UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE MOBUS

An Analysis of Estimation Methods for XML Document Selectivities

Projeto de Diplomação

Prof. Dr. Carlos A. Heuser
Advisor

Prof. Dr.-Ing. Dr. h. c. Theo Härder
Coadvisor

Porto Alegre, June 2008

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Pró-Reitor de Coordenação Acadêmica: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Prof^{a}. Valquíria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
DOM	Document Object Model
EB	End-Biased (histogram)
EH	Equi-Height (histogram)
LH	Level Histograms
MT	Markov Tables
NMRSE	Normalised Root-Mean-Square Error
PS	Path Synopsis
PSUPS	PS with Uniform Parent Selectivity
QEP	Query Execution Plan
RE	Relative Error
SAX	Simple API for XML
SPLID	Stable Path Labelling Identifier
VocID	Vocabulary Identifier
XDBMS	XML Database System
XML	eXtensible Markup Language
XPATH	XML Path Language
XTC	XML Transaction Coordinator

ABSTRACT

XML recently became a fledging field of study in Computer Science. Designed initially as standard way to represent and serialise data structures used across different platforms and systems, specially useful for the Internet environment, it has recently prompted the search for reasonable ways to store and query XML documents. It is already possible to store hierarchycal or otherwise structured information using the relational database model, but available approaches for this are not elegant enough and fundamentally break the relational abstraction.

Thus, there is strong research on developing a different kind of database, one that allows for data and structure to be intertwined in a more transparent and native form, breaking away from the relational model, whose structure was rather restricted to two-dimensional tables. Given that XML is the current dominant format for structured data representation, it is only logical that XML is the main theme when speaking of structured data storage, giving birth to the Native XML Database Systems (XDBMS) and research associated with it.

Such departure from the previous models, of course, brings a host of issues that must be addressed before XDBMSs can be considered ready for the "real world". Most aspects of XDBMSs are still incipient, with many competing ideas being evaluated, but few or none becoming accepted as solutions. Therefore, XDBMSs are a rich field of study, with many challenges yet to be solved.

One of such challenges is the execution of complex queries over potentially large sets of data. As with relational world, large sets of data usualy imply a great amount of I/O operations, with penalties to performance. The usual solution in the relational world is to keep some information on the distribution of data values and use it to rule out most of the unwanted data as early as possible. Keeping distribution data for a given column in a relational table is cheap, for usually such data is highly homogeneous; usually a simple structure called histogram will suffice. In a XML document, however, structure must also be taken into account, and data is likely to be heterogeneous.

The scope of this report is creating a functionally analogous structure to the relational histogram that will be responsible for the summarisation of the structure of an XML document, giving the Query Optimiser information it needs regarding distribution of structures in the XML document being queried. Two summarising structures were created in this work, after studying the merits and issues of the existing approaches; histogram concepts and techniques already in use in the relational world were brought as tools for compressed expression of data distributions. Those new structures give good results in terms of space even when being applied for large and complex documents.

Keywords: XML, XDBMS, summary structures, histograms, XPATH, query optimization, path synopsis, databases.

1 INTRODUCTION

When designing an XML Database System (XDMBS), several problems must be studied and solved in order for such system to perform with reasonable scalability. Scalability must be considered in several aspects of the system: concurrence management, storage requirements, query planning and optimisation, etc. The scalability aspect with which we are concerned is cost-based query optimisation, whose algorithms usually require data structures to hold analytical data of an XML document structure. For the purpose of scalability, it is important that such structures remain reasonable in terms of size even when dealing with deeply nested or otherwise complex XML documents.

1.1 Problem Summary

A cost-based XML query optimiser is the part of a XDBMS responsible for the best-effort selection of the most reasonable QEP (query execution plan), i.e., the plan that will potentially execute faster and require less resources when compared to other available plans. Analogous to what happens in the relational world, disk access is usually the major performance roadblock; hence, minimising the number of I/O operations needed for a given query is of great importance.

A query over an XML document usually consists of several low level operations, such as selections, structural joins, etc, that are usually executed as an tree of nested operations, where the result of the operations on the bottom is fed to other operations. These operations may have associative and commutative properties, i.e., the order of their operand may change without modifying the final result.

Depending on the architecture, an operation may have several possible physical implementations, which in turn may have varying I/O requirements according to the order and size of their operands, i.e., for an physical operation p_{op}, whose operands are A and B, $A\ p_{op}\ B$ may be faster or slower than $B\ p_{op}\ A$. Another possibility is trying to cut out most of the operand's size earlier in the operation tree, thus resulting in smaller intermediate operands. Additionally, some operations may be given priority when a structure such as an index is available.

Hence, there are four paths for cost reduction: choosing the best physical implementation for each given operation, arranging operators in a way that minimises I/O for a given physical implementation, trying to rule out most unwanted data as earlier as possible in the operation tree, and giving priority to operations that may have their performance boosted by index usage. All those optimisation paths are diverse in terms of implementation and complexity, but their decision-making points always require some foreknowledge of operands' sizes.

Therefore, to achieve cost reduction, it is important to know beforehand (or be able to guess) the size of the operands involved. In structural queries, this means that the distribution of hierarchical relations between different elements must be known. Looping the whole document and counting its existing structures for every single query is clearly not adequate, mainly in large XML documents. Therefore, it is important to have auxiliary structures holding information regarding the structural distribution of the document.

In more empirical terms, consider the case of a XPath (XPATH XML PATH LANGUAGE 2.0., 2005) query expression such as `/a/b/c`: we need to know how nodes with tag `c` relate to nodes with tag `b` which, in turn, relate to nodes with tag `a`, all according to the `child` axis. In simpler words, we need to estimate how many `c` are children of `b` whose parent is `a`.

1.2 Goal of This Report

The use of summarised information (generally called statistics) on stored data to drive optimiser decisions through cost-based query execution plans is very important for the selection of an "optimal" plan. In the relational world, the task of counting elements of relations involved in join operations is usually done by keeping structures such as histograms, which stores the distribution of values for a given attribute as compressed as possible without losing much of precision. This technique has been long established and improved (PIATETSKY-SHAPIRO; CONNELL, 1984), giving good results even for large sets of values with minimal error.

In the XML database world, however, besides attribute/element values, one must also consider structural characteristics of a document: how many elements `x` relate with how many elements `y` in a given axis. This adds complexity to this problem, and simply storing the estimated size of a each element/attribute of a whole XML document taking apart its structural relationships is not enough anymore. Moreover, beside the structural part of XML, there is the content part, i.e., text values which can exist as leaf nodes in an XML document. Summarising, both, structure and values is another challenging issue. As of today, the XML summarization problem has no definitive answer, and several approaches have been studied in the literature.

The goal of this report is to identify potential problems in the current proposals and improving some areas not previously thought before. We focus mostly on the space required for summary structures, because we understand that loading summary structures should require as few I/O as possible. The quality, i.e., the accuracy, of the estimated results is also important, which prompt us to avoid discarding structural information just for the sake of space: we try to coalesce information without making it too inaccurate or generic. An idea well-known in the relational world, the histogram, is heavily employed for that task, coupled with other small improvements on the organisation of the structure itself.

1.3 Overview of This Report

This report is the part of the project of the design of a XML Database System called XTC (HAUSTEIN; HARDER, 2007), developed and implemented by the AG DBIS (Databases and Information Systems Workgroup) at the TU-KL (Technical University of Kaiserslautern). As part of a cooperation project with UFRGS (Universidade Federal do Rio Grande do Sul) in the year of 2007. During a one-year research internship, data structures for summarisation of XML documents were studied, implemented, and evaluated inside the XTC framework abovementioned. Such study prompted novel approaches to overcome limitations of the state of the art that are presented in this report.

In Chapter 2, the state of the art, the basic concept of Path Synopses (PS) is defined. Its extension possibilities, uses, and implementation approaches are studied. Two state of the art implementations of different approaches are listed and discussed.

Chapter 3 discusses a tool already used in the relation world with great success, the histogram. It also lays out some definitions and classifications for it based on the current literature.

In Chapter 4, novel approaches for Path Synopsis structures are introduced, using histograms and other optimisations to achieve reduction of space requirements. Two techniques are presented, differing in the way they select data to be coalesced in to the histogram.

Chapter 5 qualitatively discusses on how all those implementations may perform for different types of XML documents, and Chapter 6 briefly shows how each of the above mentioned implementations are actually stored in the XTC XDBMS (HAUSTEIN; HARDER, 2007), allowing for size × accuracy benchmarks that follow in chapter 9.

2 STATE OF THE ART

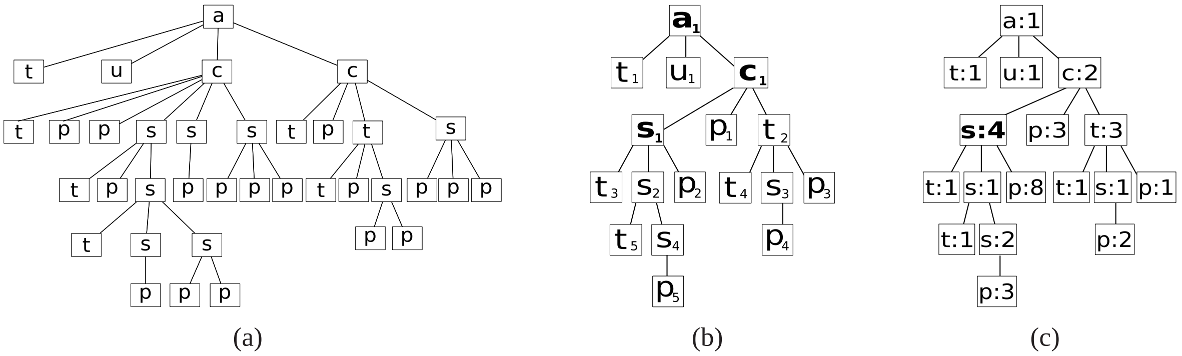
2.1 PS - Path Synopsis

2.1.1 Definitions

We study a class of structures called Path Synopses (PS) for XML data representation. Their purpose is to capture as much structural aspects of an XML tree as possible. In order to do that, we first define the concept of path class, as a set of all paths in an XML tree, from root to an inner node or a leaf node, that share the same sequence of element/attribute names. A document can (and normally will) contain a number of such path classes. PS structures, instead of storing all individual path instances of a path class, only capture the distinct path classes of an XML tree.

The resulting auxiliary data structure can be used to detect path patterns, to recommend index creation, and to keep statistical data. The latter use is our focus. In our work, we extend this structure to store the cardinalities of those paths, which will be used to compute estimations with respect to different node distributions throughout different regions of an XML document.

As an example, consider the XML tree represented in Figure 2.1(a). This document has several root-based path classes, and each of them may have several instances. In a PS structure, all those instances are collapsed into the same entry.



(a) (b) (c)

Figure 2.1: An example XML tree (a), its PS tree (b), and its extended PS tree (c)

For purposes of visualisation, in Figure 2.1(b), we organized all existing root-based path classes in the example XML tree as a prefix tree. Therefore, each node in this tree represents a root-based path class: the label of a given node is the name of the last element of the path class it represents, and its ancestors represent the preceding element names. For example, consider the path class `/a/c/s`. It is represented in this structure by node s_1, which in turn has nodes c_1 and a_1 as its ancestors (preceding element names in the path class).

2.1.2 Extending PS

For the purpose of query optimisation, it is important to keep statistical data regarding the structure of an XML tree. As such, one may extend a PS structure to store diverse information about the distribution of path classes occurring in the XML tree. A number of aspects could be stored for each path class regarding its presence in the original tree, such as frequency, fan-out related to its children, selectivity against its parents, etc., by simply adding data to that path class' corresponding PS node.

Consider for example, the extension of a PS structure to store the frequency of a path class, i.e., the number of instances belonging to that path class in the original XML tree. Such an extension is shown in Figure 2.1(c): each PS node label now carries frequency information for the associated path class. For example, consider the path class `/a/c/s`: in the original PS tree, it was represented by the node s_1; in the extended structure, it is represented by the node with label s:4, meaning that there are four instances of this path class in the XML tree.

2.1.3 Estimating cardinality and selectivity of query expressions

Extending the PS structure with the frequency of its path classes provides important information for a query optimiser to make decisions on QEPs. Given a multiple-step query expression, normally an XPath (XPATH XML PATH LANGUAGE 2.0., 2005) or XQuery (XQUERY 1.0: AN XML QUERY LANGUAGE, 2005) expression, the order in which we evaluate the steps of the expression may affect the performance of query execution. The performance measurement of a given query execution order is called cost and may be measured in terms of operations executed (complexity in terms of time), memory required (complexity in terms of space), I/O operations required (complexity in terms of interruptions and disk accesses). To be able to decide, for a given set of QEPs, which of them has the minimal cost, without executing all of them, one needs a cost model, i.e., a function that returns the expected cost for all operations required by a QEP. Defining a cost model is usually architecture-dependent and, therefore, beyond the scope of this work.

However, most cost models will, among other information, need to know how big each intermediate result of a QEP may become in order to evaluate its influence on the total cost. Considering that logical operations may be arbitrarily combined to form intermediate results, it is impractical to store and maintain statistics for all possible results. Thus, cardinality of intermediate results must be estimated from existing information, in our case, frequency of path classes. Estimation techniques usually consist of applying formulas and algorithms over path class cardinality information to estimate the selectivity of each step of a query expression.

Selectivity is a concept well-known from RDBMSs and it describes the amount of elements in a table that will return true for a given predicate, where a predicate may be a value selection or a relational join criterion. In general terms, it is a good idea to apply the most selective predicates first, for they will restrict a table faster than the less selective ones. For example, consider a table A with 1000 elements, and predicates b and c, which evaluate to true in 100 and 5 elements, respectively. If we want to know which elements of A obey both b and c, filtering first by b will leave us with 100 elements to check for predicate c; conversely, applying c first will leave us only 5 elements to check for b. This concept can be extended to XML structural queries by regarding them as multiple joins between tables, also known as a *structural join*.

We extend the concept of selectivity for XML query mechanisms such as Xpath (XPATH XML PATH LANGUAGE 2.0., 2005) by considering their intermediate results for a given query and how the sizes of those results relate. However, multiple definitions of selectivity are possible in this context. For example, (i) the ratio between the last element of the query in the desired context against all occurrences of that element in any context or (ii) the ratio of that element in the desired context against the cardinality of the elements that form the context. To illustrate that, consider an XPath query `/a/c/s/p`: The former definition would yield p_c/p_d, where p_c is the cardinality of `p` in the desired context and p_d is the sum of all cardinalities of `p` in the whole PS structure. The latter definition would result in $p_c/(a_c+c_c+s_c)$, where a_c, c_c, and s_c represent the cardinalities of the elements forming the desired context.

2.1.4 Implementation approaches

It is noticeable that we can store path class information using less nodes or entries than the XML document itself. However, a PS structure, to be efficient, must require storage space of orders of magnitude smaller than that of the XML document. Ideally, it should fit in a few memory pages, thus allowing for memory-resident use. Should the resulting structure be much larger than that it could impose a burden on the performance of the QEP optimisation process. To accomplish minimal space requirements, PS implementations use a number of approaches, which greatly vary with respect to base structure, completeness of path classes represented, accuracy, and building time. As accuracy and building time are mostly dependent on base structure and completeness, we used the two latter as classification criteria of PS.

2.1.4.1 Base structure criterion

Taking a criterion of the base structure into account, we can roughly classify them as tree-based, table-based or graph-based approaches.

Tree-based approaches resemble a shrunk tree representation of an XML document in which the hierarchical structure is completely captured and the number of occurrences of each node is summarised in one element with its cardinality. This approach has the advantage of describing all aspects of the document. However, for recursive documents, it may require a large memory budget. Examples are PSUPS (see Section 4.1) and LH (see Section 4.2).

Table-based approaches, such as Markov Tables (ABOULNAGA; ALAMELDEEN; NAUGHTON, 2001) and Bloom Histograms (WANG et al., 2004) try to represent a document tree as a sequence of *path-value* pairs, in which *value* indicates the number of occurrences of a *path* in a document. Paths are not necessarily root-to-leaf paths. Normally, there is a pruning for storing path information. In the other words, table-based approaches store paths down to a certain length, where this pruning is regulated, in most of cases, by a tuning parameter. They are of simple implementation and serialisation. However, table-based approaches do not capture very well all axes necessary to support full query processing of the XQuery language. Moreover, it is difficult to estimate path predicates.

Graph-based approaches try to condense the document's structure into a graph, where nodes represent a set of elements/attributes occurring in a specific part of the XML document and edges describe `parent-child`relationships between them. These approaches can be easily fitted in small memory budgets, but at cost of accuracy. Examples of such approaches in the literature are XSeed (ZHANG et al., 2006) and XSketch (POLYZOTIS; GAROFALAKIS, 2006).

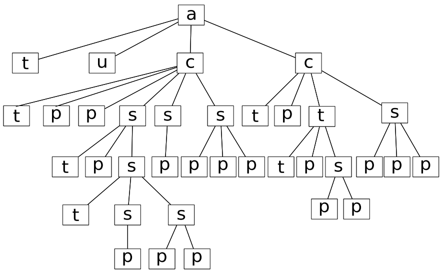
2.1.4.2 Completeness criterion

Regarding the number of path classes represented, path synopsis approaches can also be classified into two categories: path synopses may be complete or partial. **Complete PS** computes all distinct path classes in a document, whereas **Partial PS** derives cut-offs of the paths, ignoring path classes longer than what an arbitrary tuning parameter indicates.

Both criteria are orthogonal and we study five PS structures, namely Markov Tables (Section 2.2), PSUPS (Section 4.1), Level Histograms (LH) (Section 4.2), and XSeed (Section 2.3). We have implemented the three former approaches and used the latter for benchmark. Markov Tables are table-based partial synopses. XSeed falls in the category of graph-based complete synopses, and PSUPS and LH are tree-based complete synopses. We have also implemented two kinds of histograms for PSUPS and LH, namely End-biased and Equi-height histograms. Next, we discuss histograms.

2.2 Markov Tables

Markov Tables have been defined by (ABOULNAGA; ALAMELDEEN; NAUGHTON, 2001). This approach captures classes of partial paths, i.e., paths starting anywhere in the document with length $\leq m$, where $m \geq 2$. m is thus a tuning parameter. On the other words, it means that path classes with length greater than m are not represented in MT. Those partial paths are stored in a table along with their frequencies. This work also defines formulas based on a short-memory Markov model, to estimate the cardinality of longer paths based on the information stored on the mentioned table. These tables can also be fitted into a memory budget according to coalescing methods defined by work in (ABOULNAGA; ALAMELDEEN; NAUGHTON, 2001).



$length = 1$		$length = 2$					
path	card	path	card	path	card	path	card
A	1	A/T	1	A/U	2	A/C	1
T	6	T/T	1	T/P	1	T/S	1
C	2	C/T	3	C/P	3	C/S	4
S	8	S/T	2	S/P	13	S/S	3
U	1						

Table 2.1: An example XML tree and its corresponding Markov Table (m=2)

For the XML tree in Figure 2.1(b), Table 2.1 shows the resulting table with $m = 2$. This table, while small, does not immediately give us estimated sizes for paths classes of length greater than two. To estimate those cardinalities, we combine several paths of length $\leq m$ using the formula:

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

For example, given Table 2.1 and applying the above formula 2.1 for `//A/C/S/T` query would yield:

$$\begin{aligned} f(A/C/S/T) &= f(A/C) \times \frac{f(C/S)}{f(C)} \times \frac{f(S/T)}{f(S)} \Rightarrow \\ f(A/C/S/T) &= 2 \times \frac{4}{2} \times \frac{2}{8} = 1 \end{aligned}$$

Markov Tables give an accurate approximation of the structure of the XML data based on a short memory Markov Model assumption, but this structure may become too big to fit into available memory, depending on the document. It is possible to further summarise this table by deleting low frequency path classes and replacing the deleted path classes with *-path (”star paths”) that preserve some of the information lost by deletion. There are several methods for selecting the paths to be deleted. We will study two such methods, suffix-* and no-*. These algorithms will run for each path class length until a memory budget is reached.

2.2.1 Suffix-*

In this method, we define a special path class for each length up to *m*. This path class is called *star-path* and holds information regarding all deleted entries of that length. Each star-path is named with as many stars as the length of the path classes it represents, i.e., length 1 is `*`, length 2 is `*/*`, etc. We also define *suffix-** path classes which hold information of deleted entries with common suffixes.

Given a value for *m*, the algorithm proceeds as follows. For path classes of length 1, every deleted path class will be added into the special path class named “`*`”, which will, at the end, hold the average of the deleted entries. For path classes of length ≥ 2, a more elaborate process is required, as described below.

For lengths ≥ 2, we keep a set of deleted paths S_d and at each algorithm iteration, we select among entries of the current size, the one with smaller cardinality and delete it. Then, if the memory budget has not been reached yet, the following happens:

- If this path class shares the prefix of any other path class in S_d, both are removed from S_d and coealesced into special entry called “suffix-*”, whose path is only the common prefix of the original path classes. This new suffix-* entry is then added to the Markov Table. For instance, if the deleted entry is `A/C` and there is a `A/B` entry already in S_d, then it means both share a common prefix and will be coalesced into a suffix-* entry with path `A/*`.

- Otherwise, if there is a suffix-* path class with the same prefix as the path class being deleted, we add it to that suffix-*. For instance, if the deleted entry is `A/D` and there is an `A/*`, the latter will be update with information of the deleted entry.

- Otherwise, if the deleted path class is a suffix-* path, it is added to the *-path of that length.

- Otherwise, if all above steps fails, the path class is simply added to S_d.

It should be noted that, when a suffix-* path class is created, it is re-inserted into the Markov Table, sorted according to the *total* cardinality of the entries it represents, and maybe deleted in the future. At the end of the summarization, paths still remaining in S_d are added to the *-path of their length and average frequency of elements in suffix-* and *-paths is computed and stored.

This method is a *positive* method meaning that, when it does not find a path class in the table when applying the cardinality formula, it will assume that such path class exists either in a suffix-* path or a *-path and use one of them to proceed in the calculation. It will also prefer suffix-* paths over *-paths, because the latter has less information than the former.

2.2.2 No-*

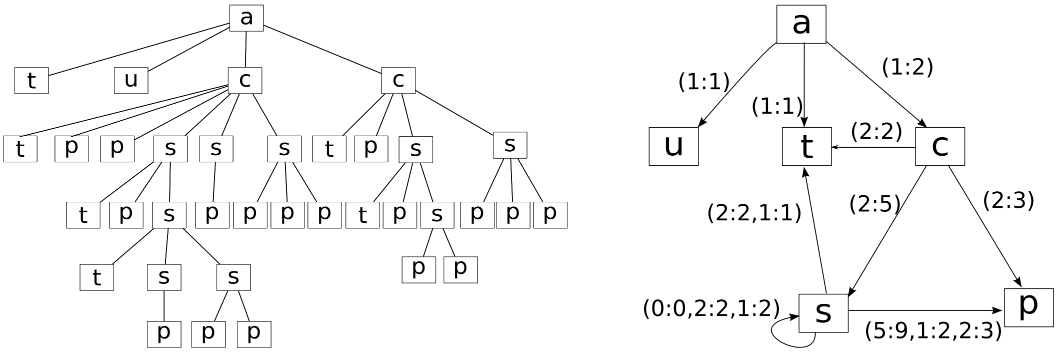
This method is substantially simpler than the suffix-* method. It does not use *-paths or suffix-* paths. Path classes of lower cardinality are simply discarded. As opposed to suffix-*, this is a *negative* method in the sense that, when it does not find a path class in the table when applying the cardinality formula, it will assume that no such path class exists in the XML tree and the whole cardinality computation will yield zero.

2.3 Xseed

This approach has been defined by Zang et al. (ZHANG et al., 2006). It is largely based on a graph-like structure. The objective of this work is to better represent recursive documents, i.e., those where an element can occur directly or indirectly under another element of the same tag name. It also supports recursive queries, i.e., those where a given tag name is tested more than once in the expression.

It consists of the following:

- A synopsis structure, called kernel, responsible for capturing structural information, as well as recursions in the XML documents.
- A structure called Hyper-Edge Table (HET), responsible for providing additional information about the tree structure. This structure can be dynamically built under a memory budget. HET enhances accuracy for path predicates in a query expression and is built based on a query feedback mechanism.
- An algorithm for traversing the synopsis structure in order to calculate path estimates.



(a) (b)

Figure 2.2: An example XML tree (a) and its Xseed kernel (b)

Figure 2.2(b) shows the kernel of the Xseed structure associated to the XML tree in Figure 2.2(a). In the kernel, each vertex in the graph represents a distinct element name in the document; each directed edge represents the existence of a parent-child relation between two vertexes, where the parent is the starting point of the edge, and the child is the ending point of the edge. Each edge has a label storing information regarding the occurrence of those relations. If there are multiple levels of recursion for that relation, the label will have information for each level separately.

For example, consider the edge $\mathtt{s} \rightarrow \mathtt{t}$ in the Xseed graph shown in Figure 2.2(b). This edge represent all relations $\mathtt{s/t}$ present in the XML document tree. As we can see in the document, we have, at two different levels, $\mathtt{s/t}$ relationships. At the third level, we have two $\mathtt{s}$ nodes linked to two $\mathtt{t}$ nodes in a $\mathtt{parent{-}child}$ relationship. At the next level, we have the same pattern, but now with only one $\mathtt{s}$ node related to one $\mathtt{t}$ node. Because of that, we annotated $(2:2,1:1)$, in the $\mathtt{s} \rightarrow \mathtt{t}$ edge of the Xseed kernel. This is called *recursion level* in the terminology used by Xseed.

3 HISTOGRAMS

3.1 Introduction

Data distributions are very useful in database systems, and particularly in XDBMS, but are usually too large to be stored accurately, so histograms come into play as an approximation mechanism. Approximate distribution of data plays an important role in the query optimisation process, because the distribution can be used to estimate the selectivity of (parts of) query expressions. For example, a PS (extended with cardinalities) could be considered a kind of hierarchical data distribution. Since two of the implemented approaches (PSUPS and LH) make use of histograms as an approximation mechanism of XML data, and some other published approaches use the same technique (WANG et al., 2004), it is worth to discuss histograms before we detail the approaches themselves.

We adapt the general histogram definition in (IOANNIDIS, 2003) as follows. A histogram on a set X of PS tree nodes is constructed by partitioning the data distribution of X into β ($\beta \geq 1$) mutually disjoint subsets called buckets and approximating the frequencies and values in each bucket in some common fashion. The number of buckets is in fact a heuristic, because there is no mathematical way (or at least, no computationally feasible way) to define an optimal number of buckets based only on a set of frequency values. Although work in (JAGADISH; KOUDAS; SEVCIK, 1998) proposes a method with quadratic complexity to compute bucket boundaries, normally, the number of buckets is heuristically set – based on, for example, the available storage space (budget), or on the number elements in a set, or roughly computed based on a some database characteristic, e.g. page size.

The histogram definition allows a great degree of freedom to characterise histograms. Hence, following (IOANNIDIS, 2003), we can characterise histograms according to following aspects.

- **Partition Rule:** This is further analysed concerning the following characteristics:
  - **Partition Class:** This indicates if there are any restrictions on the buckets. For example, if there will be some overlapping (or non-overlapping) buckets with respect to some parameter (the next characteristic).
  - **Sort Parameter:** This is a parameter whose value for each element in the data distribution is derived from the corresponding attribute value and frequencies. Attribute value (V) - which is, in our case, the element name (or its internal numeric representation), frequency (F) - which is, in our case, the number of occurrences of an element, and area (A) are examples of sort parameters that have been discussed in the literature.
  - **Source Parameter:** This captures the property of data distribution that is the most critical in an estimation problem and is used in conjunction with the next character-

istic in identifying a unique partitioning. Spread (S), frequency (F), and area (A) are the most commonly used source parameters.

  - **Partition Constraint:** This is a mathematical constraint on the source parameter that uniquely identifies a single histogram within its partition class. Several partition constraints have been proposed so far, e.g., *equi-sum*, *v-optimal*, *maxdiff*, and compressed. Equi-sum originates the so-called *Equi-height* and *equi-width* histograms, whereas v-optimal and compressed partition constraints yield the so-called *End-biased* and *biased* histograms, respectively. Equi-height and End-biased histograms are studied in sections 3.3 and 3.2, respectively. Many of the more successful partition constraints try to avoid grouping vastly different source parameter values into a bucket.

- **Construction Algorithm:** Given a particular partition rule, this is the algorithm that constructs histograms that satisfy the rule. It is often the case that, for the same histogram class, there are several construction algorithms with varying efficiency.

- **Value Approximation:** This captures how attribute values are approximated within a bucket, which is independent of the partition rule of a histogram. The most common alternatives are the continuous value assumption and the uniform spread assumption; both assume values uniformly placed in the range covered by the bucket, with the former ignoring the number of these values and the later recording that number inside the bucket.

- **Frequency Approximation:** This captures how frequencies are approximated within a bucket. The dominant approach is making the uniform distribution assumption, where the frequencies of all elements in the bucket are assumed to be the same and equal to the average of the actual frequencies.

- **Error Guarantees:** These are upper bounds on the errors of the estimates a histogram generates, which are provided based on information that the histogram maintains.

To give an example on how these aspects can influence definition and construction of histograms, we use the following notation: $H(p,s,u)$, indicating a histogram with a partition constraint p, a sort parameter s and a source parameter u. Consider the set of nodes in Table 3.1(a): its elements are tuples containing $\langle name, freq\rangle$, where *name* is the element's name and $freq$ is the frequency of that element in the context. Applying a $H(\textit{equi-height},V,F)$, the result is the histogram shown in Table 3.1(b). Notice how Equi-height distributes information of multiple elements among buckets covering certain *name* ranges, allowing for a complete overlapping on bucket formation. Applying a $H(\textit{end-biased},V,F)$ histogram, the result is the histogram shown in Table 3.1(d), where some entries are kept as they were and others are coalesced by their average in a bucket. Finally, applying $H(\textit{biased},V,F)$, we will have the histogram in Table 3.1(d), similar to the End-biased histogram, but with a slightly different distribution mechanism. In the next sections, we study each one of those approaches (Equi-height, End-biased, and biased).

3.2 Equi-height

The class of Equi-height histograms (EH) is well-known and used in most of the current commercial relational database products. In an EH histogram, the entries of the original set are sorted and partitioned into buckets of equal height, where the height is the sum of the source parameter of the entries inserted in the bucket. Each bucket is then labelled with a *start point* and an *end point*, where the start point is the sort parameter of the first entry in the bucket, and

name	freq
author	10
editor	3
price	5
title	1
year	19

(a)

name range	sum
author-editor	13
price-year	13
year-year	12

(b)

name	freq
year	19
author	10
avg(rest)	3

(c)

name range	sum
year	19
author-author	9
author-title	9

(d)

Table 3.1: A set of nodes (a) and its histograms: Equi-height (b), End-biased (c), Biased (d)

end point is the sort parameter of the last entry of the bucket. If a bucket holds only one entry, it will have equal start and end points. If an entry's value does not fit in a bucket (or what is left of it), only the portion that fits is inserted in this bucket, and the remaining portion is inserted in the next bucket. If the sum of all entries does not divide equally among all buckets, the remainder of the division is left on the last bucket, and an annotation of the value of such a remainder is made on this last bucket.

To implement an EH, one must first define which height will be the common height, either by a parameter or a heuristic. Normally, the height is a function on the number of buckets. Then, one must insert each entry in first available bucket, controlling the occurrence of overflows and labelling each bucket properly. In the end, the remainder of the entries must be annotated in the last bucket, in order to avoid assuming the last bucket is full when it actually is not.

3.3 End-biased

The class of End-biased histograms (EB) has the least approximation error among all histograms proposed in the literature (POOSALA et al., 1996). In an EB histogram, some number of the highest frequencies and some number of the lowest frequencies in a set are explicitly and accurately maintained in separate individual (*singleton*) buckets, and the remaining (middle) frequencies are all approximated together in a single bucket. Singleton buckets are thus buckets with zero approximation error.

The build algorithm for EB histograms is quite simple. Given a number of buckets β, use a heap to pick the highest and lowest $\beta - 1$ frequencies in a set and enumerate the combinations of highest and lowest frequencies in such way that the right combination minimises the following expression: $\sum(p_i * V_i)$, where $1 \leq i \leq \beta$, p_i is the number of frequencies in bucket b_i; and V_i is the variance of the frequencies in bucket b_i. This algorithm runs in $O(M + (\beta - 1) logM)$ time (IOANNIDIS; POOSALA, 1995), where M is the number of elements in a set.

3.4 Biased

We can intuitively think of biased histograms as a hybrid between Equi-height and End-biased histograms. In biased histograms, we have one (or more) singleton buckets representing the highest frequencies and the remainder of the frequencies represented in a Equi-height fashion. As shown in Table 3.1(d), we isolated the highest frequency entry "year" and constructed a Equi-height histogram of the remaining entries (from "author" to "title").

4 NOVEL APPROACHES

In this chapter, we suggest different approaches for the construction of Path Synopsis structures, as discussed in Section 2.1.1. These novel structures deviate from the previously described structures in that a tree-based approach is preferred over table- and graph-based approaches. Additionally, the previously discussed histograms, already in use in the relational world, are employed for the purpose of compressing data regarding cardinality distributions.

4.1 PSUPS - Path Synopsis with Uniform Parent Selectivity

Path Synopsis with Uniform Parent Selectivity – PSUPS is a tree-based approach that relies on histogram techniques to reduce the size of a PS. In a PSUPS tree, inner nodes store their information in the usual way, as they would be in the PS tree example shown in 2.1(b); only leaf nodes are considered for reduction by histograms and other shrinking techniques.

We define *candidate set* as a set of leaf PS tree nodes under the same PS tree parent. There can be only one candidate set under a given PS tree inner node, and it will be the largest set of leaf nodes possible. During the build time of this structure, it will be decided, according to the candidate set size (i.e. number of components) whether it is worth applying a reduction technique or not, and which reduction technique yields the smaller size in memory. The reduction techniques implemented for this structure are End-biased Histogram (see Section 3.3), Equi-height Histogram (see Section 3.2), and *Child-Shrunk*. Child shrunk is a special case of histogram where all elements of the candidate set have the same associated value, in which one only needs to store the shared value once.

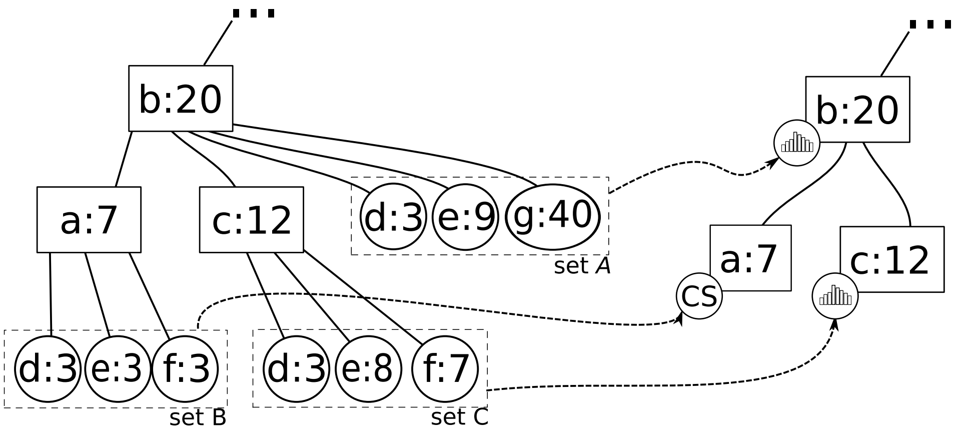


Figure 4.1: A PS tree cut-out and its PSUPS. Sets become histograms or child shrunks.

To understand the building process of a PSUPS tree, consider the PS tree cut-out in Figure

```
1  function buildPSUPStree(psTree, histogramType) {
2      psNode ← getNode(psTree );
3      leaves ← getLeafChildren(psNode );
4      inner ← getNonLeafChildren(psNode );
5      psupsNode ← addNodeToTree(psupsTree,psNode );
6      switch do
7          case all nodes in leaves have the same cardinality
8              create a childShrunk from the nodes;
9              annotate childShrunk in psupsNode ;
10         case $leaves.size() \leq 2$
11             add each node in leaves to psupsTree;
12         case $leaves.size() > 2$
13             create a histogram of type histogramType from leaves;
14             annotate histogram in psupsNode;
15     foreach node ∈ inner do
16         buildPSUPStree(node, histogramType )
17 }
```

Algorithm 1: PSUPS tree construction process

4.1. The PSUPS tree building algorithm traverses the original PS tree in a depth-first manner. The algorithm will, at some point, visits PS node `b` and, being `b` an inner node, it will be replicated as-is in the PSUPS tree. Then, it will put all leaf children of `b` in set A, which will become a histogram associated to `b`'s PSUPS node. Next, it will visit all of `b`'s non-leaf children, repeating this process recursively.

In Algorithm 1, we have a pseudo-code representation of the recursive PSUPS tree construction process described in the preceding paragraphs. Lines 2 through 5 are responsible for separating inner and leaf nodes into distinct sets and creating a PSUPS node itself, by copying information of the current node as is. Lines 6 to 14, decide which reduction mechanism, if any, will be adopted. Notice that there is a threshold, depending on the architecture, under which it is not worth to construct a histogram, as the histogram descriptor overhead would defeat the purpose of having a histogram to save storage space in the first place. For our architecture, the value of this threshold is two. Finally, after deciding a reduction technique for the leaf nodes, the algorithm proceeds recursively over the inner nodes.

The histograms of a PSUPS tree must allow estimation of a cardinality value for a given element name in the set. Thus, the histogram is constructed over a set of $\langle elementname, frequency \rangle$ pairs, meaning that the source parameter is the element name (E) and sorting parameter is the frequency (F). In terms of the notation used in Section 3.1, it is a $H(type,E,F)$ histogram, where *type* is selected by the user.

4.2 LH - Levels Histogram

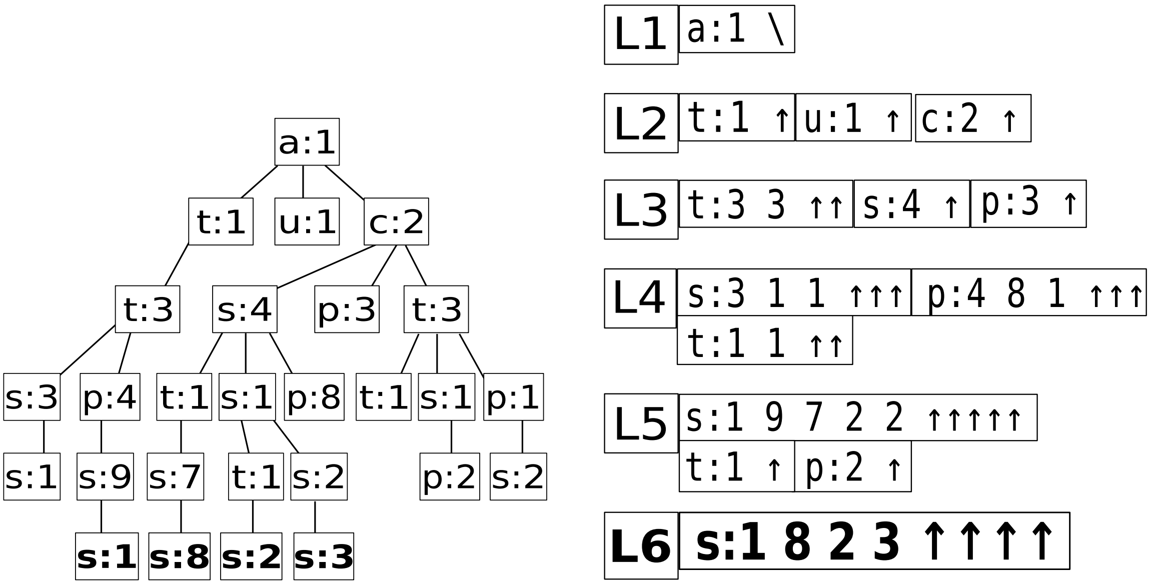
When attempting to summarise a PS structure by applying histograms, one has to decide how to group information meant to be summarised. In the case of PSUPS trees, one histogram is applied for each set of sibling leaf PS nodes, usually yielding only local histograms that are dependent on a context given by inner PS nodes. When a tree structure has such dependency, one can say it is *vertically* oriented, in the sense it is easier to store and traverse it in depth-first order.

In this section, we will study Level Histograms (LH), a PS tree structure *horizontally* oriented, in the sense that its traversal and storage will occur on a per level basis. This structure also relies on histograms, but it groups information for histograms in a level-oriented manner, i.e., it may group various PS nodes across the same level, regardless of their parents. LH aims to better capture structural information of highly recursive XML documents, but can also be effective for a non-recursive ones, e.g. dblp.

An LH structure is composed of a number of *occurrence maps*, one for each level of the PS tree. Each occurrence map describes the occurrences of each distinct element name across the corresponding PS tree level. The occurrences of a given element name on a level are captured in a left-to-right manner and stored as a tuple of $\langle cardinality, parent \rangle$, effectively reproducing the information previously available on the PS tree.

However, it is important to remark that, while cardinality is context-independent information and therefore easily storable, information regarding the parent of an occurrence is essentially a reference to another PS node. This could be solved using the index structures of the database system, but this would cause too much fragmentation, because every occurrence would require an index entry on which it could be referenced by its children. To solve this problem, we use the fact that the parent PS node is always a node of the preceding level and that by the time the level h is being built, the level $h-1$ is already built and stable.

We define the concept of *parent pointer* as a way to represent which occurrence of the preceding level was the PS node that is the parent PS node of the current element name occurrence being evaluated. In other words, once a level is built, each element name occurrence can be referenced as the parent by occurrences of the next level below. To store this information in a economical way, we use the fact that the order of elements in the preceding level is stable, i.e., will not change, and store only the order of the parent occurrence. Regardless of the underlying representation, one needs only $log_2(n)$ bits to store this order, where n is the total number of occurrences of all element names in the level above.



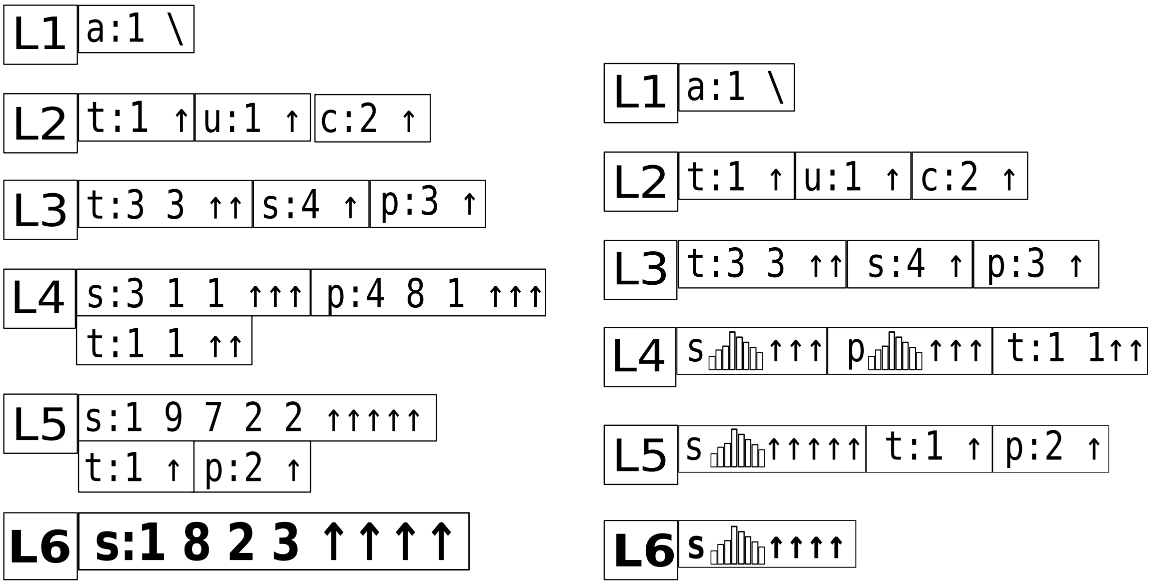
(a) (b)

Figure 4.2: A PS tree (a) and its LH representation (b)

As an example, consider the PS tree represented in Figure 4.2(a). In its LH representation (Figure 4.2(a)), each level will be represented as an occurrence map as described before. Consider the sixth level, which consists of four occurrences of element name s in the PS tree. In the LH representation, it is stored as an element name, the list of cardinalities in the order they are in the PS tree and a list of parent pointers (represented as a ↑ in the figure for brevity). For instance, the first entry of that level is a `s:1` node, child of node `s:9`, which is the second node out of the seven nodes of the preceding level. Therefore, the first pointer of that list will hold 2, and this pointer (and the others of that list) can be represented on $log_2(7)$ bits.

Further reduction in space requirements can be achieved by constructing histograms of the information. In the case of LH, we construct one histogram for each distinct element name on each level, relating each occurrence of that element name with a cardinality value. Parent pointers will not, of course, be considered for reduction, because it is not possible to calculate averages or similar operations over them. Unlike the histogram of PSUPS, where the position of a PS node among their sibling nodes is irrelevant, occurrences of an element name on a level in an LH structure must be kept in order to allow parent pointer references by the lower levels of the structure. Also, unlike the histograms of PSUPS, the histograms of LH are not constructed over a set of $\langle name, frequency\rangle$ pairs, because all occurrences represented by a histogram have the same element name. Therefore, each histogram is constructed over a set of $\langle pos, frequency\rangle$ pairs, where *pos* is the position of an occurrence in the set.

In terms of histogram parameters defined in Section 3.1, the source parameter of a histogram of an LH structure is the position (P) of the occurrence and the histogram is thus represented as $H(type, P, F)$, where type is user-selected.



(a) (b)

Figure 4.3: Applying histograms on LH

As an example, consider the sixth level of the LH structure in Figure 4.3(a). It consists of a list of occurrences for element name `s` with four occurrences. To construct a histogram for this set, we focus on the position of each occurrence on that list and its corresponding cardinality information. In this case, the key-value pairs are: $\langle 1,1\rangle$,$\langle 2,8\rangle$, $\langle 3,3\rangle$, and $\langle 4,3\rangle$. As is shown in Figure 4.3(b), the sixth level will be transformed by substituting the cardinalities with a histogram constructed the way we just described. Information like the element name and the parent pointers is kept intact.

Unlike PSUPS, access to this structure is not so straightforward. To discover the (estimated) cardinality of a path class, one visits a occurrence map for each step of the path class looking for the occurrence where the element name matches with the path class and parent pointer matches with the occurrence found on the preceding step.

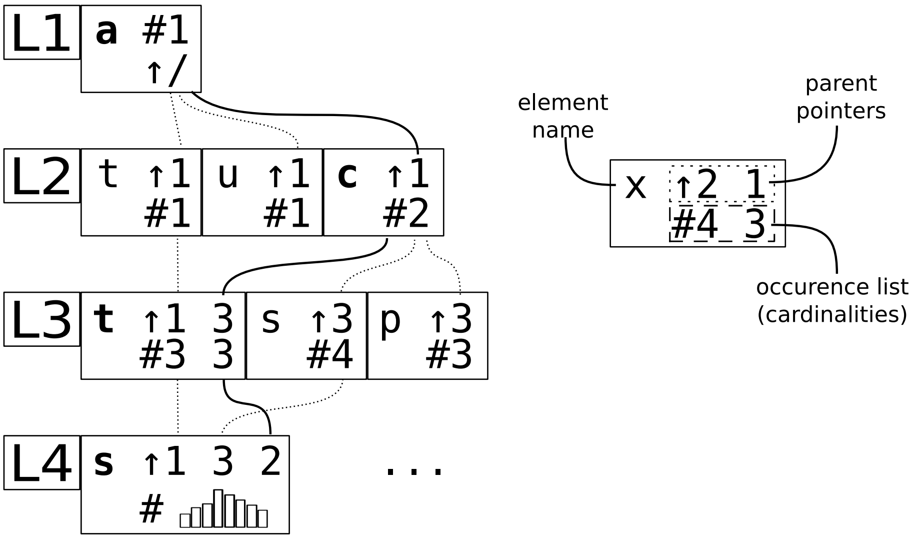


Figure 4.4: Cut-out of an LH structure with pointers

For example, consider the path class /a/c/t/s and the cut-out of an LH structure in Figure 4.4, where each pointer is above its corresponding entry in the occurrence map. The traversal described in this example is represented in Figure 4.4 by a bold line. One starts at level 1, finding a, which is a root node and therefore has its parent pointer set to null. The next step is going for level two (L2), looking for element name c with parent pointer 1; the third entry in occurrence map L2 satisfies these conditions. Then one must look for an occurrence of t, with parent pointer 3 on level 3; that is satisfied by the second entry of L3. Finally, we look for an occurrence of s, parent pointer 2 on level 4; the third pointer of L4 satisfies this condition and has its corresponding cardinality represented as part of a histogram. Therefore, one must seek, in the histogram, the entry whose sort parameter is three.

5 DISCUSSION

In this chapter, we discuss qualitative aspects of the PS structures studied in this report, explaining how these structures are expected to behave in face of different situations, which issues they have, and how those issues could be solved.

5.1 Classification criteria

PS structures, as explained in Chapter 2.1.1, may be implemented using different approaches in terms of base structure and completeness. Those design decisions are of great influence on the accuracy of the estimations made by the structure, mostly because each type of structure introduces estimation error in different situations, either by using information from histograms or by "guessing" deleted information from statistical models. Those design decisions also influence in terms structural limitations, i.e., features of query expressions or documents it cannot support. In Table 5.1, we have a classification of the structures implemented according to the classification criteria previously defined (see Section 2.1.4).

	Partial	Complete
Table-Based	Markov Tables	
Graph-Based		Xseed
Tree-Based		PSUPS, LH

Table 5.1: Classification of PS structures by completeness and base structure

Partial structures introduce errors when the queried path is longer than the maximum length pruning parameter defined at construction time of the structure. In case of Markov Tables, the use of further reduction techniques, such as suffix-* and no-* may also result in loss of precision. The occurrence of false positives, i.e., the estimation of a positive value for a query whose actual value is zero, is also common in this kind of structure, because their statistical models may presume the existence of certain path classes even though those path classes do not exist.

In complete structures, on the other hand, when a path class does exist in the document, it is guaranteed to be represented in some fashion in the structure. However, these structures introduce errors when the queried path requires the use of one or more substructures that cause data loss, such as histograms.

5.2 Qualitative analysis

To allow for a more objective analysis of the PS structures proposed, we define four qualitative criteria and study those criteria with respect to the PS structures studied. The criteria are defined as follows:

- **query axes support:** which navigational axes of Xpath (XPATH XML PATH LANGUAGE 2.0., 2005) can be used when querying this PS structure. The axes considered for this criteria are: `self`, `attrib`, `namespace`, `child`, `parent`, `descendant`, `ancestor`, `following-sibling`, `preceding-sibling`.
- **text value support:** is the ability of a PS structure to support storage and manipulation of statistical information regarding text nodes in an XML document.
- **recursion support:** is the ability of a PS structure to support estimation for either recursive XML documents or recursive XML queries in a graceful manner, i.e., without introducing too many errors.
- **storage space requirements:** is the scalability aspect of any PS structure, and may have a large impact on the performance of algorithm which uses the structure.

	query axes	text values	recursion support	space required
PSUPS	all (except `namespace`)	as an extension	handles with size penalty	depends on document height
LH	all (except `namespace`)	as an extension	handles better than PSUPS	scales better than PSUPS for recursive documents
MT	`child` and `parent`	none	none	flexible, trade-off for accuracy
Xseed	all (except `namespace`) (`*-siblings` are non-trivial)	none, but may be extended	designed for recursion	flexible, trade-off for accuracy

Table 5.2: Qualitative criteria overview

Table 5.2 shows an overview of various PS structures according to the qualitative criteria defined above. The first Xpath axis (`self`) is trivial and requires no implementation in the listed structures. The next axis (`attrib`) is inherently supported in the framework, because the system considers attribute names as if they were element names. `child` and `parent` are inherent to the path class concept, thus being supported in all structures. Axes `descendant` and `ancestor` require more elaborate estimation algorithms and are supported by all structures, except Markov Tables. Finally, `*-sibling` axes could be derived from parent-child relationships in PSUPS and LH as an extension, while Xseed would require some non-trivial mechanisms to control false-positive hits on such axes.

Regarding text value support, all the structures studied in this report were primarily designed only for storage of element and attribute distribution information. This implies that these structures do not support estimation of queries with predicates that involve text values, such as `/a/b/c[text()='e']`, because they do not store any information of the distribution of text nodes inside nodes with element name `c`. However, PSUPS and LH could be extended for such support by storing additional references (in nodes that have text values) to entries in an auxiliary structure that actually would store the text distribution data. The specification of these auxiliary structure is beyond the scope of this report, but there is plenty of research in this area in the literature (WANG et al., 2004; AL-KHALIFA; YU; JAGADISH, 2003). It should also be noted that PSUPS structures are somewhat easier to extend for this feature than LH structures, because it is easier to separate inner nodes from leaf nodes in the former structure.

The next aspect, recursion support, gives an interesting field of play for graph-based approaches like Xseed (studied in Chapter 2.3) and Xsketch (POLYZOTIS; GAROFALAKIS, 2006). A graph-based approach tries to collapse multiple recursion levels into smaller structures by making some sense of the repetition present in a recursive path class. For instance, given a path class `/a/c/s/c/s/p`, the second instance of step `c` could be coalesced into whatever is representing the first instance of `c` in the graph. This process has its drawbacks, in that there must be some flag or counter telling that a given recursion happens a limited number of times, and that sometimes it can collapse unrelated sub-structures and incur in false positives. For example, consider a query `/a/c/s/p` estimated by a structure holding the path class of the previous example: there are no `p` occurrences under the first instance of the sub-structure `c/s`, i.e., the estimation should be zero; however, as the two instances of `c/s` were coalesced, the estimation algorithm inadequately assumes there is always a `p` after a `c/s` sub-structure.

Markov Tables only support recursive documents and queries reliably if the value of the pruning parameter is almost the maximum height of the document, which defeats the purpose of choosing smaller path lengths in the first place. With lower, more reasonable, maximum length parameters, it may return correct or good results in well behaved cases, i.e. a recursive document whose structural distribution follows the short-memory model on which MT is based, but this is more of an exception than a rule.

Both tree-based approaches studied so far support recursive path classes, but they only focus their reduction techniques on horizontally disposed sets (as explained in Chapter 4); they do not attempt to collapse repetitive vertical structures the way a graph-based approach would, therefore incurring in storage space penalties. PSUPS and LH also differ in the impact a recursive structure has in the storage space required: PSUPS structure will only attempt reduction techniques on the very last step of a path class, and only if this step does not happen as a middle step of any other path class; LH, on the other hand, attempts reduction techniques at any level of the original PS tree, which is likely to result in middle steps of a path class being reduced.

The last aspect, storage space requirement, is of great importance to the performance of the optimisation algorithms that will use a PS structure. Some documents are easily summarised into tiny PS structures; others, however, require more space because of their irregularity, e.g. `treebank`. Some structures, such as MT and Xseed, can be adjusted to given memory budgets, by being more aggressive on their decision to discard information, usually at a cost on accuracy, because the amount of information being guessed from non-existing information is proportional to the space saved. In other structures, like PSUPS and LH, the amount of space saved depends more on the distribution of structural patterns: PSUPS is very efficient in flat documents, because a great proportion of the PS nodes are leaf nodes and, therefore, candidates for reduction; LH is good at supporting recursive or deeper documents, because it may group PS nodes into histograms regardless of their status as inner or leaf nodes. The parameters of the construction of histograms in PSUPS and LH could also be adjusted to achieve further reduction (again at cost of accuracy), but this would only give marginal advantage, because in the former structure, the larger volume of the structure is the representation of the inner nodes, and in the latter structure, parent pointers are a large, immutable part of the volume of the structure.

6 IMPLEMENTATION

In this chapter, we discuss implementation issues of the structures studied so far. First, in Section 6.1, we give an overview of the XTC XDBMS, a system developed as a testbed for XML manipulation, indexing, and querying techniques. In Section 6.2, we discuss the implementation of a new service for the XTC architecture, showing what features it exposes to another parts of the architecture and to the user and describing implementation details, i.e., how the framework was modeled and how it can be extended in the future. In Section 6.3, we discuss implementation details for each estimation structure such as classes and binary storage model.

6.1 The XTC XDBMS

XTC is a full-fledged native XML database manager which has proven to be effective for storing and controlling concurrent access to XML documents in a multi-user environment (HAUSTEIN; HARDER, 2007; HäRDER et al., 2007). It allows the use of several XML APIs, such as DOM and SAX (BROWNELL, 2002), and querying XML data through descriptive languages such as XPATH (XPATH XML PATH LANGUAGE 2.0., 2005) and XQuery (XQUERY 1.0: AN XML QUERY LANGUAGE, 2005). The XTC design strictly adheres to the well-known layered database architecture (HäRDER; REUTER, 1983). In Figure 6.1, we have an overview of the XTC architecture and its components, followed by a brief description of the most important XTC components.

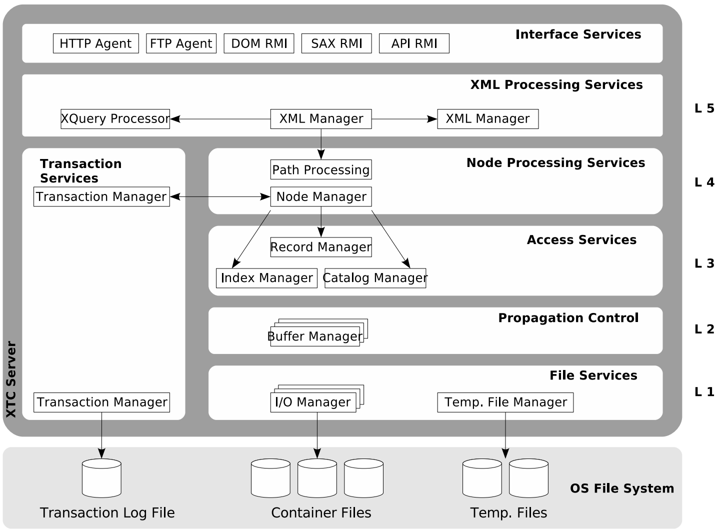
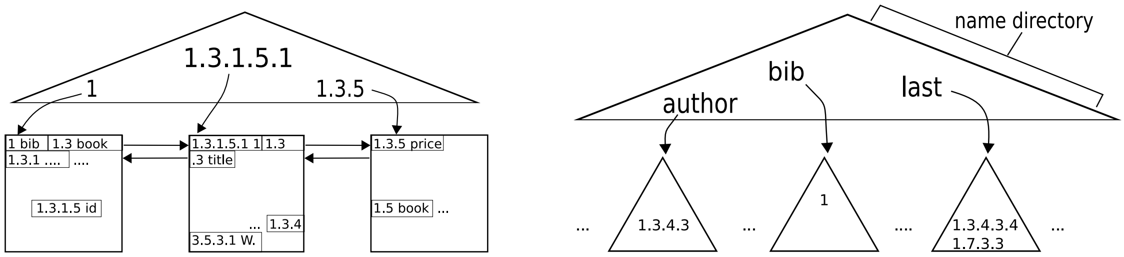


Figure 6.1: XTC architecture

The File Services Layer has the necessary functionality to attach and operate on external,



(a) Storage structure (b) Element index

Figure 6.2: XTC storage model

non-volatile storage devices. In collaboration with the operating system's file system, this layer copes with the physical characteristics of each type of storage device. The Propagation Control Layer provides for pages which are fixed-length partitions of a linear address space and mapped into physical blocks of the File Service Layer below. The Access Service Layer maintains all physical object representations, that is, data records, fields, etc. as well as access paths structures, such as lists, B- and B*-trees, as well as internal catalogue information. For performance reasons, the partitioning of data into pages is still visible at this layer. The Node Processing Services Layer maps physical objects to their logical representations and vice versa. For example, efficient support for navigation and structural joins thereby traversing hierarchies of XML nodes is mandatory in this layer.

The XML Processing Services Layer provides logical data structures with declarative operations or a non-procedural interface to the database. It provides an access-path-independent data model with descriptive languages.

In XTC, an XML document is stored into a B*-tree in which prefix-based labelling, the *SPLID* scheme (stable path labelling identifier) in XTC terminology, is used. The resulting structure is called document store (Figure 6.2(a)). Thus, XTC controls the document order and enables fast direct and navigational accesses (parent/child/sibling) to XML data. To encode the element names in an XML document, XTC maintains a vocabulary which is represented by an ordered list of $\langle VocID, name \rangle$ pairs where VocID contains the internal numeric representation of an element name or attribute name. Using B-tree indexes, an element index is additionally created and maintained by XTC. The element index keeps a name directory with (potentially) all element names – so-called tags – occurring in an XML document. Each tag entry in the name directory points to a set of SPLIDs (or DeweyIDs (HAUSTEIN; HARDER, 2007)) in document order, which address all corresponding element nodes in the document store (Fig. 6.2(b)). In fact, all SPLIDs indexing the nodes of a single element name are organised as a B*-tree which maintains the document order among the SPLIDs and enables direct access to each of them. With these storage mechanisms, XTC provides all "external" input sets for querying data in document order.

We have extended the XML Processing Services Layer of the XTC with summarization structures to support XML cost-based query optimisation. We have not modified any other layer, just used its services. More specifically, we have made an extensive use of both: VocIDs to represent nodes in the implemented structures (see Chapter 4 and following); and indexes (B*-tree indexes) to store such structures in the XTC database.

6.2 Estimation framework

All relevant implementation is done by a package named `docStatInfo`. It covers the features of creating, viewing, listing, and deleting estimation structures for a given XML document.

It also covers the actual estimation process for a given Xpath query expression on a given document in all structures. It is also possible to estimate only specific steps of a query expression, what, we believe, is the feature of most interest for future query planning and optimisation packages.

In order to use the services of `docStatInfo` package, one must access an instance of `XTCdocStatInfo`, which is a core class responsible for exposing all the features listed in the above paragraph. The services implemented are detailed below:

- **runXmlStats:** when invoked, builds an estimation structure according to the parameters passed and stores it using indexing services of XTC. All structures must implement a serialisation and deserialisation process in order to allow reloading the structure when the system is restarted.

- **viewXmlStats:** when invoked, shows the content of structures selected according to the parameters passed. All structures must implement a visualisation method through the `XTCestimator` interface.

- **listXmlStats:** similar to `viewXmlStats`, but only gives a short description (mostly construction parameters) of structures stored in memory.

- **deleteXmlStats:** deletes all structures matching the parameters passed from the memory.

- **estimateExpression:** returns an analysis of cardinality and selectivity information for a given Xpath query expression on a document according to the structures matching the parameters passed. Depending on the structure, only cardinality is returned (such as Xseed), or only some specific types of queries are possible (such as Markov Tables).

- **estimateStep:** returns a selectivity or cardinality estimation for a specific step of an Xpath query expression in the structures selected by the parameters passed. However, not all structures provide this feature.

- **generateQueryWorkload:** generates a comprehensive query workload for a given document, described in better detail in Section 7.2.

As shown in Figure 6.3, the core of the package consists of `XTCdocStatInfo` and the classes it uses to manage all estimation structures.

- `XTCdocStatInfo` is responsible for making the services mentioned above to other parts of XTC.

- `XTCestimator` is the interface all estimation structures must implement to be available as part of the framework.

- `XTCestimatorDescriptor` defines a way in which estimation structures can be described in terms of their type and their construction parameters. Each `XTCestimator` instance must have a corresponding descriptor. It also defines **enum** structures in order to provide a clear namespace.

- `XTCestimatorList` manages a set of estimation structures for a given document, wrapping serialisation and deserialisation routines of diverse estimation structures, providing a way to select groups of estimation structures for further use according to their construction parameters.

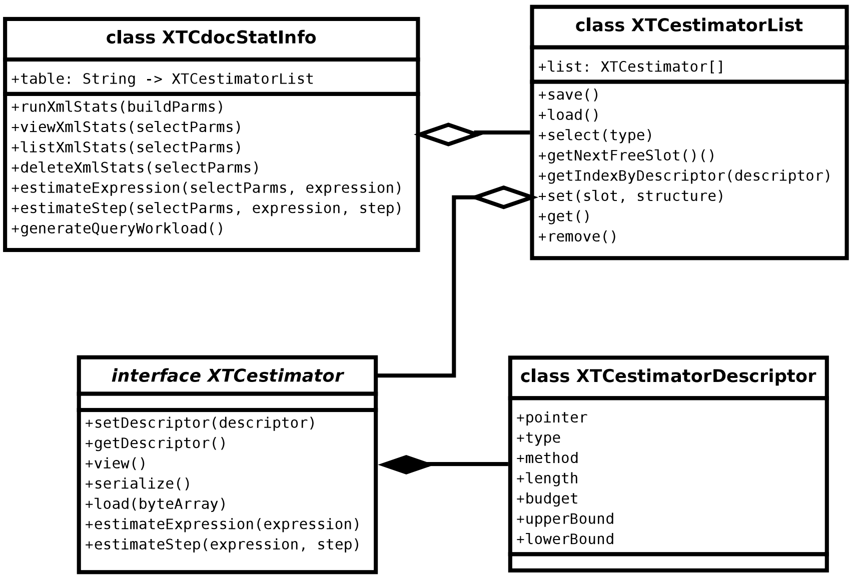

class XTCdocStatInfo
+table: String -> XTCestimatorList
+runXmlStats(buildParms)
+viewXmlStats(selectParms)
+listXmlStats(selectParms)
+deleteXmlStats(selectParms)
+estimateExpression(selectParms, expression)
+estimateStep(selectParms, expression, step)
+generateQueryWorkload()
class XTCestimatorList
+list: XTCestimator[]
+save()
+load()
+select(type)
+getNextFreeSlot()()
+getIndexByDescriptor(descriptor)
+set(slot, structure)
+get()
+remove()
interface XTCestimator
+setDescriptor(descriptor)
+getDescriptor()
+view()
+serialize()
+load(byteArray)
+estimateExpression(expression)
+estimateStep(expression, step)
class XTCestimatorDescriptor
+pointer
+type
+method
+length
+budget
+upperBound
+lowerBound

Figure 6.3: Core classes of the XTCdocStatInfoPackage

To implement the feature of reloading estimation structures after the system is restarted, the framework stores it on disk using XTC storage and indexing services. An entry is added to the catalogue of the document pointing to a structure called `statPage`, which holds serialised versions of descriptors of estimation structures. `statPage` is actually a physical page in the XTC container file that is created by the building process of the first PS structure built for a given document. Each descriptor in a `statPage` holds a pointer to an index entry where its associated PS structure is stored and the parameters with which it was built. The format of the record for a descriptor on the `statPage` is shown in Figure 6.4. Each record consists of 16 bytes, where the first four are an integer holding the number of the index holding the structure, followed by one byte holding the type of the structure. The next 9 bytes will store *method* (1B), *length* (2B), *budget* (4B), *lowerbound* (2B), *upperbound* (2B). Each structure may interpret these parameters freely, assigning any semantic or even ignoring them at all.

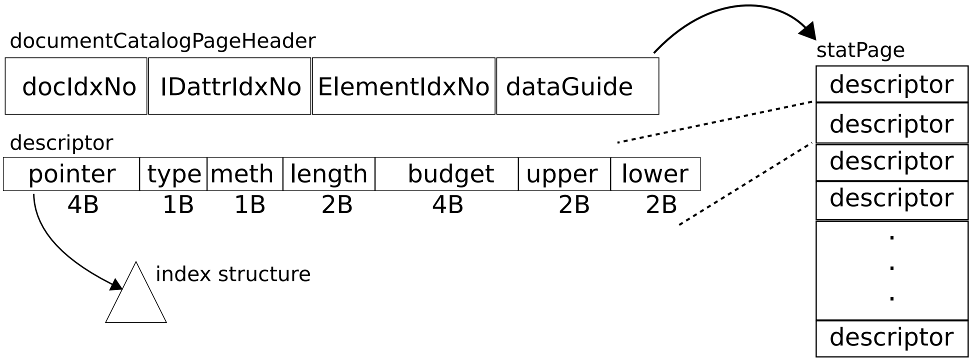
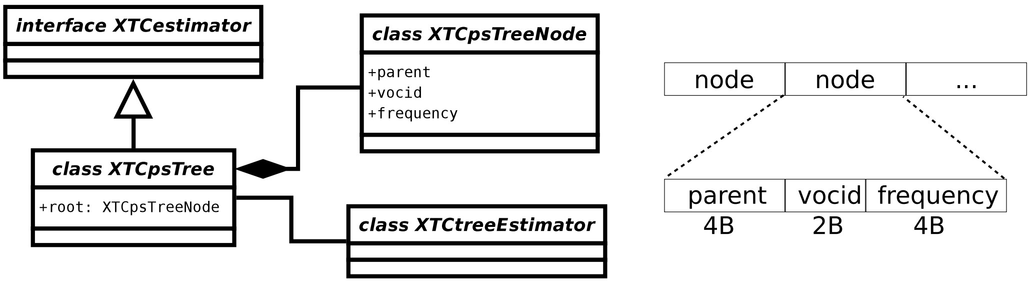


Figure 6.4: Storage model of the `statPage`

The `statPage` structure also allows for easy verification of whether or not a structure with a specific parameters has already been built. To check for that, one needs only to load the `statPage` structure looking for a descriptor matching the desired parameters. This type of checking is done by the `runXmlStats` service in order to avoid rebuilding already existing structures. The user may choose, however, to force such rebuilding, which is useful when the XML document itself has changed; in this case, the old structure will be deleted and the new



(a) (b)

Figure 6.5: Class diagram (a) and storage detailing (b) for the reference structure

version will be written over it.

6.3 Implemented Structures

Initially, a reference structure was implemented, the **PS tree**, from which most of the other structures are derived at construction time. This structure is a very basic tree-based implementation of a PS structure, employing no reduction techniques, thus giving always correct values for any given path class. All other structures are essentially an improvement over the PS tree, trying to solve its scalability problems with diverse reduction techniques.

PS tree is the reference structure and its implementation is provided by classes `XTCpsTree` and `XTCpsTreeNode`, where the former is responsible for implementing the `XTCestimator` interface, with its build, serialisation and deserialisation contracts, and the latter is responsible for storing element name and cardinality information for each node in the structure. Estimation features are implemented by `XTCtreeEstimator`, which is shared by PSUPS implementation, mostly because the traversal mechanism is the same for both of them, with only the accuracy possibly varying.

In all structures, the element name, which is originally a string value of arbitrary length, is stored with the XTC internal representation, named *VocId*. A VocId is currently a short integer (2 bytes) value that is attributed to each distinct tag name when a document is loaded. If two different loaded documents share the same tags, the same VocId will be used. Cardinality information is usually stored as an integer (4 bytes), but structures that work on average or other formulas may use float (4 bytes) or double (4 bytes).

For example, consider the storage model for the reference structure PS tree, shown in Figure 6.5(b). Each tree node takes 10 bytes to be represented: four bytes store the index location where the parent node of that node is stored, then two bytes to store a VocId corresponding to the element name of that node and finally four bytes to represent the cardinality of that node. In other structures, storage size for each entry or node may vary depending on the parameters applied to histograms and memory budget constraints.

In order to add another estimation structure to the framework, the `XTCestimator` interface must be implemented and its contract obeyed. If the structure is built by parsing the whole document (such as PS and Xseed), it must define its own parser. On the other hand, some estimation structures like Markov Tables, LH, and PSUPS are based on already existing structures such as PS, and require no parsing to be built.

6.3.1 Histograms

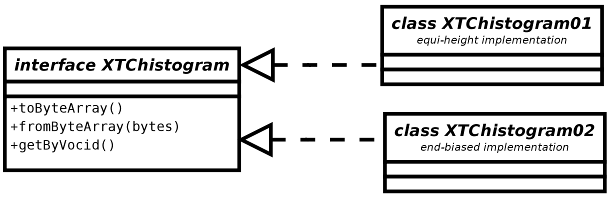


Figure 6.6: Class diagram for histograms

Many of the structures that follow share implementations of histograms. To add a histogram to the framework, one must implement the `XTChistogram` interface. The general contract for this interface is to construct a histogram from a given set of *key-value* pairs that can be retrieved by later queries to that histogram. Histograms must also define serialisation and deserialisation routines, in order to be reloaded when the server is restarted. We have implemented two models of histogram, namely the Equi-height and the End-biased histograms.

The Equi-height histogram consists of series of buckets sharing a common height, where the height is the sum of cardinalities of entries covered by a bucket. Each bucket covers a specific range of VocIds determined by its *start* and *end* points. If the distribution was uneven, i.e., the last bucket could not be filled, it must also describe the how much cardinality was left on the last bucket. However, in terms of storage, as shown in Figure 6.7(a), it can be assumed that the start point of a bucket is equal to end point of the preceding bucket. Thus, the storage consists of only (a) the common height, (b) the number of buckets, (c) the cardinality left on the last bucket, and (d) the end point of each bucket . The number of buckets is variable, and can be defined by heuristics or manually as a tuning parameter.

The End-biased histogram consists of series of singleton buckets - buckets with information of one single entry - where both the VocId and the cardinality are stored, and an average bucket, where the average of the cardinality of the remaining entries is stored. To minimise errors in estimation, some grouping criterion must be devised. In our case, we try to store the more skewed entries in singleton buckets by choosing the group of entries with the least variance to form the average bucket. As shown in Figure 6.7(b), in terms of storage, we need to store (a) the number of singleton buckets, (b) an average bucket, and (c) the VocID and the cardinality of each bucket.

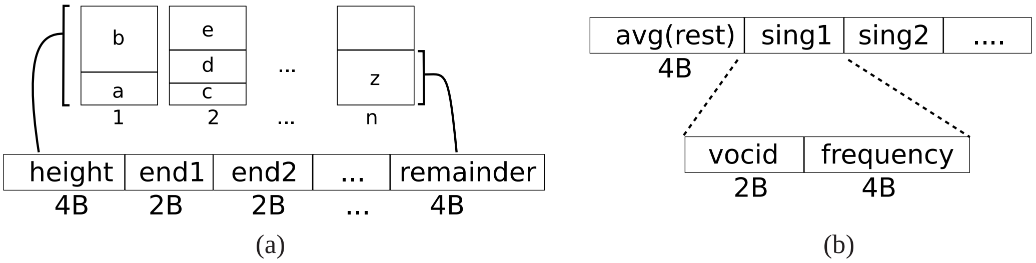
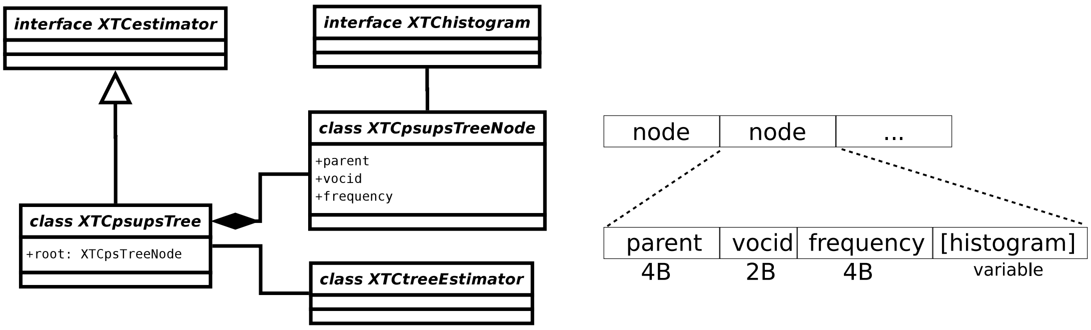


Figure 6.7: Storage model for Equi-height (a) and End-biased (b) histograms

In some situations, those histograms may return values for entries that were not originally in the candidate set, mostly because the histogram cannot tell exactly which entries were coalesced into each bucket (Equi-height) or into the average bucket (End-biased). To solve this, we have implemented an optional mechanism called BitField, that avoids false positives in situations where the sort parameter values for a given set were discontinuous. To accomplish that, a stream of bits represents which entries actually existed in original set, by having the bits associated with existing entries set to 1 and the bits associated to non-existing entries set to 0. There will be as many bits in the BitField as the difference between the smallest and largest VocIds involved in that set. The use of this mechanism can significantly affect the precision of estimations in some types of queries, as we discuss in Section 7.3.

6.3.2 PSUPS

An implementation of the PSUPS tree structure was made as a derivation of a PS tree structure, implementing the algorithm described in Section 4.1. As shown in Figure 6.8(a), PSUPS is implemented by `XTCpsupsTree` and `XTCpsupsTreeNode`, with the former being responsible for implementing the `XTCestimator` interface, with its build, serialisation and deserialisation contracts, and the latter being responsible for storing element name, its cardinality, and possibly a histogram representation of its children. Estimation features are implement by `XTCtreeEstimator`, which is shared with the PS implementation, because the traversal mechanism is the same for both of them, with only the accuracy possibly varying. To implement histogram representations, it may use any class implementing the `XTChistogram` interface, as described in the previous section and may have different accuracy results depending on the use of the bitfield structures in its histograms.



(a) (b)

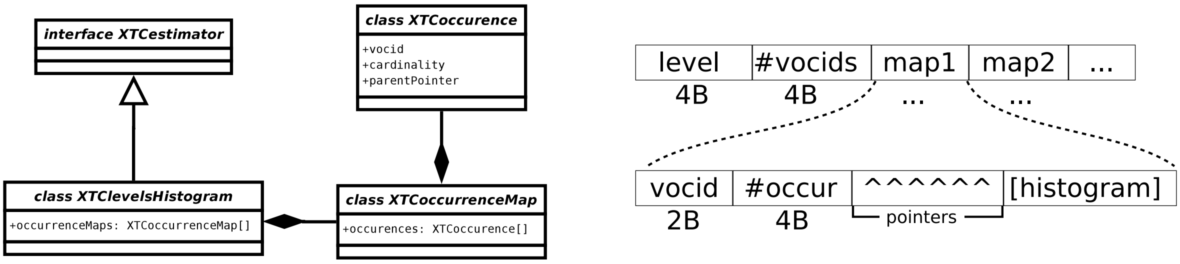
Figure 6.8: Class diagram (a) and storage detailing (b) for the PSUPS structure

The storage model of the PSUPS structure is based on the one of the PS structure, the main difference being that leaf nodes may be stored as a histogram annotation of their parent node. In situations where implementing a histogram causes too much overhead, the structure stores the leaf nodes as they would be stored in the PS structure. As shown in Figure 6.8(b), each PSUPS tree node consists of at least 11 bytes: four bytes to reference a parent in the index; one byte for the type of node, describing which kind of histogram this node has, if any; two bytes for the VocId; four bytes for the cardinality.

6.3.3 LH

Like the PSUPS structure, the implementation of the LH structure described in Section 4.2 was made as a derivation of a PS tree structure. As shown in the diagram in Figure 6.9(a) this implementation consists of classes `XTClevelsHistogram`, responsible for implementing the `XTCestimator` interface, and `XTCoccurrenceMap`, responsible for holding and accessing multiple instances of `XTCoccurrence`. To implement histogram representations proposed by this structure, any class implementing the `XTChistogram` interface may be used. The sort parameter for a histogram in an LH structure is based on the position of elements in a set, which means that the possible sort parameter values for a set are continuous; therefore, it does not make sense to apply false-positive preventive measures like BitField (described in Section 6.3.1).

Despite the fact that LH is a tree-based structure, its storage is oriented internally by levels. There is an index entry for the occurrence map of each level, stored in a sequential fashion, i.e. the index entry associated to level n is stored in the index with the key n, right after the entry associated to the preceding level $n - 1$. As detailed in Figure 6.9(b), each level consists of: an



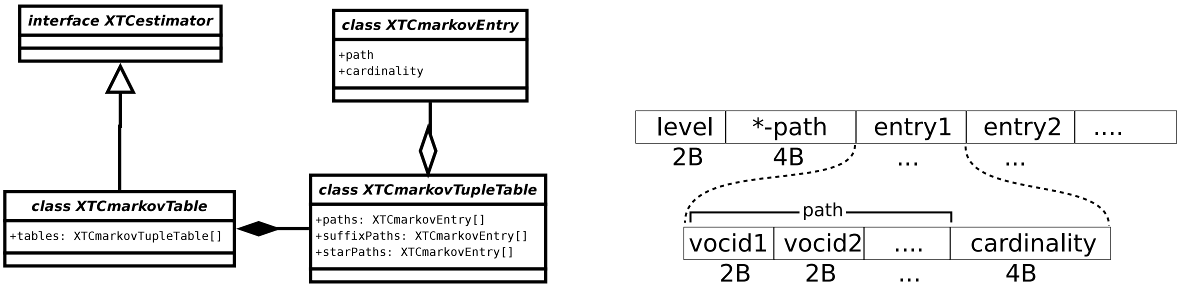
(a) (b)

Figure 6.9: Class diagram (a) and storage detailing (b) for Level Histograms

integer (4 bytes) indicating the number of element names occurring at that level; then, for each element name, the corresponding VocId (2 bytes) and an integer (4 bytes) with the number of occurrences of that element name at that level, followed by the list of parent pointers for the occurrences of that element name, followed by a histogram representation of the cardinalities of all occurrences of that element name.

6.3.4 Markov Tables

The implementation of the Markov Tables structure, as described in Section 2.2, was made as a derivation of the PS tree structure, traversing it to obtain all path classes (root-based or not) up to the length defined by the parameters used in construction time. As shown in Figure 6.10(a), this implementation is done by the following classes: `XTCmarkovTable`, responsible for implementing the `XTCestimator` interface; `XTCmarkovEntry`, responsible for storing information of a given path class; and `XTCmarkovTupleTable`, responsible for controlling access to instances of `XTCmarkovEntry` and performing reduction methods suffix-* and no-* as described in sections 2.2.1 through 2.2.2.



(a) (b)

Figure 6.10: Class diagram (a) and storage detailing (b) for Markov Tables

Being a table-based structure, it is fairly easy to implement a storage model to a Markov Table. For each path class length possible, i.e., as many as the parameter *m*, there is an index entry associated, stored sequentially, as is done in LH storage model (see previous section). Then, in each index entry, we store the associated path length in a short integer, followed by a *-path (depending on the reduction method employed), followed by all other instances of `XTCmarkovEntry` (including suffix-* entries) stored sequentially, each entry stored as an array of the VocIds associated to the element names forming the path class of the entry and a float representing the cardinality. The “*” path step is represented with a special value of VocId. This process is better illustrated by the diagram in Figure 6.10(b).

7 EXPERIMENTAL RESULTS

In this section, we show and discuss experimental results of diverse aspects of the PS structures. We evaluate the behaviour of each structure in face of diverse types of XML documents, by analysing their required storage size, their build time and the accuracy of their estimations.

We have implemented a PS tree reference using the framework discussed in the previous chapter, in which the cardinalities of all path classes are stored without any reduction technique, thus always keeping zero estimation error. This structure also serves as a reference for analysing the reduction provided by each PS structure. In some cases, this basic structure cannot handle the XML document without requiring impractical storage space, thus motivating different approaches and various reduction techniques.

In this chapter, we will detail the performance of all PS structures studied in this work. In Sections 7.1 and 7.2, we discuss the workload used for experimentation, both in terms of selected documents and queries used against them. In Section 7.3, we show and discuss the size × accuracy results found in our experimental procedures.

7.1 Document Workload

For the purposes of this experiment, we have selected a set of XML documents widely used in the literature as a testbed for XML related research. Those XML documents are a sample of the documents with which an XML DBMS might have to deal in a real-world use scenario, rather than synthetic documents like the ones tested in some experiments in the literature (ABOULNAGA; ALAMELDEEN; NAUGHTON, 2001).

The selected documents are the following: `uniprot`, a central repository of protein sequences and function created by joining the information from various sources; `dblp`, a collection of Computer Science related bibliographies; `psd7003`, a database of protein sequences; `nasa`, a repository of astronomical data; `swissprot`, which is one of the sources used to build the `uniprot` mentioned before; finally, `treebank`, a text corpus extracted from Wall Street Journal archives, in which each sentence has been annotated with its syntactic structure in a tree-like manner.

An analysis of the selected XML documents is provided in Table 7.1. In this table, we show diverse characteristics of the documents. The columns of this table represent the documents selected for this experiment. The first data row lists the size of the original XML document in Megabytes. The second row lists the number of elements and attributes present in the document. The third line lists the number of distinct element names, represented in the system as VocIds, as previously explained in Section 6.3. Fourth row lists the number of text nodes presents in the document. The fifth rows lists the size of the reference PS tree structure associated to the document, followed by the number of nodes of such structure in the sixth row. The seventh row lists the maximum depth of the document, i.e., the largest path class present in the document. Finally, the eighth row lists the average depth of the document, i.e., the average of the size of each instance of each path class.

document	uniprot	dblp	psd7003	nasa	swissprot	treebank
size (MB)	1,820	330	716	25.8	109.5	86.1
elements	81,983,492	9,070,558	22,596,465	532,967	5,166,890	2,437,667
vocids	89	41	70	70	100	251
text nodes	53,502,972	8,345,289	17,245,756	359,993	2,013,844	1,391,845
PS (bytes)	1,932	2,004	1,164	1,332	3,168	5,419,984
PS (nodes)	161	167	97	111	264	338,749
max. depth	7	7	8	9	6	37
avg. depth	4.53	3.39	5.68	6.08	4.07	8.44

Table 7.1: Characteristics of the selected XML documents

From the data shown in Table 7.1, we can see that the selected XML documents can have different “shapes”, i.e., their structural distributions can greatly vary. Some documents, despite requiring vast amounts of storage space, have quite regular structures, in the sense that the few existing path classes are repeated many times in the document; those documents are also known as *flat* documents. On the other hand, some documents, despite being relatively small, have a quite complex structure with a huge amount of path classes; those documents are usually of recursive manner, in the sense that a given element name may be present under another instance of the same element name, thus being called *recursive* documents. Flat documents usually have a signature of low maximum depth and few distinct element names. Conversely, recursive documents can be identified by their high maximum depth and the abundance of element names, additionally, it is common for recursive documents to have a higher deviation of the maximum depth in relation to the average depth.

One example of flat document is `uniprot`, which is mostly formed by large groups of entries sharing the same structure but different values. On the other side of the spectrum, we have recursive documents such as `treebank`, formed by a number of structures that are deeply nested and highly recursive. In this table, we show that recursive structures are likely to require an order of magnitude more of storage space for PS structures, which is a problem if one considers that PS structures should have fast access and load times. A PS structure with 5 Megabytes, such as the one associated with `treebank` would likely bring excessive I/O operations, which is not desirable in query optimisation algorithms. Therefore, it is necessary to study alternative structures that can reduce storage requirements in detriment of accuracy. As discussed in Chapter 5, characteristics of the reduction techniques applied affect the quality of this space/accuracy trade-off. The remaining documents are are the middle-ground, somewhat closer to the flat side of the spectrum, but are nevertheless of diverse structural distributions and of interest for this study.

7.2 Query workload generation

One important aspect of any experiment involving Xpath query expressions is the selection of expressions to be tested. One could accidentally choose those where errors happen to be minimal or non-existent and skew the results. To solve this problem, we devise a way to select a set of query expressions that, we believe, is fair and unbiased, by defining four classes of query expressions and methods of selecting queries in each of those classes.

- *Simple Parent* (SP) expressions are root-based Xpath expressions where only `parent` relations are tested, e.g. `/a/b/c`. All path classes of the document are selected for this class;

- *Simple Descendant* (SD) expressions are those where `parent` and `descendant` relations are tested, with the restriction that `descendant` relations are always the last step of the query, e.g., `/a/b//d`. Expressions are generated for this class by taking an SP expression, removing some steps and adding a randomised element name as the `descendant` step.

- *Predicate Path* (PP) expressions are SP expressions where a predicate is tested, and those predicates may be full SP or SD expressions connected by logical operators like `AND` and `OR`, e.g. `/a/b[./c]`, `/a/b[./c and ./d]`, `/a/b[./c or ./d//e]`. Expressions are generated by taking SP expressions and adding other randomised SP expressions as predicates of their last step.

- *Negative Query* (NQ) expressions are expressions of all the abovementioned classes to which the correct result is zero, but that may fail depending on how the PS structure was designed and with which parameters it was built. To generate them, we take positive queries of the other classes and randomise some of their steps looking for nonexistent path classes.

7.3 Result evaluation

In this section, we will analyse the results of experiments done on all structures discussed in this report, using the query workload over the selected documents, as described in the previous sections. We will discuss aspects of the resulting size of the PS structures in Section 7.3.1, followed by a brief discussion of the times required for building each structure in Section 7.3.2. Finally, we discuss the accuracy of each PS structure using different error metrics in Section 7.3.3

7.3.1 Size

Memory space consumption is analysed for each approach and for each document. As a general observation, we reach high reduction factors. For example, in almost all documents, the sizes vary between 0,001% and 0,055% of the original document sizes. Even for treebank, the space required for the summarization structures is between 0.2% (XSeed) and 2.3% (UPS) of the document size (Figure 7.1). MTSuf* structures exhibit the least space consumption for highly recursive documents, outperforming thus LH and PSUPS for such documents. In MT and Xseed, to achieve good reduction rates, a great deal of accuracy is lost, when compared to PSUPS and LH. PSUPS is more accurate than MT and Xseed in general, but it does not scale well concerning memory consumption when highly recursive documents are summarised. LH, on the other hand, keeps the accuracy while scaling well for any type of document. PSUPS consumes about 2MB for `treebank`, while LH and XSeed use 760KB and 50KB, respectively for such a document (see Table 7.2).

Variations of PSUPS sizes are due to choice histogram types and histogram parameters, and irregularity degree of XML documents. Normally, an EB histogram requires more storage space than an EH one. Moreover, we have implemented EH histograms in a non-canonical way. Instead of storing each bucket as a pair $(boundary, estimate)$, which consumes about 6 bytes per bucket, we store a sequence of boundary values, each one consuming 2 B, and the total sum of frequencies. If we divide the sum of frequencies by the number of buckets, we obtain the height of each bucket. The bucket estimation is calculated in a straightforward way, by averaging the values in a bucket with the calculated height. Thus, this implementation gives us all information needed to compute the estimate for each bucket without explicitly storing such estimation. For

	nasa	psd7003	dblp	swissprot	uniprot	treebank
Document size	25.79 M	716 M	330 M	109.5 M	1.77 G	86.09 M
PSUPS EB	0.77 K	0.5 K	0.72 K	1.45 K	0.92 K	1.95 M
PSUPS BitField	0.79 K	0.54 K	0.86 K	1.56 K	0.95 K	2.04 M
PSUPS EH	0.74 K	0.41 K	0.64 K	1.3 K	0.83 K	1.85 M
LH	1.2 K	1.12 K	1.12 K	1.83 K	1.58 K	745 K
MT Uncomp	1.2 K	1.14 K	1.33 K	2.63 K	1.60 K	15.2 K
MT Suffix-*	0.37 K	0.33 K	0.41 K	0.36 K	0.36 K	0.38 K
MT No-*	1.11 K	1.14 K	0.94 K	1.23 K	1.22 K	0.38 K
Xseed	1.46 K	1.38 K	1.75 K	3.40 K	2.01 K	49.22 K

Table 7.2: PS structure sizes in bytes for different documents

example, a 10-bucket histogram requires 60 bytes in a straightforward implementation, whereas our implementation only needs 24 bytes.

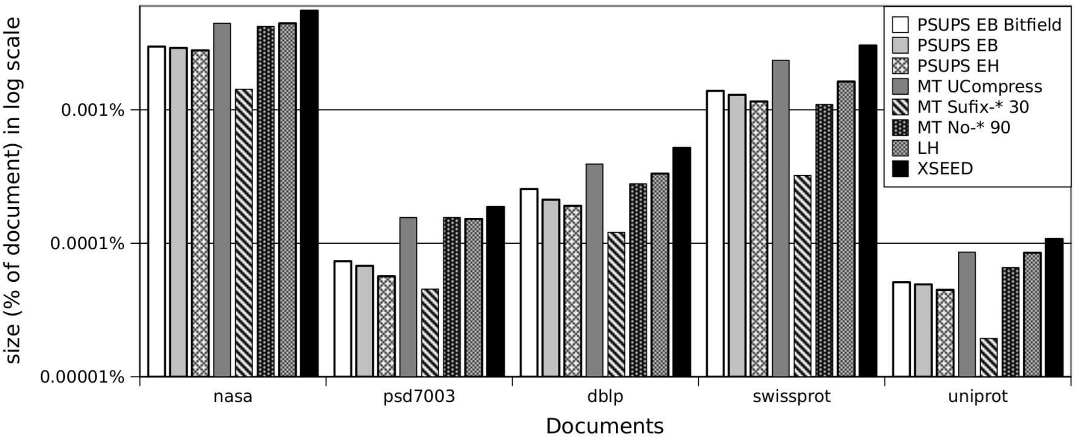


Figure 7.1: Comparison of structure sizes for different documents

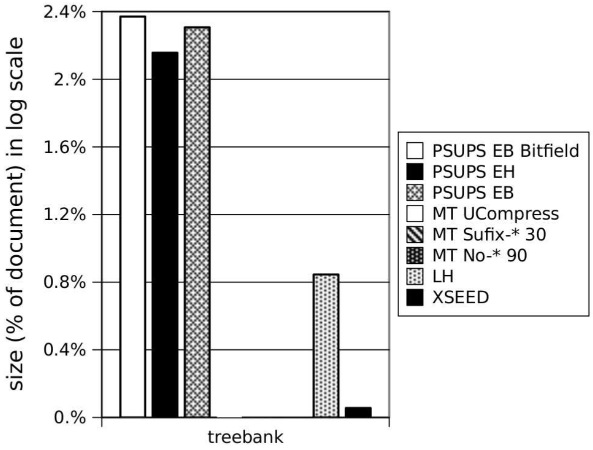


Figure 7.2: Comparison of structure sizes for `treebank`

7.3.2 Building time

We have performed our experiments using a 2-GHz Pentium Centrino Duo processor with 1 GB main memory, running under Windows XP SP2. We have implemented and integrated all approaches using the estimation framework described in Chapter 6, which was written in Java 6. Throughout the performance measurements, we use 512 MB of main memory for the Java Virtual Machine and 16 MB for the XTC buffer.

document	EB	EH	LH	XSeed
uniprot	0.328	0.547	0.063	339
dblp	0.016	0.031	0.031	51
psd7003	0.063	0.078	0.016	96
nasa	<0.001	<0.001	0.016	2
swissprot	0.015	0.016	0.141	22
treebank	0.594	0.688	0.844	11

Table 7.3: PS structure building times for different documents

Table 7.3 shows the building times for UPS and LH trees (derived from the corresponding PS trees present in memory) as well as the time for building the XSeed kernel. Building times of MT are always under 0.01 seconds, so we have omitted them here. We have observed that typically building times are very short (almost insignificant when compared to document scan time). If we take into account the whole process, i.e., document scan and tree construction, it takes just some minutes for each one. Of course, huge documents, such as uniprot, require longer processing times. Even for a highly recursive XML document such as treebank, we have computed LH in just about 0.8 second and XSeed in approximately 0.18 second.

7.3.3 Accuracy

To calculate the accuracy of all approaches, we calculate two metrics: Relative Error (RE) and Normalised Root-Mean-Square Error (NRMSE). The former error measure is the absolute error divided by the magnitude of the exact value, often used to compare approximations of numbers of widely differing size. The latter measures the average error per unit of the accurate result size. They are defined by the following formulas:

$$RE = \frac{|e-a|}{a} \qquad RMSE = \sqrt{\sum_{i=1}^{n} \frac{(e_i - a_i)^2}{n}} \qquad \overline{a} = \sum_{i=1}^{n} \frac{a_i}{n} \qquad NMRSE = \frac{RMSE}{\overline{a}}$$

Where e is the estimated value returned by the estimation structure for a query, a is the correct, expected cardinality value for the query and n is the total number of queries selected for a document and tested against a given PS structure. The first formula defines RE and is quite straightforward. The remaining formulas compose the NMRSE measure, by introducing a normalising factor $\overline{a}$ to the Root-Mean-Square (RMSE) measure.

We have performed a number of empirical experiments whose comparative results are analysed. We compare PSUPS, LH, MT, and Xseed against a reference PS structure. For PSUPS structures, we implement both Equi-height and End-biased histograms, with and without Bitfields. For LH structures, only End-biased histograms were used. For MT structures, we adopt a pruning parameter of 2 and test different reduction strategies, such as suffix-* and no-*, using a budget of 30 entries for the former and 90 entries for the latter; we also test a version of MT without any reduction techniques.

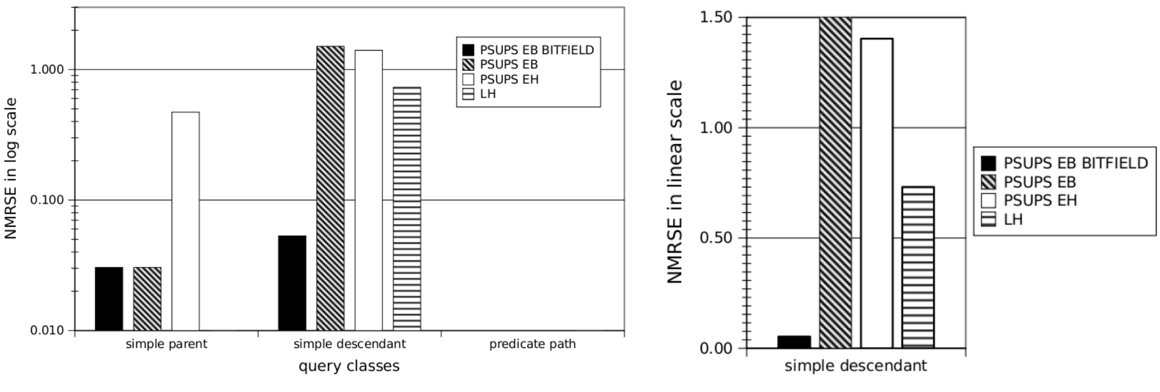
	PSUPS EB	PSUPS BitField	PSUPS EH	LH
simple parent	0.030	0.030	0.471	0.002
simple descendant	1.507	0.053	1.404	0.731
predicate path	0.000	0.000	0.000	0.004

Table 7.4: NMRSE measures for tree-based approaches in all query classes

In Table 7.4, we show the NMRSE values for all tree-based approaches, also shown in Figure 7.3(a), as a comparison between PSUPS using EB and EH histograms, and LH. PSUPS EB produces, in general, higher errors than EH ones. This is an interesting result. In fact, because of their definition, EB histograms should have the least approximation error. Figure 7.3(b) quantifies errors that occur when estimating descendant-axis queries, which is a manifestation of the false-positive problem with EB histograms mentioned in Section 6.3.1 : To perform estimation of `descendant` queries, one must look through the entire subtree of a given context, which likely implies hits in various histograms; some of those histograms may have the desired VocId, while others may not, introducing a false positive when the algorithm wrongly assumes the VocId is in the bucket of the average.

Figure 7.3(b) shows, in greater detail, that standard EB histograms are not suitable for SD query types, whereas for other query types (e.g. SP), it satisfies our expectations. As a consequence, EB histograms should be adapted to the characteristics of XML query processing. In the light of this observation, we defined the Bitfield, described in 6.3.1 as an optional measure, and tested it against all referenced documents using the same query workload, eliminating the false positive error in PSUPS, because we can now identify exactly whether or not a VocId is in an EB histogram. Clearly, this is a trade-off situation between storage space and accuracy. Therefore, we have to check whether or not the increase on size justifies the benefit of lowering the false-positive error. By analysing the (Bitfield) numbers in Table 7.4, one can see we obtained, by using PSUPS EB Bitfield, an average increase of only 7.4% on PSUPS sizes for a good increment in accuracy. Hence, with such a small increase, we reach very good estimation results.

Because PSUPS trees built with EH and EB histograms do not allow the derivation of good estimation results, we use only PSUPS trees with EB Bitfield histograms (PSUPS Bitfield) in further comparative benchmarks. The results in terms of RE and NMRSE measures are shown in Tables 7.5 and 7.6, respectively. A comparative view of that data is shown in Figures 7.4 and 7.5, where we compare PSUPS LDB and LH against MT and XSeed. Our proposals out-



(a) (b)

Figure 7.3: NMRSE measures for tree-based structures

	PSUPS	MT uncomp.	MT suffix-*	MT no-*	LH	Xseed
nasa	0.028	0.134	0.195	0.083	0.000	0.134
psd7003	0.005	0.000	0.808	0.000	0.000	0.069
dblp	0.003	0.018	15.692	0.014	0.142	0.361
swissprot	0.003	0.000	0.905	0.016	0.000	0.312
uniprot	0.003	0.198	0.198	0.198	0.020	0.011
treebank	0.388	0.098	0.096	0.098	0.000	7.247

Table 7.5: RE measures for all structure in all query classes

	PSUPS	MT uncomp.	MT suffix-*	MT no-*	LH	Xseed
nasa	0.00544	0.14534	0.19985	0.08016	0.00000	0.12341
psd7003	0.01510	0.00000	0.90241	0.00000	0.00000	0.13252
dblp	0.00013	0.00344	0.07079	0.00341	0.00184	0.02444
swissprot	0.01360	0.00001	0.97373	0.02327	0.00000	0.18097
uniprot	0.01022	0.44174	0.76688	0.46609	0.00000	0.00962
treebank	0.00003	0.09932	0.09694	0.09931	0.00000	1.90243

Table 7.6: NMRSE measures for all structure in all query classes

perform the others in almost all cases. XSeed does not behave well for non-recursive XML documents. For example, in `uniprot`, XSeed has an estimation error similar to MTSuf* (about 14% NRMSE error), which is twice as worse than that of LH (7%). MT approaches tend to underestimate queries with long paths, even if these queries are posed on non-recursive documents, as `swissprot` and `psd7003`.

Although the XSeed kernel is designed for highly recursive documents, it does not provide good estimates, compared to LH. The specific reason is the attempt to minimise false positives by using a pruned search method in the XSeed estimation algorithm which is controlled by a tuning parameter. Clearly, if this parameter is large, XSeed tends to give better results, obviously, at the cost of increased evaluation overhead. In the XSeed paper (ZHANG et al., 2006), the authors reported an NRMSE error of 169% (kernel) for only a 4MB part of a `treebank` document. We have summarised the XSeed estimation errors on an entire (86MB) `treebank` document and found that, applying the same pruned search, the error is even higher (see Figure 7.5). In contrast, LH has an error of <0,02% for the entire `treebank` document.

LH does not prune any tree path (as XSeed and MT do) and it presents, on average, a better space-accuracy trade-off for all documents than the competing approaches. Although still small enough for memory-resident use (see Table 3), one may argue that it is difficult to keep a large LH synopsis in memory. In this case, a possible strategy is to load parts of the structure on demand. For example, we may load LH level by level according to query optimiser use. Such a strategy would keep only the top-most levels in memory, say the first 6 levels, and would fetch nodes from deeper levels on demand. In fact, LH lends itself to evaluation based on such fragments. For treebank, only the 11 top-most levels (out of 37) heavily contribute to its size. Because most of the queries only encompass the top-most levels, optimisation heuristics could additionally be successfully applied.

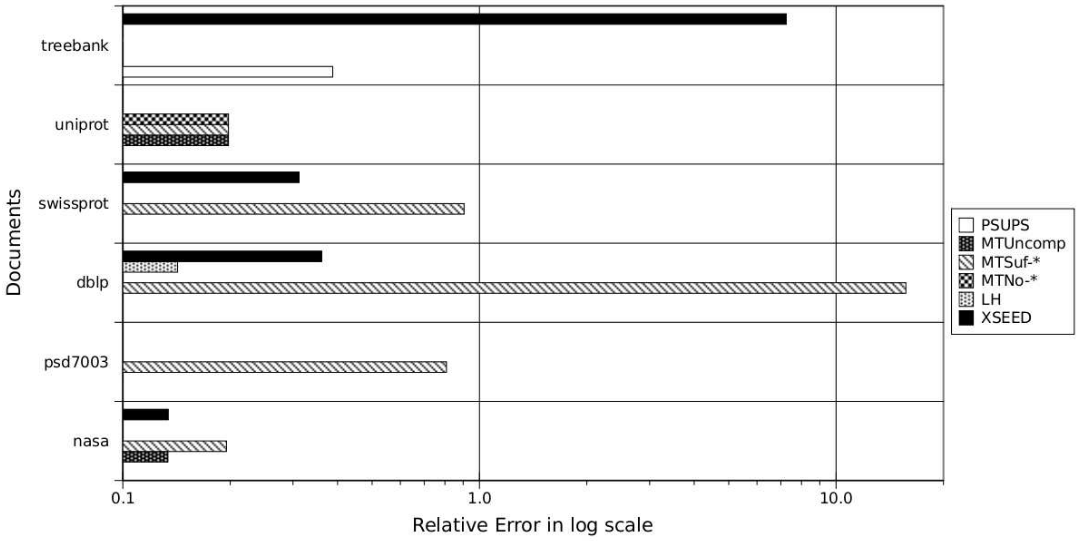


Figure 7.4: Relative Error measures for all documents

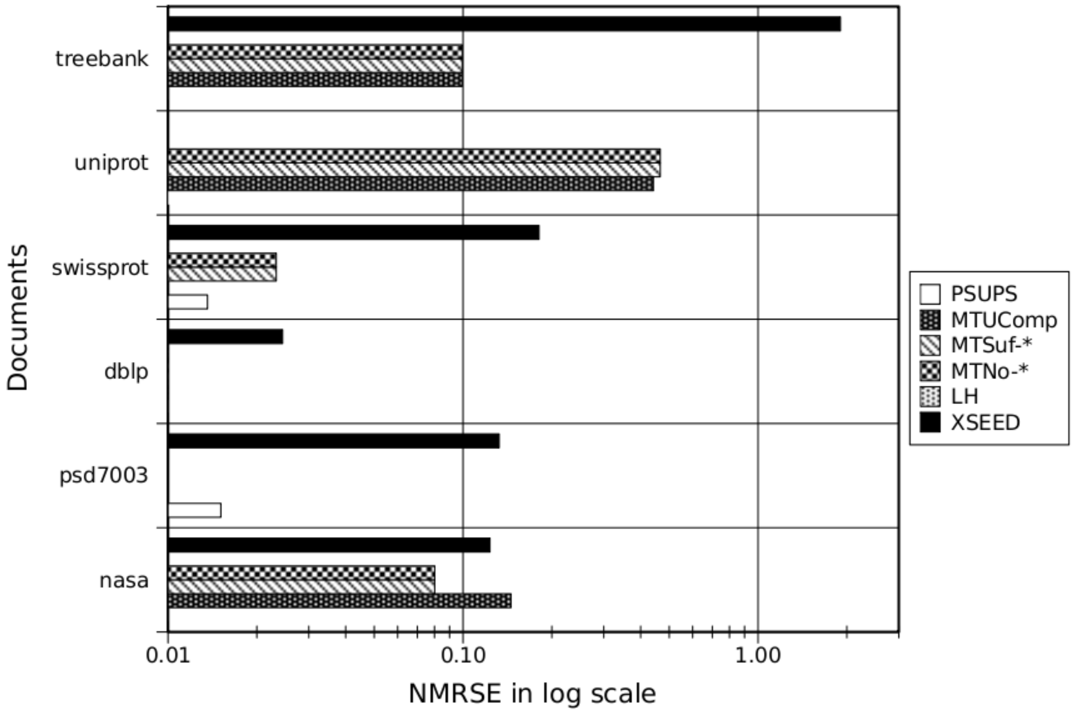


Figure 7.5: NMRSE measures for all documents

8 CONCLUSION

8.1 Summary

We studied various approaches for XML summarization and classified them using different criteria, such as base structure and completeness. We also studied histogram techniques, relevant for their use in PS structures like PSUPS and LH. We have studied four different PS structures, namely PSUPS, LH, Markov Table and Xseed, where the first two are our proposed solutions, and the other two are present in the literature (ABOULNAGA; ALAMELDEEN; NAUGHTON, 2001; ZHANG et al., 2006).

We compared, in theoretical terms, all those PS structures, further categoryzing them and analysing their behaviour, advantages, and drawbacks on a number of criteria. We showed implementation details for the implemented structures inside an estimation framework built as a service for the XTC XDBMS. We experimented with all studied structures, posing an query workload that, we believe, is unbiased, over a set of XML documents noticeable for its variety, consisting of documents of different shapes and sizes, flat and recursive, regular and irregular.

Regarding our two proposals, we found that PSUPS gives fairly accurate estimate, but storage space requirements can be impractical in highly-recursive documents like `treebank`, mostly because only a small proportion of the nodes will be considered for reduction with histograms. In face of this problem, we developed the LH structure, which aims to select nodes for histograms in a more aggressive manner, and effectively reduces storage space requirements when compared to PSUPS, nevertheless keeping estimations within acceptable error margins.

8.2 Future Work

The focus of this report was on estimation methods for structural features of XML documents. However, in real-world use scenarios, there is also a demand for estimation of the distribution of text values under structural patterns. None of the studied structures support such feature explicitly; however, our two proposals (PSUPS, LH) have potential to be extended for that purpose, by simply adding, in the leaf nodes, references to entries in auxiliary structures that actually store the information for text value distribution.

We also highlight the fact that LH and PSUPS support the main navigational axes of Xpath and Xquery expressions, and that they achieve reduction without pruning any part of the document or dismembering any of its path classes; all achieved without losing too much accuracy. Even though one could argue that LH is still too large for recursive documents, we observe that LH is a structure that can be loaded on-demand, on a level by level basis, depending on the design of the query optimiser algorithm that is using the structure.

In theory, PSUPS and LH could also be extended in the future to capture statistics in relational DBMS, because they naturally map parent-child relationships that are typical of foreign-key/primary-key in the relational world. This could be exploited by query optimisation algorithms in multi-join or other complex situations, where currently only statistical guesses are used.

REFERENCES

ABOULNAGA, A.; ALAMELDEEN, A. R.; NAUGHTON, J. F. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In: VLDB, 2001. **Anais...** [S.l.: s.n.], 2001. p.591–600.

AL-KHALIFA, S.; YU, C.; JAGADISH, H. V. Querying structured text in an XML database. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2003., 2003. **Proceedings...** ACM Press, 2003. p.4–15.

BROWNELL, D. **Sax2**. [S.l.]: O'Reilly, 2002.

HAUSTEIN, M.; HARDER, T. An efficient infrastructure for native transactional XML processing. **Data Knowl. Eng.**, Amsterdam, The Netherlands, v.61, n.3, p.500–523, 2007.

HäRDER, T.; HAUSTEIN, M.; MATHIS, C.; WAGNER, M. Node labeling schemes for dynamic XML documents reconsidered. **Data Knowledge Engineering**, Amsterdam, The Netherlands, v.60, n.1, p.126–149, 2007.

HäRDER, T.; REUTER, A. Concepts for Implementing a Centralized Database Management System. In: INT. COMP. SYMPOSIUM ON APPLICATION SYSTEMS DEVELOPMENT, 1983. **Anais...** [S.l.: s.n.], 1983. p.28–60.

IOANNIDIS, Y. E. The History of Histograms (abridged). In: VLDB, 2003. **Anais...** [S.l.: s.n.], 2003. p.19–30.

IOANNIDIS, Y. E.; POOSALA, V. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In: SIGMOD CONFERENCE, 1995. **Anais...** [S.l.: s.n.], 1995. p.233–244.

JAGADISH, H. V.; KOUDAS, N.; SEVCIK, K. C. **Choosing Bucket Boundaries for Histograms**. [S.l.]: University of Toronto, 1998.

PIATETSKY-SHAPIRO, G.; CONNELL, C. Accurate Estimation of the Number of Tuples Satisfying a Condition. In: SIGMOD'84, PROCEEDINGS OF ANNUAL MEETING, BOSTON, MASSACHUSETTS, JUNE 18-21, 1984, 1984. **Anais...** ACM Press, 1984. p.256–276.

POLYZOTIS, N.; GAROFALAKIS, M. N. XSKETCH synopses for XML data graphs. **ACM Trans. Database Syst.**, [S.l.], v.31, n.3, p.1014–1063, 2006.

POOSALA, V.; IOANNIDIS, Y. E.; HAAS, P. J.; SHEKITA, E. J. Improved Histograms for Selectivity Estimation of Range Predicates. In: SIGMOD CONFERENCE, 1996. **Anais...** [S.l.: s.n.], 1996. p.294–305.

WANG, W.; JIANG, H.; LU, H.; YU, J. X. Bloom Histogram: path selectivity estimation for xml data with updates. In: VLDB, 2004. **Anais. . .** [S.l.: s.n.], 2004. p.240–251.

XPATH XML Path Language 2.0. , [S.l.], November 2005.

XQuery 1.0: an xml query language. , [S.l.], November 2005.

ZHANG, N.; ÖZSU, M. T.; ABOULNAGA, A.; ILYAS, I. F. XSEED: accurate and fast cardinality estimation for xpath queries. In: ICDE, 2006. **Anais. . .** [S.l.: s.n.], 2006. p.61.