ 
12:         $T_{sr} = T_s$ 
13:        return  $T_{sr}$ 
14:      end if
15:    end for
16:  end for
17: end for
18: for all distinct  $n_j \in T_{sr}$ 
19:   for all distinct  $n_i \in T_s$ 
20:      $countRatios.add(n_j, \frac{n_i.count}{n_j.count})$ 
21:   end for
22: end for
23: return  $countRatios$ 
```

Consider the example in Figure 19, the selectivity count is generated using the summary tree T_s and it indicates that there are 5 sub-trees rooted at “A” in the data tree T_d that match the structure of the query in Figure 19(a). If this estimate is generated using the reduced summary tree, T_{sr} , then T_d could have more “A” nodes with the underlying substructure depicted in Figure 19(a). Since we know that the number of “A” elements in T_s is reduced by $\frac{1}{r_A}$ during the generation of T_{sr} , we multiply the selectivity count generated using T_{sr} by r_A to estimate the number of matches in T_d .

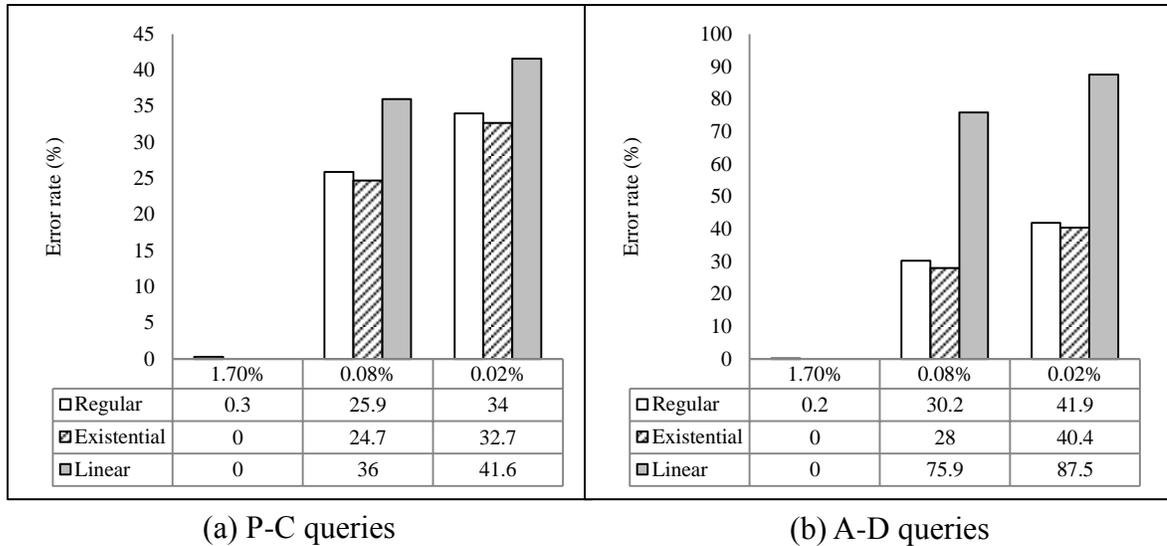


Figure 27: P-C & A-D error rates for the node-factor method on XMark with different summary sizes

7.2.1. Experiments

In order to test the performance of the above method, we used the same query sets discussed in Chapter 6 for the XMark dataset. The query sets include 100 P-C path queries, 100 A-D path queries, 100 P-C twig queries, and 100 A-D twig queries. Figure 27 shows the impact of the storage on the accuracy using the node-count ratio method on each query set and for all the three types of supported queries, namely linear path, existential twig, and regular twig queries on the XMark dataset.

It can be seen that this approach produces a high error rate when the summary size is very small. This is because the summary is generated by removing elements with low counts from the tree and as the summary gets smaller, it fails to estimate the selectivity counts for element types with very low or close to zero counts. For this reason, we will present a hybrid approach that is capable of estimating the selectivity counts of element types with

very low counts in the summary tree. First, we will explain the statistical approach that is going to be used in our hybrid approach.

7.3. Statistical Approach for Selectivity Count Estimation

We implemented the technique that is based on the node ratios concept explained in [30]. Moreover, we extended this approach to make it applicable for regular twig queries since regular twig queries are not addressed in [30]. For this purpose we store all parent-child binary-paths along with the ratios of the parent and child nodes in the binary-path. For example, for a binary-path “ p/c ” we store the ratio of the parent node p , denoted $\tilde{R}(p, p/c)$, as:

$$\tilde{R}(p, p/c) = \frac{\hat{c}(p, p/c)}{\hat{c}(p, T_d)} \quad (17)$$

where $\hat{c}(p, p/c)$ and $\hat{c}(p, T_d)$ are the count of p nodes in “ p/c ” and the count of p nodes in the whole data tree respectively. Similarly, we store the ratio of the child node c , denoted $\tilde{R}(c, p/c)$, as:

$$\tilde{R}(c, p/c) = \frac{\hat{c}(c, p/c)}{\hat{c}(c, T_d)} \quad (18)$$

7.3.1. Regular Twig and Linear Path Queries

In order to estimate the selectivity count of a regular twig or a simple path query q with k nodes n_1, n_2, \dots, n_k , we use the following equation:

$$\|T_q\| = \prod_{i=1}^k \left(\frac{\hat{c}(n_i, T_d) \times \tilde{R}(n_i, p_i/n_i) \times \prod_{j=1}^{|n_i|} \tilde{R}(n_{i,j}, n_i/n_{i,j})}{\hat{c}(p_i, T_d) \times \tilde{R}(p_i, p_i/n_i)} \right) \quad (19)$$

where p_i represents the parent node of node n_i ; $n_{i,1}, n_{i,2}, \dots, n_{i,|n_i|}$ represent the child nodes of n_i ; and $|n_i|$ represents their count.

In Eq. 19, the ratio inside the first product will estimate the number of n_i elements under p_i elements in T_q . Note that if n_i is the query root, then $\check{R}(n_i, p_i/n_i)$ and the denominator $\hat{c}(p_i, T_d) \times \check{R}(p_i, p_i/n_i)$ are set to 1 since the root node has no parent. Also, if n_i is a leaf node, then the product of $\prod_{j=1}^{|c|} \check{R}(n_{i,j}, n_i/n_{i,j})$ is set to 1 since a leaf node has no children. Note that the ratios for A-D paths are calculated recursively from the stored ratios of the binary P-C paths. In the following subsection we explain how existential twig queries are estimated in [30] and we also explain how Eq. 19 can be used for the same purpose.

7.3.2. Existential Twig Queries

In [30], the node factor, F , of a target node is defined as the ratio of the frequency of the target node in a given path to the frequency of the root node in the same path. For example, the node factor of t in a binary-path $P = "r/t"$ is:

$$F(t, P) = \frac{\hat{c}(t, P)}{\hat{c}(r, P)} \quad (20)$$

Which is equivalent to the ratio inside the outer product in Eq.19. In [30] the selectivity count of a binary-path $P = "r/t"$ is:

$$\hat{c}(p_i, T_d) \times \check{R}(r, P) \times F(t, P) \quad (21)$$

where $\check{R}(r, P)$ refers to the node ratio of the root node r in P and $F(t, P)$ is the node factor of the target node t in P .

Two binary paths can be joined if the leaf node of one of them is of the same node type as the root node of the other. For example, if $P_1 = "r_1/t_1"$ and $P_2 = "r_2/t_2"$, and if t_1 and r_2 have the same name, then P_1 and P_2 can be joined to form $P_1 + P_2 = r_1/t_1/t_2$. The node ratio and the node factor of a node in $P = P_1 + P_2$ are computed as follows. $F(t_1, P) = F(t_1, P_1) \times \check{R}(t_1, P_2)$. If $F(t_1, P) \geq 1$ then, $\check{R}(r_1, P) = \check{R}(r_1, P_1)$ and $F(t_2, P) = F(t_1, P) \times F(t_2, P_2)$. If $F(t_1, P) < 1$ then, $\check{R}(r_1, P) = \check{R}(r_1, P_1) \times F(t_1, P)$ and $F(t_2, P) = F(t_2, P_2)$. The selectivity count of $P_1 + P_2$ is then computed as:

$$\|P_1 + P_2\| = \hat{c}(r_1, T_d) \times \check{R}(r_1, P) \times F(t_2, P) \quad (21)$$

For existential twig queries, the node ratio of the root node r is calculated by first calculating the node ratio for r in every outgoing path from r and then calculating their product.

The general selectivity count equation in [30] for a given query Q with a root node n_1 and a target node n_q is given by:

$$\|\hat{T}_q\| = \hat{c}(n_1, T_d) * \check{R}(n_1, Q) * \prod_{i=1}^{|q-1|} F(n_i, P_i) \quad (22)$$

where P_i is a path from the query root n_1 to the target node n_q . From the above we can see that only node factors that are in the path to the target node n_q are involved in the selectivity count estimation. Moreover, from the multiple paths formula we can see that whenever the node factor is less than 1 it contributes to the new value of the node ratio for the query root. Therefore, to estimate the selectivity counts of existential twig queries using Eq. 19 for regular twigs and linear paths, we need to filter out the node factors that do not contribute to the answer which are the node factors that do not fall in the path to the target node T and at the same time larger than 1. This can be done by dividing $\|T_q\|$

by the product of the node factors meeting these conditions. Alternatively, to estimate the selectivity counts of existential twig queries we apply Eq. 19 only on the set of nodes $M = n_1, n_2 \dots n_q$ such that for $0 < i \leq q$, n_i belongs to M only if

$$\frac{\hat{c}(n_i, T_d) \times \tilde{R}(n_i, p_i/n_i) \times \prod_{j=1}^{|n_i|} \tilde{R}(n_{i,j}, n_i/n_{i,j})}{\hat{c}(p_i, T_d) \times \tilde{R}(p_i, p_i/n_i)} < 1$$

OR

n_i is in the path from the query root R to the target node T .

7.3.3. Experiments

In order to test the performance of the above approach, we used the same query sets discussed in Chapter 6 for the XMark dataset in Table 3. The query sets include 100 P-C path queries, 100 A-D path queries, 100 P-C twig queries, and 100 A-D twig queries. Figure 28 shows the accuracy of the extended statistical approach on each query set and for all the three types of supported queries, namely linear, existential twig, and regular twig queries. Note that the statistics generated for the XMark dataset required only 1.7KB of storage which is equivalent to *SSR* of 0.0014%. The figure shows that the statistical approach gives an acceptable error rate given a very small storage budget (i.e. 1.7KB). In the following section we present a hybrid approach for selectivity count estimation based on the statistical approach and our proposed structural approach.

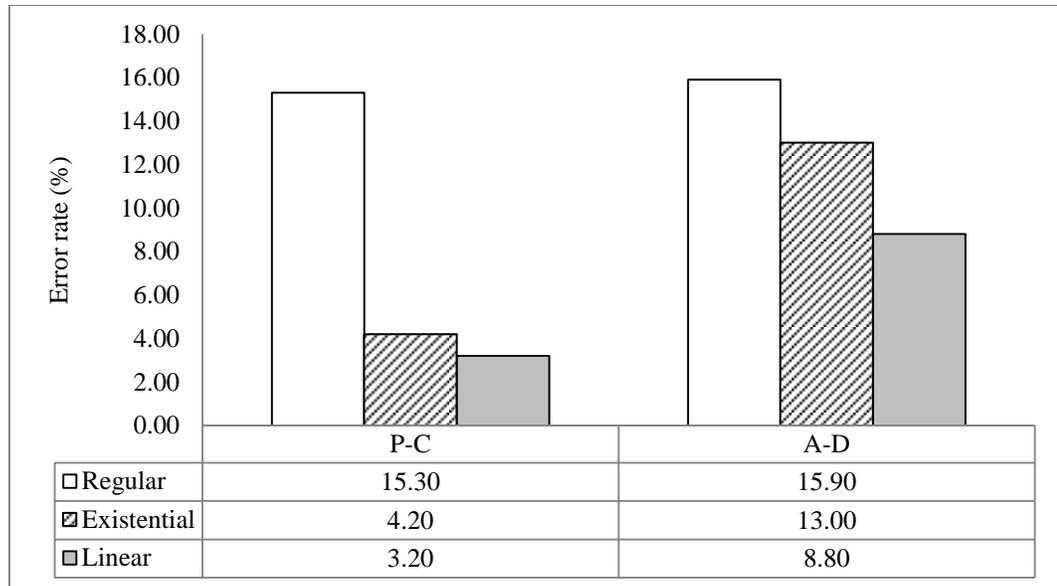


Figure 28: Error rates for the extended statistical approach

7.4. Hybrid Approach for Selectivity Count Estimation

We implemented a hybrid solution for selectivity count estimation using our summary approach and the statistical approach explained in the previous section. Both approaches are not scalable by themselves in terms of storage. In other words, although the statistical approach requires a very small memory budget, the error rate cannot be improved if additional space is available since this approach does not make use of this additional space. Similarly, our summary approach requires large space for irregular data trees such as the XMark and it cannot estimate the selectivity count if the available storage budget is not sufficient to store the summary tree. That being said, our proposed hybrid approach aims to take the advantage of both approaches and combine them to create a scalable selectivity count estimation system. We propose two methods to combine the two approaches, namely summary delta and query delta. In what follows we explain these two methods

Summary-delta: The basic idea is to calculate the selectivity count first using the reduced summary tree and then apply the statistical approach on the difference between the query-root node count in the original summary tree and the reduced summary tree. The count of the query-root node in the original summary tree is acquired using Eq. 16. For instance, if we have a query $a/b/c$ where the count of “ a ” in the original summary is 50 and 10 in the reduced summary, then we apply our summary approach on the 10 “ a ” nodes existing in the reduced summary tree. After that, we apply the statistical approach on the 40 remaining “ a ” nodes. Finally, we add up the answers from both approaches to get the final selectivity count.

Query-delta: Like the summary-delta method, we first use the reduced summary tree to generate the initial selectivity count and then compensate for the missing nodes by applying the statistical approach. The difference is that rather than using the delta between the reduced and actual summary trees we use the query delta. This means, after we apply the summary approach, we check how many query-root nodes in the reduced summary tree actually participated in producing the selectivity count estimate. After that, we use Eq. 19 to estimate the count of the query-root nodes participating in producing the selectivity count using the statistical approach. We then set the frequency of the query-root in Eq. 19 to the difference between the two counts and then apply Eq. 19. Finally we sum up the selectivity counts generated by both approaches. For example, if we have a query $a/b/c$ where the count of “ a ” in the original summary is 50 and 10 in the reduced summary. Say after applying our summary approach on the 10 “ a ” nodes we found that only 4 actually participated in generating the selectivity count. We then apply Eq. 19 on the 50 “ a ” nodes which, for example, estimated that the count of “ a ” in the query is only 20. Then we reduce this estimate (20) by 4 since we already estimated the selectivity

count for these 4 nodes using the structural approach and generate the selectivity count estimate for the 16 remaining nodes by setting the frequency of “ a ” to 16 in Eq. 19 and then applying Eq. 19 on the rest of the nodes. Finally, we add up the selectivity count generated by our summary approach on the 4 nodes with the selectivity count generated by the statistical approach on the 16 nodes to get the final estimate. This is explained by Algorithm 8, *CalcSel*, which takes as an input a reduced summary tree and a query and it outputs the selectivity count of the input query using the reduced summary tree. In lines 2-6, *CalcSel* estimates the ratio of the query-root node in the input query by multiplying the parent-child ratios of all the nodes in the query. In line 7, *CalcSel* estimates the count of query-root node satisfying the query by multiplying the query-root ratio by the query-root count in the data tree. In lines 8-9, *CalcSel* calls the structural component of the hybrid system to get its estimate for the query-root elements satisfying the query and calculates the query delta, line 9. Finally, in lines 10-25, *CalcSel* applies the selectivity count estimation equation (Eq.19) on the query-delta taking into consideration the type of the query (i.e. existential, regular or linear) to calculate and then output the estimated query selectivity count. Note that Algorithm 8, performs two top-down linear scans of the input query. The first scan, lines 2-6, is performed to calculate the query-root ratio, and the second scan, lines 10-24, to estimate the selectivity count. In both scans every query node is visited at most twice, one as a parent and another as a child to calculate its ratio and count in the query which means that every scan requires at most $2M$ operations or $O(M)$ where M is the number of nodes in the input query. Thus, the overall complexity of *CalcSel* depends on the complexity of the structural estimator *SynopeCalc*, line 8, and the query processor chosen as explained in Chapter 5.

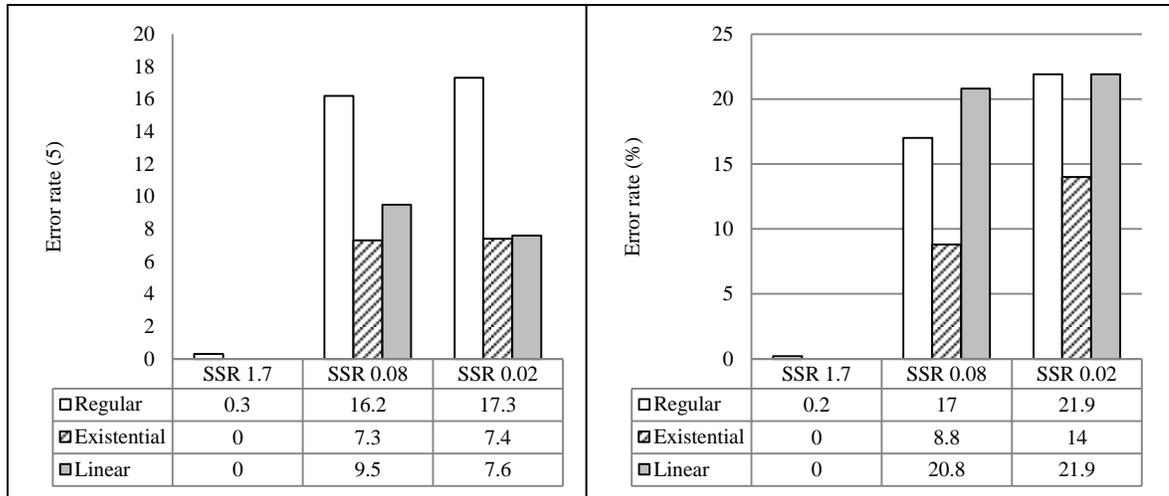
Algorithm 8: CalcSel

Input: T_{sr}, T_q **Output:** *SelectivityCount*

```
1: SelectivityCount  $\leftarrow$  1; rootRatio = 1; queryRootCount = 0;
2: for all  $n_i \in T_q.nodes$ 
3:   for all  $n_{i,j} \in n_i.children$ 
4:     rootRatio = rootRatio  $\times \hat{R}(n_{i,j}, n_i/n_{i,j})$ 
5:   end for
6: end for
7: queryRootCount = rootRatio  $\times \hat{c}(T_q.root, T_d)$ 
8: StructuralSelectivity = SynopCalc( $T_{sr}, T_q$ )
9: queryDelta = queryRootCount – StructuralSelectivity.rootCount
10: for all  $n_i \in T_q.nodes$ 
11:   childrenRatio = 1;
12:   for all  $n_{i,j} \in n_i.children$ 
13:     childrenRatio = childrenRatio  $\times \hat{R}(n_{i,j}, n_i/n_{i,j})$ 
14:   end for
15:   if  $n_i == T_q.root$ 
16:      $\hat{c}(n_i, T_d) = \text{queryDelta}$ 
17:     SelectivityCount = SelectivityCount  $\times \hat{c}(n_i, T_d) \times \text{childrenRatio}$ 
18:   else
19:      $\text{ratioVal} = \frac{\hat{c}(n_i, T_d) \times \hat{R}(n_i, p_i/n_i) \times \text{childrenRatio}}{\hat{c}(p_i, T_d) \times \hat{R}(p_i, p_i/n_i)}$ 
20:     if  $T_q.isExestintial() \ \&\& \ \text{ratioVal} \geq 1 \ \&\& \ n_i \notin T_q.criticalPath$ 
21:       continue
22:     else
23:       SelectivityCount = SelectivityCount  $\times \text{ratioVal}$ 
24:     end for
25: SelectivityCount = SelectivityCount + StructuralSelectivity
26: return SelectivityCount
```

7.4.1. Experiments

Summary-Delta Experiments: To test the performance of the summary-delta method, we used the XMark dataset and the same query sets discussed in Chapter 6 which include 100 P-C path queries, 100 A-D path queries, 100 P-C twig queries, and 100 A-D twig



a) P-C queries

a) A-D queries

Figure 29: P-C & A-D error rates on XMark using summary-delta

queries. Figure 29 shows the accuracy of the summary-delta method. The figure shows that the summary-delta approach has a higher error rate than the statistical approach shown in Figure 28. This is because the summary-delta assumes that all missing query-root node from the reduced summary tree can contribute to the query selectivity count estimate which often results in overestimates. Unlike the summary delta, the query delta rarely suffers from this issue and, as will be shown below, has a better performance than the summary delta. Therefore, we continued our experiments with the query-delta approach and compared it with the Sampling and TreeSketch approaches.

Query-Delta Experiments: In our experiments, we extensively tested and compared the query delta hybrid method with the same datasets, query sets, and experimental settings described earlier in Section 6.1. Figure 30 to Figure 33 show the summary generation times for the three approaches (Sampling, TreeSketch, and Hybrid) on the four datasets and for different *SSR* values. Note that the Sampling approach collects sample sub-trees

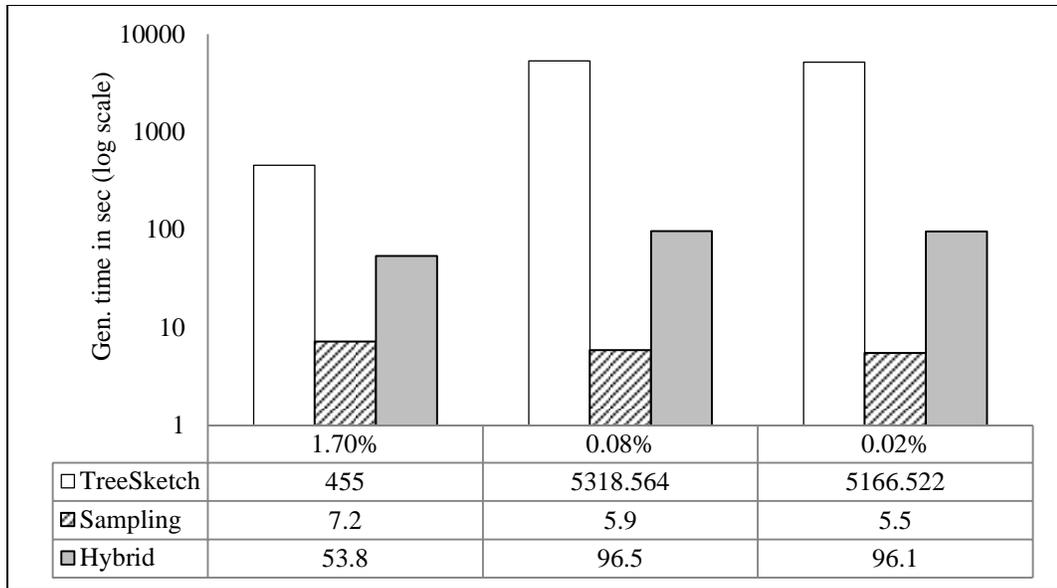


Figure 30: Gen time for Sampling, TreeSketch and Hybrid approaches on XMark

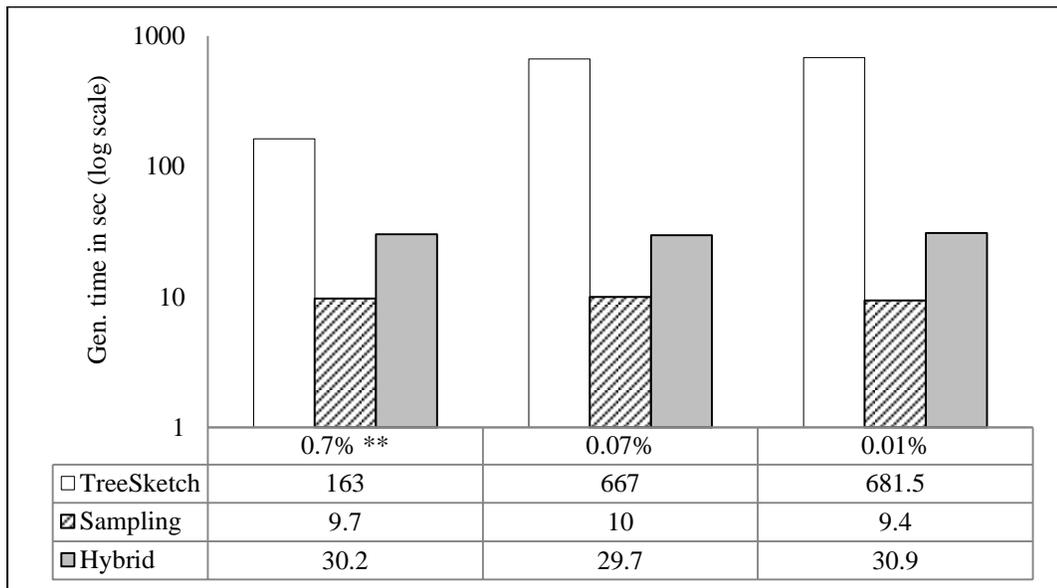


Figure 31: Gen time for Sampling, TreeSketch and Hybrid approaches on Uniprot (** TreeSketch needed only *SSR* of 0.5 to store the summary)

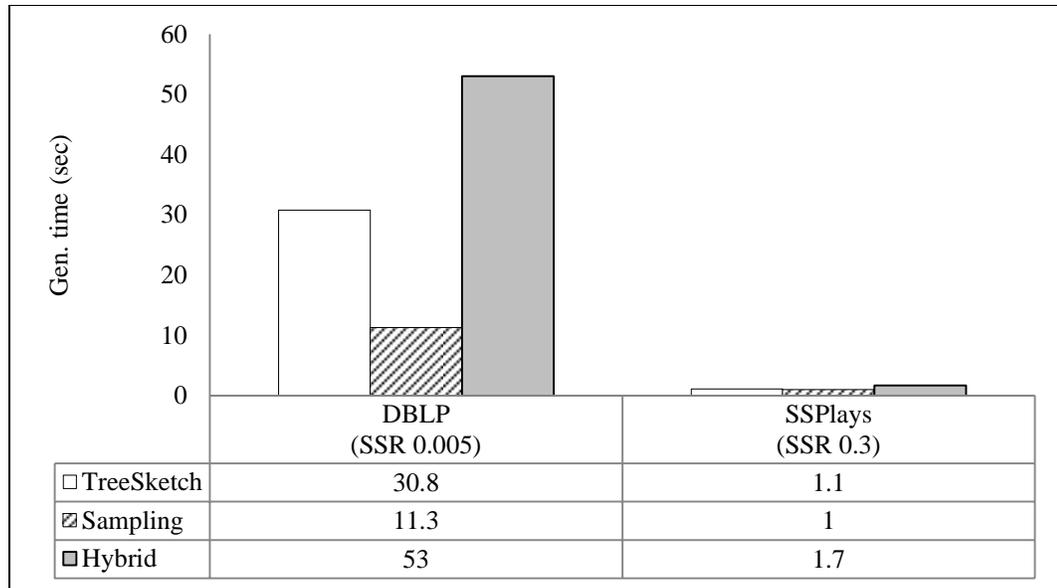


Figure 32: Gen time for Sampling, TreeSketch & Hybrid on DBLP & Ssplays

randomly and it does not perform any merge operations or structural comparisons to generate the summary tree and therefore it had the fastest generation time on all datasets. Also, it is noted that for the DBLP and Ssplays datasets (Uniform) the TreeSketch had a faster generation time than the Hybrid. The TreeSketch first generates the full summary tree which is called the original summary tree (stored as a graph). Then it reduces the summary by scanning the original summary graph level by level looking for nodes that are candidates for merge operations which are normally nodes with similar structures before applying the merges. During the original summary generation, the TreeSketch needed more summary nodes and storage, hence less merge operations, to capture the structure of the DBLP and Ssplays than what was needed by the Hybrid which managed to generate very small summary trees to capture almost the complete structures and distribution of elements in the source trees. This is why the TreeSketch had a faster generation time on those datasets. Moreover, the generated original summaries by the TreeSketch for the DBLP and Ssplays had to be reduced to match the *SSR* values of the

Hybrid (0.005 for DBLP and 0.3 for Ssplays) but since these datasets are relatively shallow (Table 3) compared to the XMark, and also the size reduction needed was not huge as in the cases of XMark and Uniprot, the overall generation time for the TreeSketch remained small. On the other hand, the XMark is a deeper dataset and therefore the process of generating merge candidates and applying merges greatly impacted the generation time for the TreeSketch especially for small *SSR* values such as 0.08 and 0.02. This is also true for the Uniprot, although it is not as deep as the XMark, the required size reduction caused more candidates generation and merge operations and consequently a higher generation time. In fact in the XMark, the Hybrid managed to generate the summary around 54 times faster than the TreeSketch at *SSR* value 0.02 and 22 times faster in the case of Uniprot at *SSR* value 0.01.

Uniform datasets (DBLP and Ssplays): The experiential results using the DBLP and Ssplays datasets are shown in Figure 23 to Figure 25 while Figure 33 shows the overall error rate on both datasets. The overall error rate represents the average error rate achieved by each approach on all types of queries. The Hybrid used a memory budget of 0.005 *SSR* for the DBLP and 0.3 for the Ssplays dataset. These values of memory budget were too small for the Sampling approach to capture considerable amount of representative samples from the data tree. This is why its overall error rate reached around 85.8% , Figure 33, whereas the worst overall error rate for the proposed approach was only 0.2% on the Ssplays dataset. Also, since samples are collected randomly, the error rate on a specific query set depends on whether the randomly selected samples contain the structural information needed to satisfy the queries on that specific query set. This indicates that the error rate on a specific query set might change every time a new

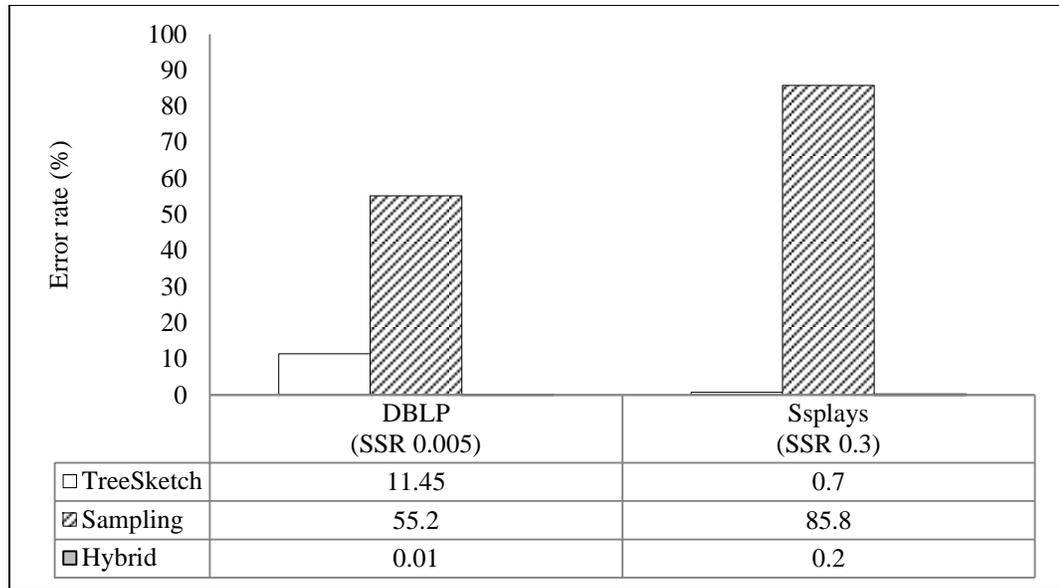


Figure 33: Overall error rates for all queries on DBLP and Ssplays

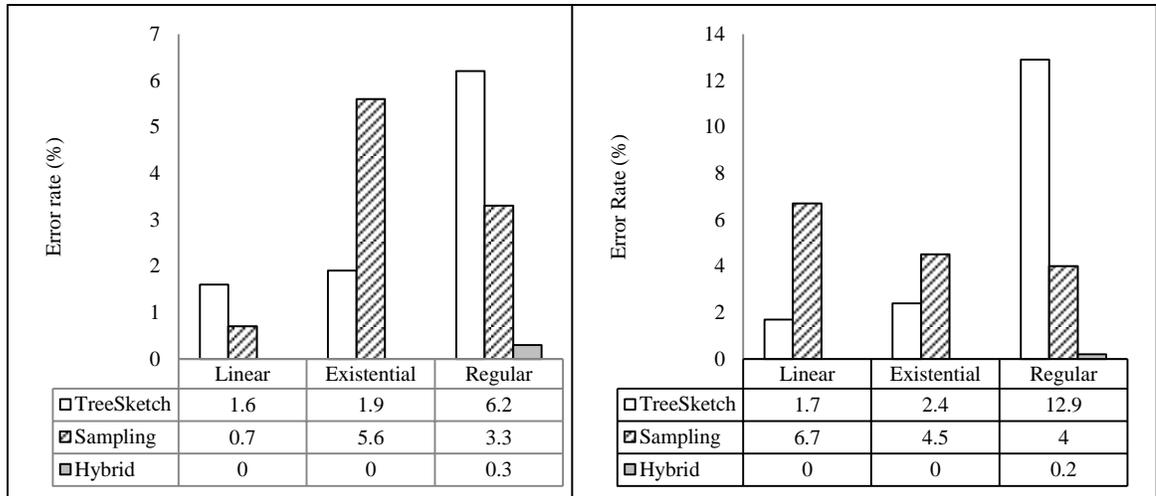
sample tree is generated. On the other hand, our approach consistently showed a very low error rate on all types of queries. In addition the TreeSketch error rate exceeded 15%, Figure 25, for the twig queries on the DBLP since it could not capture the accurate distribution of elements at the *SSR* value of 0.005% on the DBLP. Even though the proposed approach outperformed the TreeSketch on the Ssplays, the TreeSketch accuracy improved with the Ssplays and this is because the TreeSketch needed an *SSR* of 0.4 to capture the complete distribution of elements for the Ssplays which is a very close value to the one used in our experiments and therefore the loss of the elements distribution data was minimal. In the DBLP experiments on the other hand, the TreeSketch needed an *SSR* of 0.03 to capture the complete distribution data which is 6 times greater than the *SSR* used by the Hybrid approach and hence, the TreeSketch accuracy on the DBLP dataset was worse than its accuracy on the Ssplays.

Irregular datasets (XMark and Uniprot): Figure 34 to Figure 39 show the error rates for the Sampling, TreeSketch, and the Hybrid approaches on the Uniprot and XMark datasets while Figure 40 shows the overall error rates for the three approaches on both datasets. In this set of experiments we used different *SSR* values to test the scalability of the three approaches. For the XMark dataset, the Hybrid used an *SSR* value of 1.7 to capture almost the complete structure of the data tree so we set it as the maximum *SSR* value for the other two algorithms. We then reduced the *SSR* value to 0.08 which is equivalent to 100k and then reduced it further to 0.02 which is equivalent to 20k. At the max *SSR* the Hybrid approach outperformed both the Sampling and the TreeSketch approaches with an overall error rate of 0.08%. As the storage was reduced the accuracy of the three approaches decreased as well. At the lowest *SSR* the sampling algorithm had the worst accuracy with an overall error rate of 10.4 % while the Hybrid and the TreeSketch error rates remained below 10% with the TreeSketch having the lowest overall error rate of 7.6%. Although, the Treesketch showed a slightly better accuracy, the Hybrid at the low *SSR* values, the summary generation time for the TreeSketch at these values was too high. In fact the Hybrid approach at the *SSR* value of 0.02 managed to generate the summary around 54 times faster than the TreeSketch. Also, note that the biggest deterioration of accuracy happened when we decreased the *SSR* from 1.7 to 0.08 and as we decreased the storage further both the TreeSketch and the Hybrid had a very slight deterioration in the error rate while the error rate for the sampling almost doubled. This is because as the storage decreases the sampling approach cannot keep a good amount of samples to represent the structure of the data tree while the TreeSketch and the Hybrid approaches try to store enough structural or statistical data to compensate for the data lost during the

summary generation at different sizes. Also, taking a closer look at the results for the Hybrid error rates we noticed that for some queries the error rate slightly improved as we reduced the *SSR* value from 0.08 to 0.02. For example, in Q1 and Q2 in Table 8 the statistical component of the Hybrid system had a high estimate for the query root elements “description” and “text” and therefore the query delta was large. At the same time the structural component of the hybrid system had a good amount of structural data about the elements in Q1 and Q2 and produced a high selectivity count estimate resulting in a slight overestimation for Q1 and Q2 selectivity counts by the combined Hybrid estimator as shown in the table. On the other hand, as we reduced the storage the structural estimator lost more structural data for the elements in Q1 and Q2 but the statistical data remained the same and hence the selectivity count estimates decreased and so did the error rate. This was apparent in the case of P-C linear queries where the Hybrid achieved an error rate of 3.2% at *SSR* 0.08 and a slightly lower error rate of 2.8% at *SSR* 0.01 as shown in Figures 35(a) and 36(a).

For the Uniprot dataset the TreeSketch managed to capture the complete distribution of elements at an *SSR* value of 0.5 and therefore achieved an overall error rate of 0% at this *SSR* value. The Hybrid approach on the other hand needed an *SSR* value of 0.7 to capture almost the complete distribution of elements and achieved an overall error rate of 0.1% at this *SSR* value. This is because in the Uniprot elements with identical types and sub-tree-structures tend to have identical sub-tree counts as well, and since the TreeSketch uses a graph to store the summary, which is normally more succinct than a tree structure, it requires less number of nodes to store the summary and consequently less storage for this dataset. In our experiments we set the maximum *SSR* to 0.7. The sampling overall error rate at the maximum *SSR* was 3.6% which is more than three times higher than that of the

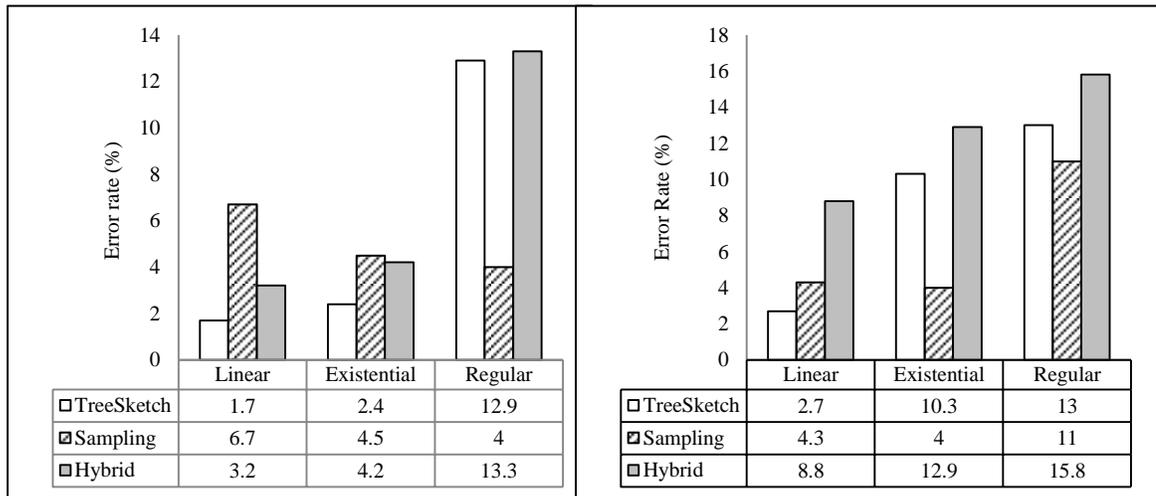
Hybrid. Similar to the XMark results, at the lowest *SSR* value the sampling had the highest overall error rate of 22.4%, Figure 40, while both TreeSketch and Hybrid had an error rate below 15% with the TreeSketch achieving the lowest error rate with 12.5%. Like in the XMark experiments, even though the TreeSketch had a slightly lower error rate than the Hybrid at the lowest *SSR* value, this came at the expense of the summary generation time where the Hybrid managed to generate the summary 22 times faster than the TreeSketch at this *SSR* value. Also, as we decreased the *SSR* value from 0.07 to 0.01 both the TreeSketch and the Hybrid showed a slight deterioration on the error rate while the error rate for the sampling more than doubled. Taking a closer look at the results we noticed that, unlike the XMark experiments on the twig queries, the TreeSketch at the lower *SSR* values showed an error rate for the existential twig queries that was higher than that of the regular twig queries. For example, Table 9 shows two twig sample queries with their existential and regular estimates and error rates. The TreeSketch at *SSR* 0.02 had a better distribution data for the elements in the non-critical paths in these queries, namely “//reference/scope” in Q1 and “//citation//dbReference” in Q2, and since the selectivity count estimates for existential twigs, unlike regular twigs, depend mainly on the counts of the elements in the critical path the regular estimates for those queries were more accurate than the existential estimates. Another thing to be noted in the Uniprot results is that the three approaches had lower error rates on the A-D linear queries than the P-C linear queries at the different *SSR* values, Figure 37 to Figure 39. This is because it took the TreeSketch approach a long time to estimate the selectivity counts for deep (i.e. *height* > 3) A-D queries at high *SSR* values and therefore we had to limit the depth for A-D queries to 3 resulting in simpler queries than the ones generated for the P-C experiments.



a) P-C queries

b) A-D queries

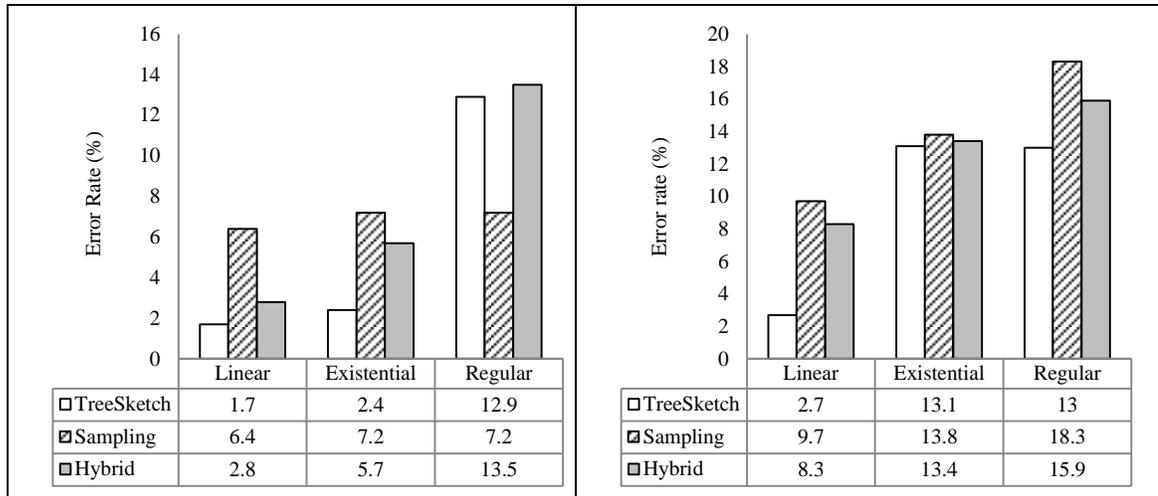
Figure 34: P-C & A-D queries error rates for Sampling, TreeSketch and Hybrid (query-delta) on XMark at SSR 1.7



a) P-C queries

b) A-D queries

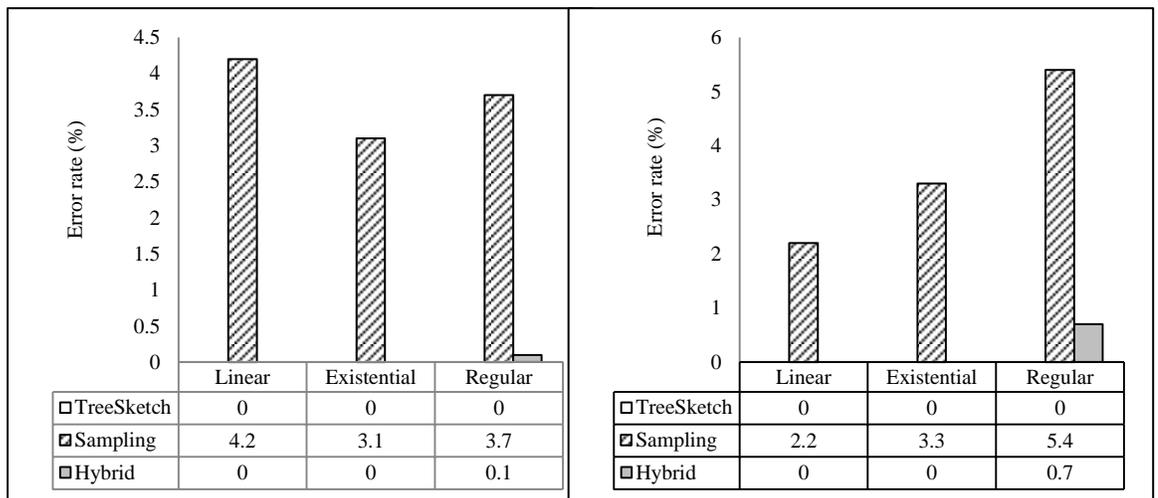
Figure 35: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on XMark at SSR 0.08



a) P-C queries

b) A-D queries

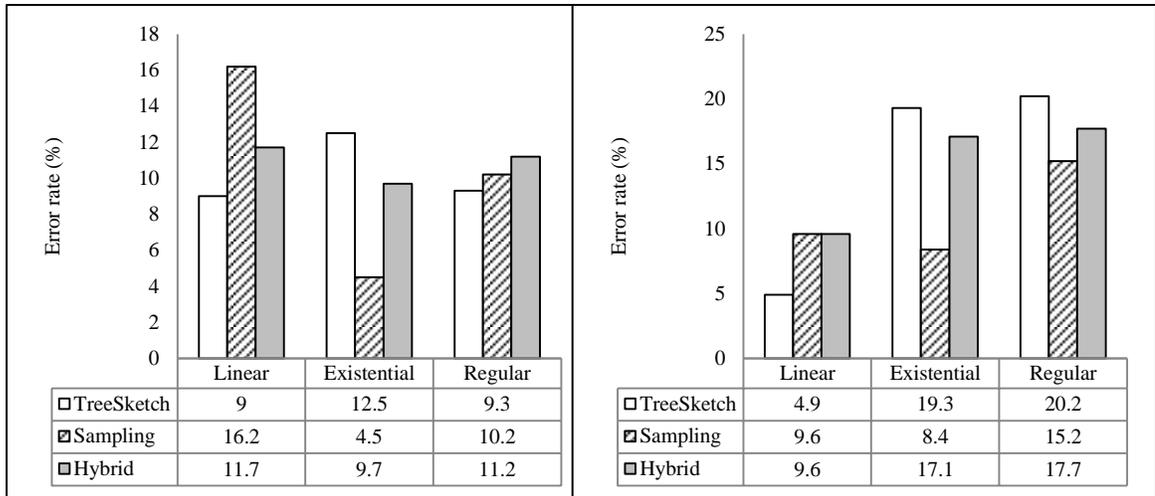
Figure 36: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on XMark at SSR 0.02



a) P-C queries

b) A-D queries

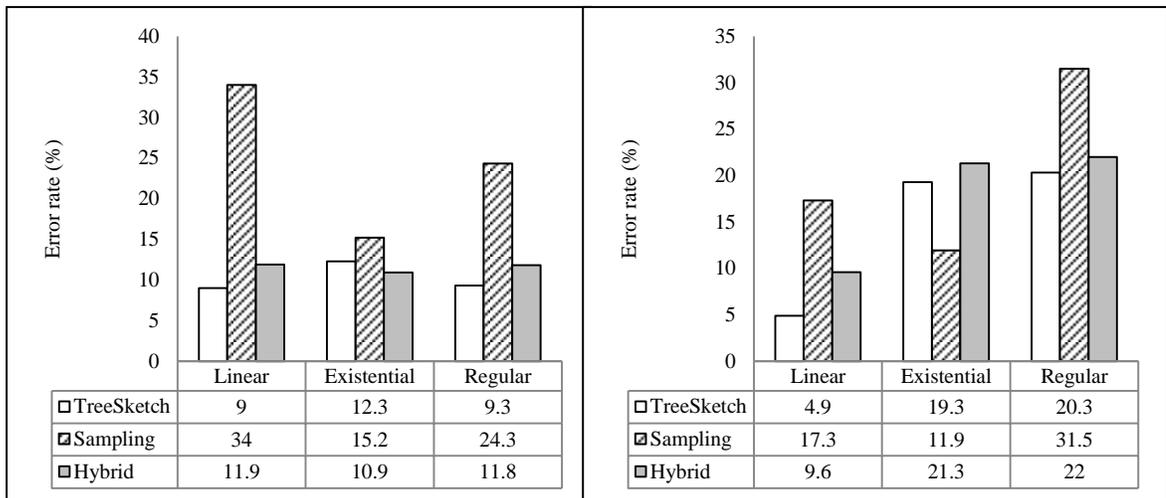
Figure 37: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at SSR 0.7



a) P-C queries

b) A-D queries

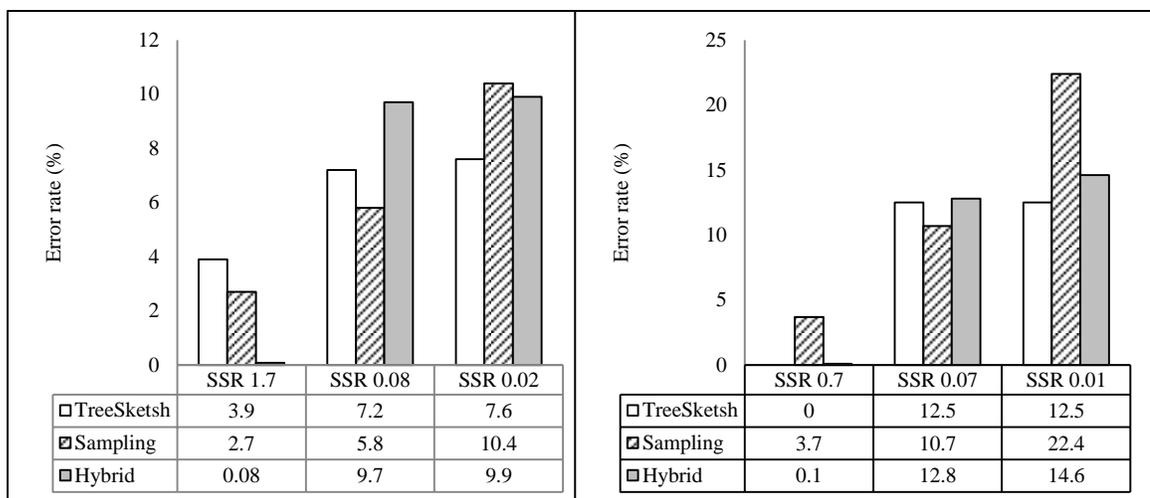
Figure 38: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at SSR 0.07



a) P-C queries

b) A-D queries

Figure 39: P-C & A-D error rates for Sampling, TreeSketch and Hybrid (query-delta) on Uniprot at SSR 0.01



a) XMark

b) Uniprot

Figure 40: Overall error rates for all queries on XMark and Uniprot

Table 8: Sample queries and the Hybrid results on the XMark

Query	actual	SSR 0.08 estimate	error rate (%)	SSR 0.02 estimate	error rate (%)
Q1://description/text//keyword//emph	1150	1230.8	7	1214.3	5.6
Q2://text//keyword/emph	3794	3804.6	0.3	3795.5	0.04

Table 9: Sample twig queries and TreeSketch results on the Uniprot at SSR 0.01

Query	actual (reg)	actual (exist)	estimate (reg)	error reg (%)	estimate (exist)	error (exist) (%)
//reference[/scope]/source/strain	16259	14832	10362	36.3	8308.6	44
//citation[//dbReference]//title	45362	19389	42204.2	7	23241	19.9

In summary, the experimental results showed that the Hybrid approach outperformed the other two approaches on the uniform datasets where it required very small *SSR* values to achieve an overall error rate that is close to 0% while in some cases the TreeSketch overall error rate was more than 11% and the Sampling 85%. For irregular datasets, although the sampling approach had a lower error rate at specific *SSR* values than the Hybrid and TreeSketch, both the Hybrid and TreeSketch showed better scalability than the sampling approach and achieved a higher accuracy at the lowest *SSR* values. Moreover, the TreeSketch accuracy was slightly higher than the Hybrid approach at the low *SSR* values but the summary generation time was significantly higher as we showed earlier. That being said, we plan to continue the research to find more accurate ways to capture the irregularities in XML data trees with minimal storage requirements. One possibility we plan to explore is the use of histograms to capture the irregular distributions of some or all elements in the data tree and complement the statistical and structural data of the Hybrid system with such histograms. In the following chapter we conclude this thesis and shed some light on the possible future extensions of this work.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this work, we proposed two summarization techniques for XML trees. The first is based on a bottom-up prime-number labeling scheme while the other is based on a fingerprint hash function. Based on the resulting summary tree, we developed a selectivity count estimation algorithm that can be used with different types of XML queries. We compared our approach with two other state-of-the-art approaches, namely, the Sampling and the TreeSketch. In all our experiments, the proposed approach outperformed the other two approaches in terms of estimation error rate on all datasets except with Uniprot in which the TreeSketch was marginally better only for regular twig queries. For instance, our technique had perfect estimation accuracy (0% error rate) for linear and existential twig queries on all datasets while the other two approaches showed higher error rates for these types of queries. Moreover, the worst error rate exhibited by our approach was only 0.8% on regular twig queries while the TreeSketch error rate reached 18.8% and the Sampling error rate reached 98% for some types of queries. Additionally, to extend our approach for environments with memory constraints, we proposed a hybrid approach that combines a statistical technique with our summarization technique. We showed that in our experiments our hybrid approach always outperformed the Sampling in terms of error rate when the storage budget is very small while the error rate for the TreeSketch was slightly lower on irregular datasets at the expense of significantly higher summary generation time. In fact, our approach was more than 54

times faster than the TreeSketch in generating the XMark summary tree and around 24 times faster in the case of the Uniprot. On the other hand, the improvement in the TreeSketch error rate over our approach on these datasets did not exceed 2%.

As future work, we intend to continue the research in this area in order to provide a selectivity count estimation framework for more XPath axes such as the following/preceding axis. We also intend to examine the possibility of adding value predicates to our selectivity count estimation framework. Moreover, we plan to explore the use of histograms to capture the irregularities of some or all elements in the source XML data tree with minimal storage requirement and complement the statistical and structural data of the Hybrid system with such histograms to improve the estimation accuracy when the storage budget is extremely limited.

References

- [1] T. Bray, J. Paoli, C. M. Sperberg-McQueen and E. Maler, Extensible Markup Language (XML) 1.0, Second Edition, 2000. [Online]. Available: <http://www.w3.org/TR/REC-xml>.
- [2] A. Fomichev, M. Grinev and S. Kuznetsov, "Sedna: A native XML DBMS," in *Proc. 32nd Conf. Current Trends in Theory and Practice of Computer Science*, 2006.
- [3] H. Schoning, "Tamino –A DBMS designed for XML," in *Proc. Int Conf. Data Engineering (ICDE)*, 2001.
- [4] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Papparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu and C. Yu, "Timber: a native XML database," in *Proc. Very Large Databases (VLDB)*, vol. 11, no. 4, pp. 274–291, 2002.
- [5] Y. H. Chu and J. L. Yu, "The research of database query optimization based on XML," *Advanced Materials Research*, vol. 546, pp. 519-525, 2012.
- [6] T. L. J. Lu and C. W. Z. Bao, "Extended XML tree pattern matching: theories and algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, pp. 402-416, 2011.
- [7] J. Kim and S. Park, "XQuery speedup by deploying structural redundancy in mapping XML into relations," *Information and Software Technology*, vol. 48, no. 1, pp. 12-30, 2006.
- [8] S.-C. Haw and C.-S. Lee, "Data storage practices and query processing in XML databases: a survey," *Knowledge-Based Systems*, vol. 24, no. 8, pp. 1317-1340, 2011.
- [9] C. Luo, Z. Jiang, W.-C. Hou, F. Yu and Q. Zhu, "A sampling approach for XML query selectivity estimation," in *Proc. 12th. Int Conf. Extending Database Technology*, Saint Petersburg, Russia, 2009.
- [10] S. Sakr, "Towards a comprehensive assessment for selectivity estimation approaches of XML queries," *International Journal of Web Engineering and Technology*, vol. 6,

pp. 58–82, 2010.

- [11] J. F. Naughton, D. J. DeWitt, D. Maier, et al. "The Niagara Internet query system," *IEEE Data Engineering Bulletin*, vol. 24, no. 2, pp. 27-33, 2001.
- [12] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. Sperberg-McQueen, L. Wood and J. Clark, "W3C XML Specification DTD," 1998. [Online]. Available: <http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm>.
- [13] D. C. Fallside, "XML Schema part 0: Primer," 2001. [Online]. Available: <http://www.w3.org/TR/xmlschema-0/>.
- [14] C. Zhang, J. Naughton, D. DeWitt, Q. Luo and G. Lohman., "On supporting containment queries in relational database management systems," in *Proc. Special Interest Group on Management of Data, (SIGMOD)*, 2001.
- [15] X. Wu, M. L. Lee and W. Hsu, "A prime number labeling scheme for dynamic ordered XML trees," in *Proc. 20th Int. Conf. Data Engineering (ICDE)*, 2004.
- [16] J. CLARK and S. DEROSE, "XML Path Language (XPath) 1.0 W3C," 1999. [Online]. Available: <http://www.w3.org/TR/xpath>.
- [17] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon and M. Stefanescu, "XQuery 1.0: An XML query," 2002. [Online]. Available: <http://www.w3.org/TR/xquery>.
- [18] S. Abiteboul, D. Quass, J. Mchugh, J. Widom and J. Wiener, "The Lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, no. 1, pp. 68-88, 1997.
- [19] D. D. Chamberlin, J. Robie and D. Florescu, "Quilt: An XML query language for heterogeneous data sources," in *Proc. Web and Databases Workshop (WebDB)*, 2000.
- [20] Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng and D. Srivastava, "Counting twig matches in a tree," in *Proc. 17th Int. Conf. Data Engineering (ICDE)*, 2001.
- [21] A. Aboulnaga, A. Alameldeen and J. Naughton, "Estimating the selectivity of XML path expressions for internet scale applications," in *Proc. Int. Conf. Very Large Databases (VLDB)*, Rome, Italy, 2001.

- [22] M. Alrammal, G. Hains and M. Zergaoui., "Path tree: document synopsis for XPath query selectivity estimation," in *Proc. 5th Int. Conf. Complex, Intelligent, and Software Intensive Systems (CISIS-2011)*, IEEE Computer Society, Seoul, Korea, 2011.
- [23] N. Zhang, M. T. Ozsü, A. Abounaga and I. F. Ilyas, "XSeed: accurate and fast cardinality estimation for XPath queries," in *Proc. 20th Int. Conf. Data Engineering (ICDE)*, 2006.
- [24] N. Polyzotis and M. Garofalakis, "XSketch synopses for XML data graphs," *Transactions on Database Systems (TODS)*, vol. 31, no. 3, p. 1014–1063, 2006.
- [25] N. Polyzotis, M. Garofalakis and Y. Ioannidis, "Selectivity estimation for XML twigs", in *Proc. 20th Int. Conf. Data Engineering (ICDE)*, 2004.
- [26] N. Polyzotis and M. Garofalakis., "Statistical synopses for graph-structured XML databases," in *Proc. Special Interest Group on Management of Data (SIGMOD)*, 2002.
- [27] Y. Wang, H. Wang, X. Meng and S. Wang, "Estimating the selectivity of XML path expression with predicates," *Advances in Web-Age Information*, vol. 3129 of Lecture Notes in Computer Science, pp. 409-418, 2004.
- [28] Y.Wu, J. Patel and H. Jagadish, "Estimating answer sizes for XML queries," in *Proc. Extending Database Technology (EDBT)*, Prague, Czech Republic, 2002.
- [29] W.Wang, H. Jiang, H. Lu and J. X. Yu, "Bloom histogram: path selectivity estimation for XML data with updates," in *Proc. Verey Large Database (VLDB)* , 2004.
- [30] M. Lee, H. Li, W. Hsu and B. Ooi, "A statistical approach for XML query size estimation," in *Proc. DataX Workshop*, 2004.
- [31] H. Li, M. L. Lee and W. Hsu, "A histogram-based selectivity estimator for skewed XML data," in *Proc. 16th Int. Conf. Database and Expert Systems Applications*, Copenhagen, Denmark, 2005.
- [32] N. Polyzotis and M. Garofalakis., "XCluster synopses for structured XML content," in *Proc. Int. Conf. Data Engineering (ICDE)*, 2006.
- [33] L. Lim, M. Wang and J. S. Vitter, "CXHist: An on-line classification-based histogram for XML string selectivity estimation," in *Proc. 31st Int. Conf. Very Large*

Data Bases (VLDB), New York, 2005.

- [34] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter and R. Parr, "XPathLearner: an on-Line self-tuning markov histogram for XML path selectivity estimation," in *Proc. Very Large Databases (VLDB)*, 2002.
- [35] S. Comai, S. Marrara and L. Tanca, "A synopsis based approach for XML fast approximate querying," in *Proc. Flexible Databases Supporting Imprecision Uncertainty*, 2006, pp. 241-265.
- [36] A. M. Weiner, "Advanced cardinality estimation in the XML query graph model," in *Proc. 14th Conf. Database Systems for Business, Technology and Web*, Kaiserslautern, Germany, 2011.
- [37] N. Drukh, N. Polyzotis, M. Garofalakis and Y. Matias, "Fractional XSketch synopses for XML databases," in *Proc. Database and XML Technologies*, Springer, 2004, pp. 189-203.
- [38] N. Polyzotis, M. N. Garofalakis and Y. Ioannidis., "Approximate XML query answers," in *Proc. Special Interest Group on Management of Data (SIGMOD)*, 2004.
- [39] M. Alrammal, G. Hains and M. Zergaoui, "Path tree: document synopsis for XPath query selectivity estimation," in *Proc. 5th Int. Conf. Complex, Intelligent, and Software Intensive Systems (CISIS), IEEE Computer Society.*, Seoul, Korea, 2011.
- [40] H. Li, M. L. Lee, W. Hsu and G. Cong, "An estimation system for XPath expressions," in *Proc. 22nd Int. Conf. Data Engineering (ICDE)*, Atlanta, Georgia, USA, 2006.
- [41] N. Zhang, M. Ozsu, A. Aboulnaga and I. Ilyas, "XSeed: accurate and fast cardinality estimation for XPath queries," in *Proc. 20th Int. Conf. Data Engineering*, 2006.
- [42] C. Wang, S. Parthasarathy and R. Jin, "A decomposition-based probabilistic framework for estimating the selectivity of XML twig queries," in *Proc. Extending Database Technology (EDBT)*, 2006.
- [43] M. O. Rabin, "Probabilistic algorithm for testing primality," *Number Theory*, vol. 12, p. 128–138, 1980.
- [44] C. Pomerance, J. L. Selfridge and S. S. Wagstaff Jr., "The pseudoprimes to $25 * 10^9$," *Mathematics of Computation*, vol. 35, no. 151, p. 1003–1026, 1980.

- [45] G. Jaeschke, "On strong pseudoprimes to several bases," *Mathematics of Computation*, vol. 61, pp. 915-926, 1993.
- [46] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, 1987.
- [47] E. Jiao, T. W. Ling and C.-Y. Chan, "A holistic path join algorithm for path query with not-predicates on XML data," in *Proc. Database Systems for Advanced Applications*, Springer, 2005, pp. 113-124.
- [48] N. Bruno, N. Koudas and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," in *Proc. Special Interest Group on Management of Data (SIGMOD)*, 2002.
- [49] J. D. M. Hachicha, "A survey of XML tree patterns," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, pp. 29-46, 2013.
- [50] "DBLP," [Online]. Available: <http://dblp.uni-trier.de/xml/>. [Accessed April 2013].
- [51] "Xmark," [Online]. Available: <http://www.xml-benchmark.org/>. [Accessed April 2013].
- [52] "Shakespeare Plays," [Online]. Available: <http://www.ibiblio.org/xml/examples/shakespeare/>. [Accessed April 2013].
- [53] I. Mlynkova, K. Toman and J. Pokorny, "Statistical analysis of real XML data collections," in *Proc. Special Interest Group on Management of Data (SIGMOD)*, 2006.
- [54] L. H. Yang, M. L. Lee, W. Hsu, D. Huang and L. Wong, "Efficient mining of frequent XML query patterns with repeating-siblings," *Information and Software Technology*, vol. 50, no. 5, pp. 375-389, 2008.
- [55] T. Amagasa, M. Yoshikawa and S. Uemura, "QRS: A robust numbering scheme for XML documents," in *Proc. 19th Int. Conf. Data Engineering (ICDE)*, IEEE, 2003, pp. 705-707.
- [56] "Uniprot," [Online]. Available: <http://www.uniprot.org/>. [Accessed November 2013].

- [57] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava, "Counting Twig Matches in a Tree," In *Proc. 17th Data Engineering (ICDE)*, IEEE, 2001, pp. 595-604.
- [58] J. Freire, J. R. Haritsa, J. R., M. Ramanath, P. Roy, and J. Siméon, "StatiX: Making XML Count," in *Proc. Special Interest Group on Management of Data (SIGMOD)*, ACM, 2002, pp. 181-191.
- [59] P. Bohannon, J. Freire, J. R. Haritsa, P. Roy, and J. Siméon, "LegoDB: Customizing relational storage for XML documents," in *Proc. Very Large Databases (VLDB)*, 2002, pp. 1091–1094, 2002.

CURRICULUM VITAE

Ahmad Faisal Hashim Barradah
Exploration Network Operations Department
Saudi ARAMCO
ARAMCO Box # 10073
Dhahran 31311, Saudi Arabia
E-mail: barradaf@aramco.com

CURRENT POSITION:

Computer Operating Systems Specialist.
PHONE : 873-1181
ENGINEERING BUILDING.

EDUCATION:

B.S. in Software Engineering , KFUPM 2004

Experience:

2006-2007 Redhat Linux administrator
2007-2008 Data Storage Specialist
2008-2010 High performance computing specialist
2010 operations technical support group lead at Saudi Aramco ECC

Projects:

- Implementing a service and system monitoring solution covering all computing technologies in Saudi Aramco ECC datacenter.
- Redhat upgrade project and migration of company's applications to the new operating environment.

- Commissioning 310 TB of high performance storage (Netapp GX).
- Developing tools to facilitate the migration of more than 500 TB of data through SAN.
- Morphological Analyzer for Arabic
- A statistical spell checker for Arabic

Publications:

Al-Jamimi, Hamdi A., Ahmed Barradah, and Salahadin Mohammed. "Siblings Labeling Scheme for Updating XML Trees Dynamically.", *4th international conference on Computer Engineering and Technology (ICCET)*, 2012

Certifications:

- ITIL V3 Foundation (2010)
- Red-Hat Certified Engineer (2008)
- Netapp Certified Data Management Administrator (2008)
- Win 2003 server Microsoft Certified Professional (2007)
- Win 2003 Network Infrastructure Microsoft Certified Professional (2007)
- Win Microsoft Certified XP Professional (2006)
- Sun Solaris 9 system Administrator (2006)
- Helpdesk Analyst (2006)