# On-line Methods for Database Optimization

by

Lipyeow Lim

Department of Computer Science
Duke University

Date:____________________
Approved:

____________________
Dr. Jeffrey Scott Vitter, Supervisor

____________________
Dr. Min Wang

____________________
Dr. Ron Parr

____________________
Dr. Jun Yang

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2004

ABSTRACT

(Computer Science)

# On-line Methods for Database Optimization

by

Lipyeow Lim

Department of Computer Science
Duke University

Date:________________________
Approved:

________________________
Dr. Jeffrey Scott Vitter, Supervisor

________________________
Dr. Min Wang

________________________
Dr. Ron Parr

________________________
Dr. Jun Yang

An abstract of a dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2004

# Abstract

Databases and information retrieval systems storing data from the World Wide Web require new optimization strategies to deal with the different types of data (textual, semi-structured and relational), the massiveness of the data, and the dynamic nature of the data. This dissertation investigates on-line methods for the optimization of databases storing web data. On-line methods avoid access to the underlying data, avoid the rebuilding of data structures from scratch, and adapt to changes in the data and query workload characteristics. On-line techniques are therefore especially suited to web data.

In this dissertation, we present on-line techniques for two general problems: how to update inverted indexes and how to estimate the selectivity or result size of queries in database systems. For the index update problem, we present the landmark-diff method for updating the inverted index in response to changes in previously indexed documents. The landmark-diff method allows indexes to be updated incrementally without a complete rebuild.

For selectivity estimation, we propose three on-line techniques that address the selectivity estimation problem in three different context. In the context of relational databases, we present SASH, a Self-Adapting Set of Histograms, that uses probabilistic graphical models to address the following issues in an on-line manner: which sets of attributes to build histograms on, how to build these histograms without looking at data, and how much memory should be allocated to these histograms.

For XML native databases, we present XPathLearner, an on-line workload-aware method for estimating the result size of given XPath queries. XPathLearner learns the path statistics from query feedback (past query answers) in an on-line manner, and adapts itself to changes in the data and the query workload characteristics.

In a more general context, we present CXHist, an on-line classification based histogram for estimating the selectivity of a broad class of queries. CXHist models queries instead of data and stores the mapping between queries and their selectivity using a naive Bayesian classifier. We show that CXHist is very accurate in estimating the selectivity of exact match and substring predicates on leaf values reachable by a given XPath in XML databases.

# Acknowledgements

I would like to thank my advisor, Jeffrey Scott Vitter, for his guidance through my doctoral research. Jeff has zero tolerance for fuzzy or sloppy thinking and working with Jeff has forced me to sharpen my thinking and elucidate my ideas in concrete terms. Much credit also goes to my collaborator Min Wang, who has often played the role of a co-advisor.

I thank my committee members, Min Wang, Ronald Parr, and Jun Yang, for their criticisms, comments, and suggestions on my doctoral research work.

Much of the research in this dissertation started as summer internship projects at IBM T. J. Watson Research Center and I would like to thank my mentor Min Wang and my manager Sriram Padmanabhan for giving me the opportunity to intern with them. I also thank Ramesh Agarwal for getting me started on the inverted index update problem.

My life as a graduate student at Duke has been enriched by many friends. I thank my friends in the Computer Science department for their companionship in my studies: Marty Gilbert, Andy Danner, Sara Sprenkle, Vijay Natarajan, Prachi Thakar, Priya Mahadevan and Apostol Natsev to name a few. I thank Scott and Jenny Hawkins, the Inter-cultural Christian Fellowship, Steve and Christi Hinkle, and the Graduate Inter-Varsity Christian Fellowship for their fellowship, love and support. I thank my friends from the taijiquan community for keeping me sane by exercising with me.

No words can express my appreciation and gratitude for my wife, Puiwai Bun, who forsook her career in Singapore to look after me these last two years of my doctoral studies.

I thank my parents and my brother for their encouragement and love.

I would not have embarked on my Ph.D. studies if my undergraduate advisor, Dr. Y. C. Tay, had not planted the idea in my head. I also thank Master's advisor, Dr. Philip M. Long for nudging me along.

I gratefully acknowledge the financial support given by IBM through an IBM PhD Fellowship and by the Army Research Office through grant DAAD19–03–1–0321.

Last but not least I thank my Lord and Savior, Jesus Christ, Creator and Sustainer of all things.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The World Wide Web has revolutionized information storage and retrieval. Data from the web is dynamic, massive, and heterogeneous, posing unique challenges to storage and retrieval techniques for traditional data types (such as relational tables and text documents). Web data are often a mixture of relational (structured), multi-media, textual, and semi-structured data. In particular, semi-structured data as exemplified by documents in the extensible mark-up language (XML) [8] cannot be neatly classified into relational data or text documents in the traditional sense and hence require new optimization strategies in database and information retrieval systems. The massiveness and the dynamic nature of web data also render optimization techniques based on off-line scans of the data infeasible if not impossible. Methods that are incremental and do not require costly off-line data scans are needed.

This dissertation investigates on-line methods for two general problems arising from the optimization of databases and information retrieval systems storing web data. On-line methods are algorithms that do not require costly scans of underlying data, and do not rebuild data structures from scratch. On-line methods often have the characteristics of being able to adapt to changes or updates in the underlying data, and being able to tune themselves to workload characteristics as well.

The two general problems studied in this dissertation are the inverted index update problem and the selectivity estimation problem. The inverted index update problem pertains to databases and information retrieval systems storing textual data from the web. A search engine is one such system. It constructs an inverted index for web documents in order to support keyword queries. Web documents change over time at different rates and the inverted index needs to be incrementally updated so that updated documents are searchable. Previous work has addressed inverted index updates in response to the addition and deletion of documents. We present the landmark-diff method that addresses the complementary problem of inverted index updates in response to changes in previously indexed documents. The landmark-diff method is a general approach applicable to many update scenarios including off-line partial index rebuilds, but it is especially suited to on-line, incremental updates.

The selectivity estimation problem pertains to databases storing structured and semi-structured (web) data. The selectivity of a query refers to the size of the query result. Cost-based optimizers use selectivity estimates to evaluate the cost of query execution plans during query optimization. Obtaining a quick estimate of the selectivity of query without actually executing the query is therefore an important problem in database systems. We propose on-line selectivity estimation methods for three particular selectivity estimation problems.

The first selectivity estimation problem is building a set of multidimensional histograms for estimating the selectivity of multi-attribute queries on relational data. Previous work has focused mainly on off-line methods for building single multidimensional histograms. We investigate on-line methods that address the problem of which sets of attributes to build multidimensional histograms on, how to build these histograms without looking at data, and how much memory or storage should be allocated to these histograms. We propose SASH, a Self-Adapting Set of Histograms, that uses probabilistic graphical models to address these issues. SASH differs from related work in that it uses only query feedback information to decide which sets of attributes to build histograms on, to learn the statistics for building the set of histograms, and to allocate memory to each histogram.

The second selectivity estimation problem we address is estimating the selectivity of XPath [22] queries. XPath is a navigational scheme for tree-like XML documents similar to path names in the directory structure of file systems. Selectivity estimation of XPath queries involves estimating the number of nodes in the XML tree that are reachable via the given XPath. We propose XPathLearner, an on-line method for estimating the selectivity of XPath query expressions. All previous work require an off-line scan of the underlying XML data tree in order to obtain the statistics required for selectivity estimation. XPathLearner models path expressions as Markov chains, and learns the associated statistics from query feedback (past query answers) in an on-line fashion. XPathLearner does not require an off-line scan of the XML data, and tunes itself to changes in the data and in the query workload characteristics.

The third selectivity estimation problem we address is building an on-line histogram for estimating the selectivity of a broad class of queries on possibly diverse data types. We propose CXHist, an on-line histogram that is capable of storing statistics of different types of queries on diverse data types in a single unified framework. CXHist models queries instead of data and captures the mapping between queries and their selectivity using a Bayesian classifier.

In the next two sections, we describe the problem of updating inverted indexes and the problem of estimating selectivity in greater detail.

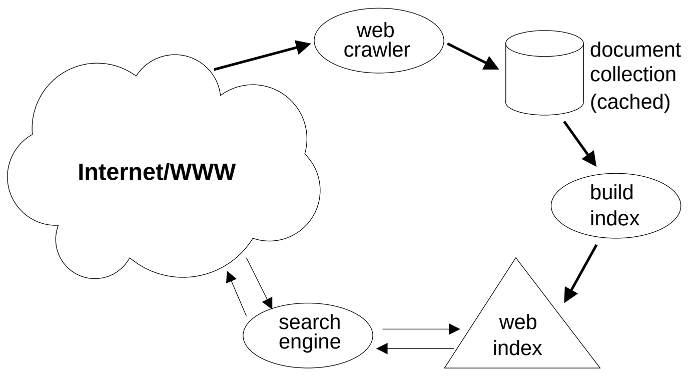


**Figure 1.1**: Schematic of a simple search engine system.

## 1.1 The Inverted Index Update Problem

In this section, we describe how a search engine for web documents works and introduce the inverted index update problem.

Most of the information on the World Wide Web consists of web pages. Web pages are essentially textual information formatted in HTML or XML; hence, we will refer to a web page as a *web document*. To find web documents on a particular subject, users submit a keyword query to a search engine. The search engine processes the query and returns an ordered list of uniform resource locators (URLs) [65] to web documents containing the query keyword(s) ranked by some criterion such as relevance to the given query.

Search engines maintain keyword indexes in order to answer keyword queries efficiently. The indexing method of choice is the *inverted index* (also known as an *inverted file*). Inverted indexes are similar to the indexes at the end of a book: they store the locations (URLs and position in the document) where each keyword can be found. A search engine system periodically downloads a collection of web documents in a process called *web crawling*, and constructs an inverted index for the downloaded documents (see Figure 1.1).

Before the advent of the Internet, inverted indexes have been mainly used for keyword search over a collection of *static* documents, i.e., documents whose contents do not change over time. Web documents, however, differ from static documents in that their contents may change: web authors may edit the contents of their websites, and some web documents have content that is inherently time sensitive. How do we ensure that newly updated documents are searchable? Updated documents need to be detected, downloaded and indexed. Techniques to detect and download updated documents already exist [9, 19, 20, 52]; however, efficient methods to update the inverted index to incorporate information from the updated documents are still lacking.

Chapter 2 of this dissertation addresses the problem of how to update inverted indexes efficiently and incrementally without the need to rebuild the index from scratch. We present the landmark-diff method that addresses the problem of updating inverted index in response to changes in previously indexed documents. We show that the landmark-diff method can be used in combination with existing techniques for efficient, on-line, incremental index updates.

## 1.2 The Selectivity Estimation Problem

In this section, we introduce the selectivity estimation problem and its role in database query optimization. More in-depth description can be found in several excellent survey articles [14, 38].

The *selectivity* of a query is defined to be the size of the query result. In relational databases, the selectivity of a query or query operator is traditionally specified as a fraction of the input data size; however, in keeping with the use of the term in XML databases (where data do not come in neat units of tuples), the more general

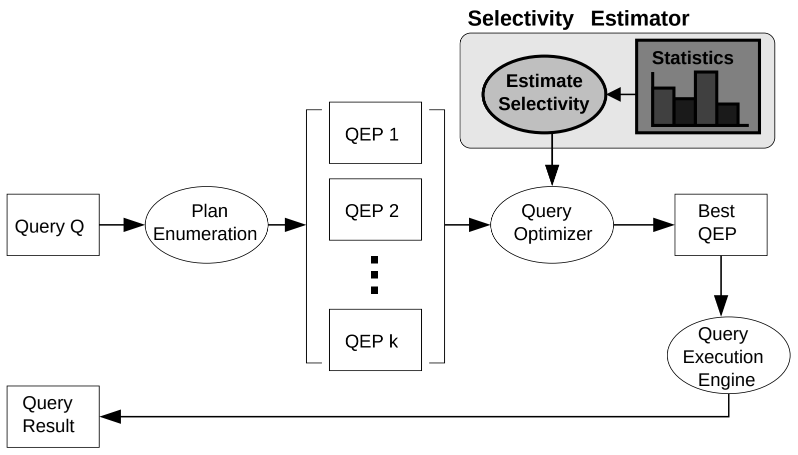


**Figure 1.2**: Schematic of the query optimization process in database systems.

definition of selectivity in terms of query result size will be used throughout this dissertation.

The process of approximating the result size (selectivity) of a given query without actually executing the query is called *selectivity estimation*. Accurate approximations to query result size are more than crucial to query optimization in database systems (see Figure 1.2). When a database system processes a query, it first translates the query into a set of equivalent query execution plans (QEPs) in a process called *plan enumeration*. Each QEP will yield the correct query result when executed by the query execution engine and differ in its execution time. In cost-based query optimization, the query optimizer will approximate the cost of each QEP using a *cost model* and choose the least cost plan to be executed by the query execution engine. QEPs often resemble operator trees where the leaves are relations or indexes. To approximate the cost of a QEP, the cost model requires as input the QEP itself, system parameters like buffer pool usage etc., and most importantly, selectivity estimates of the operators in the QEP. The accuracy of the selectivity estimates therefore has a direct impact on the choice of the QEP. The following example illustrates how selectivity estimates affect the choice of a QEP.

**Example 1.1** *Consider the following Product(PID, Price, Sales) relation or table in a relational database,*

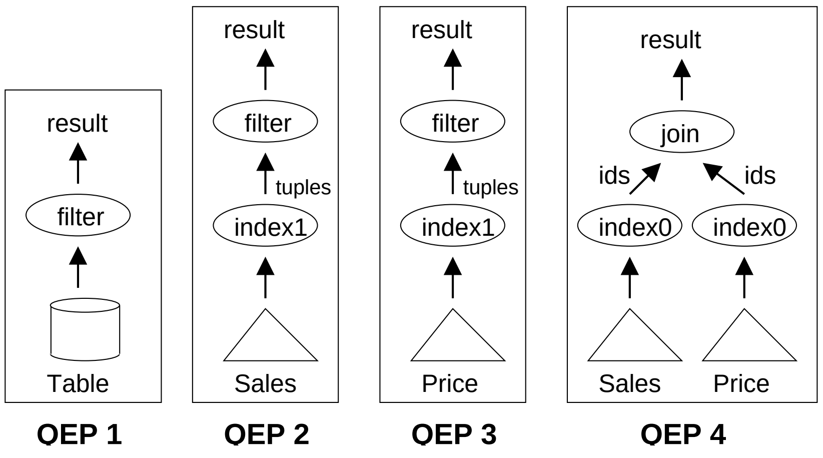
*Product*

| *PID* | *Price* | *Sales* |
|---|---|---|
| *1* | *10.50* | *120* |
| *2* | *13.00* | *1000* |
| *3* | *30.25* | *220* |
| *4* | *3.35* | *2020* |

*and the following two queries that are similar in structure and differ only in the range constraint they specify:*

| *Query 1:* | *Query 2:* |
|---|---|
| *SELECT ** | *SELECT ** |
| *FROM Product* | *FROM Product* |
| *WHERE 100 < Sales < 500* | *WHERE 100 < Sales < 2000* |
| *AND 10 < Price < 20* | *AND 10 < Price < 50* |

*Consider Query 1. Assuming that indexes have been created on both attribute Price and Sales, the query will first be rewritten into a set of query execution plans (QEPs):*



*Note that Query 2 will be rewritten into the same set of QEPs, because of its similarity in structure to Query 1. The cost of each QEP will depend (among other parameters) on the selectivity of the query itself and of the different operators. The selectivity of Query 1 is 1 since only the tuple with PID 1 satisfy both range constraints. The cost of QEP 1 is almost certainly higher than the cost of the other index-based QEPs. In the case of Query 2, the selectivity of the query is 3, that is, almost the entire relation will be returned as the query result. Hence, QEP 1 (table scan) is most likely the least cost plan.*

Selectivity estimation is usually done using pre-computed statistics that reside in a small amount of pre-allocated memory. The statistics need to fit into a small amount of memory primarily because the selectivity estimates need to be computed very efficiently. The query optimizer needs to evaluate the cost of many QEPs and for each QEP, the optimizer needs to compute potentially as many selectivity estimates as there are operators in the QEP. If the selectivity estimator needs to scan through a large amount of memory in order to compute a single estimate, the processing time for query optimization will be unacceptable. Hence, statistics for selectivity estimation need to fit into a small amount of memory.

The selectivity estimation problem is: Given a fixed amount of memory to store statistics, estimate the selectivity of a given set of queries using the stored statistics only. Selectivity estimation methods differ in

1. what kind of queries are supported,
2. what kind of statistics are used,
3. how the statistics are stored,
4. how the selectivity estimates are computed from the statistics,

5. how the statistics are collected (and updated).

The type of queries supported is closely related to the characteristics of the data stored in the database. For example, if a column of a table stores string data (`varchar` type), then the database may need to support substring queries and selectivity estimates of substring queries may be needed. In relational databases, the more conventional query types are point queries, range queries and join queries. These queries may involve more than one attribute, in which case, the correlation between different attributes may need to be captured in the statistics.

Closely related to the type of queries supported is the type of statistics collected. The two most popular types of statistics are random samples and frequency statistics in the form of histograms. Histogram-based statistics can be further subdivided into various categories [39]. The type of queries often determine the type of statistics. For example, substring queries will require substring statistics and join queries will require correlation across relations to be captured in the statistics.

For the required statistics to fit into the given amount of memory, further aggregation, pruning or compression may be needed. For example, a random sample may have irrelevant columns removed, duplicated counted, and stored as tuple-count pairs. Histograms can sometimes be compressed in a lossy fashion using transforms like the discrete wavelet transform [59].

Whatever statistics are collected, compressed and stored, the selectivity estimator needs to be able to compute a selectivity estimate from the stored statistics. The exact algorithm for computing these selectivity estimates depends on the type of statistics and how they are stored. If a wavelet transform has been applied on some histogram-based statistics, the inverse transform needs to be performed before the statistics can be recovered. If some portion of the statistics has been discarded or pruned, error correction strategies need to be used.

How are statistics collected? Traditionally, statistics are collected in an off-line process (eg. `runstats` in IBM DB2) often initiated by the database administrator. A scan is made through the database to collect the relevant statistics. An off-line database scan can be very time-consuming if the database is large. In federated databases, database scans on some remote databases may even be impossible. On-line statistics collection is an alternative statistics collection approach that learns relevant statistics by looking at past query answers, also known as *query feedback*. After the query execution engine computes the answer to a query, the size of the answer is fed back to the selectivity estimation system, so that the stored statistics can be updated. A small overhead is incurred per query feedback in this update, but the benefit of on-line statistics collection is that the statistics adapt to updates in the underlying data and to query workload characteristics. The on-line selectivity estimation methods that we propose in this dissertation all rely on on-line statistics collection to avoid costly database scans.

Three particular selectivity estimation problems are addressed in this dissertation. Chapter 3 addresses the problem of building and tuning a *set* of multidimensional histograms in relational databases using only query feedback information. Building a set of histograms entails two additional challenges: how to decide which sets of attributes to build the multidimensional histograms on, and how to distribute a fixed amount of memory among the histograms. We present a Self-Adapting Set of Histograms (SASH) that addresses these challenges in an on-line manner.

Chapter 4 addresses the problem of estimating the selectivity of XML path expressions in XML databases. Path expressions are similar to path names in the directory structure of a file system and are used for locating sets of nodes in an

XML data tree. We propose XPathLearner [55], an on-line method for estimating the selectivity of XPath query expressions. All previous work require an off-line scan of the underlying XML data tree in order to obtain statistics.

Chapter 5 addresses the problem of building an on-line histogram for diverse types of data and queries. Previous histogram techniques have been restricted to single data model and query type. We present CXHist, a unifying histogram framework that builds histograms for diverse data and query types using query feedback information only. In particular we apply CXHist to the selectivity estimation of exact match and substring queries on the leaf nodes reachable by a given path in XML data.

# Chapter 2

# The Landmark-diff Method for Updating Inverted Indexes

## 2.1 Introduction

Textual data form a large fraction of the information on the World Wide Web. Web pages or documents are often textual information marked-up with HTML or XML. Information retrieval over these unstructured (or semi-structured) data is achieved mainly by using keyword indexes maintained by search engines. Search engines typically download web documents in a process called *web crawling*, and constructs a keyword index for the downloaded documents (see Figure 1.1).

The inverted index is the indexing technique of choice for such unstructured documents and is sometimes used for semi-structured documents as well. Search engines use inverted indexes to provide keyword search over web documents [64], and DBMSs use it to support containment queries in XML documents [54, 88].

Inverted indexes have been traditionally used for keyword search over a collection of *static* documents, i.e., documents whose contents do not change over time. Web documents, however, differ from static documents in that their contents may change over time. How do we ensure that newly updated documents are searchable? Specifically, how do we keep inverted indexes up-to-date? This is the question we investigate in this chapter.

An *inverted index* is a collection of inverted lists, where each list is associated with a particular word. An *inverted list* for a given word is a collection of document IDs of those documents that contain the word. If the position of a word occurrence in a document is needed, each document ID entry in the inverted list also contains a list of location IDs. Positional information of words is needed for proximity queries and query result ranking [64]. Omitting positional information in the inverted index is therefore a serious limitation. Positional information is usually stored in the form of location IDs. The location ID of a word is the position in the document where the word occurs. An entry in an inverted file is also called a *posting*; it encodes the information $\langle word\_id, doc\_id, loc\_id \rangle$.

A *crawler* is a program that collects web documents to be indexed. Cho and Garcia-Molina [19] have shown that an in-place, incremental crawler can improve the freshness of the document collection. However, the index rebuild method commonly used for updating the inverted index cannot take advantage of an incremental crawler, because the updated documents crawled in between rebuilds are not searchable until the next index rebuild. Moreover, complete index rebuild at high frequency is prohibitively expensive. An efficient and, ideally, incremental technique for keeping inverted indexes up-to-date is crucial in making the most recently crawled web documents searchable. Previous work on incremental inverted index updates have been restricted to adding and removing documents. Updating the inverted index for previously indexed documents that have changed has not been addressed.

In this chapter, we propose the landmark-diff update method, an efficient method to incrementally update the inverted index for previously indexed documents whose contents have changed. We show that solving this particular problem results in very efficient solutions to the more general index update problem. Our method uses the idea of landmarks together with the `diff` algorithm to significantly reduce the number of postings in the inverted index that need to be updated. Our experiments validate the effectiveness of our landmark-diff method.

The rest of this chapter is organized as follows. We describe common approaches to the index update problem and their relationships to each other in Section 2.2. Related work is then discussed in Section 2.3. As motivation for the landmark-diff method, we present an empirical analysis of web document change in Section 2.4. Section 2.5 presents the landmark-diff method followed by an analytical evaluation in Section 2.6, and an empirical evaluation in Section 2.7. Section 2.8 summarizes and concludes this chapter.

## 2.2 A Taxonomy of Index Update Approaches

We describe a concise taxonomy of index update approaches that will be helpful in classifying and understanding the different update methods (see Figure 2.1). Our taxonomy has three general approaches for keeping an index up-to-date: index rebuild, lazy update, and incremental update. For incremental update methods, we analyze the update methods by how they handle three essential update tasks: deleting old documents, inserting new documents and updating common documents. We show that our landmark-diff method is a method for updating common documents, and also describe existing methods that can be used to update the index for common documents that have changed.

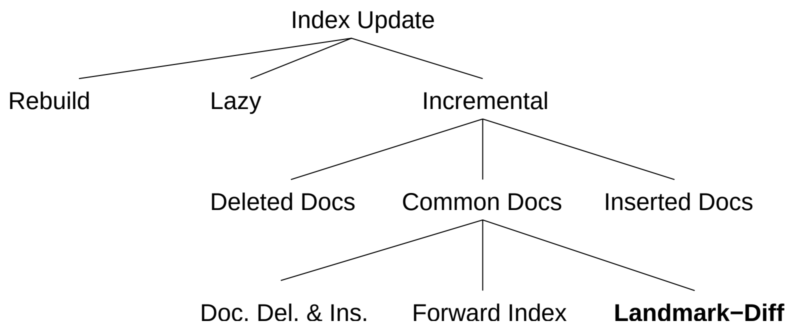


**Figure 2.1**: Taxonomy of index update approaches.

### 2.2.1 Index Rebuild.

The most straightforward way of updating an index is to discard the old index and build a new one using the updated document collection (consult the book *Managing Gigabytes* [83] for details). The disadvantages are: (1) the entire updated document collection has to be available to the index construction algorithm at the time of index rebuild, and (2) every word in the collection has to be scanned to construct the inverted index. Distributed rebuilding techniques, such as the one by Melnik et al. [62], parallelizes (and pipelines) the process, but does not eliminate the need of scanning every word in every document.

Index rebuild is recommended only if (1) there are very few common documents or (2) the set of common word occurrences is small, i.e., a large portion of each updated document has changed. If the magnitude of change is small, scanning and re-indexing the words in documents that did not change is wasteful and unnecessary. Naturally, if document contents change more frequently, the index will need to be rebuilt more frequently. As the interval between rebuilds gets smaller, the magnitude of change between the two snapshots of the indexed collection also becomes smaller [19]. A large portion of the inverted index will remain unchanged, and a large portion of the work done by the rebuild is redundant.

### 2.2.2 Lazy Update.

The second approach is to be 'lazy' in updating the index. Instead of updating the index whenever some subset of documents has been updated, the updates or changes are stored in a searchable *update log*. Real index updates (or rebuild) is performed only after a certain period of time or when the update log reaches a certain size. This approach is similar to the 'stop-press' technique [83] used to store the postings of documents that need to be inserted into the indexed collection. Each entry in this update log is a delete or insert posting operation.

The advantages of lazy updates are that (batches of) newly-crawled updated documents can be made searchable and that real index updates are deferred and can be batched. The implicit assumption is that the update log can be maintained very efficiently. The disadvantages of lazy updates are its negative impact on query processing and the potentially large size of the update log. Query processing time will increase, because both the inverted index and the update log need to be searched, and the results of both merged. The size of the update log can become prohibitively large especially if positional information is stored in each posting. The landmark-diff method that we propose can be applied to potentially reduce the size of the update log.

### 2.2.3 Incremental Update.

Frequent rebuild is inefficient because it rebuilds portions of the index that did not change. The update log is unsatisfactory because it may become too large and affect query response time. Incremental updates is another way of updating the index that neither rebuilds the index from scratch nor does it use an update log. An incremental update method has to handle three types of changes:

1. documents that no longer exist (henceforth “deleted documents”);
2. new documents (henceforth “inserted documents”);
3. common documents that have changed.

Compared to index rebuild, an incremental update method has to deal with deleted documents, but does not need to re-index common documents that did not change. Compared to lazy update, it does not have the query processing overhead of an update log.

Incremental updates are especially important if incremental crawling is used in the search engine system. Incremental crawling [9, 19, 20, 52] crawls different documents at different frequency depending on the documents’ frequency of change. (Optimal crawling frequency has also been addressed in [9, 20, 52]). Ideally an incremental update method would be used to update the index, so that updated documents could be made searchable as soon as they are crawled.

Previous work on incremental inverted index updates has addressed changes due to inserted documents [11, 34, 77] and deleted documents [23, 24, 77]. Index updates due to common documents that have changed have not been addressed. The landmark-diff method that we propose in this chapter is an efficient update method that deals with the common documents. We will preview the landmark-diff method after briefly describing two naive update methods (the document delete and insert method, and the forward index method) for the common documents that have changed. The forward index method is used as a benchmark in our experiments because it is the most efficient of the two naive methods.

### Document Delete and Insert

A simple way of dealing with changes in the set of common documents is to delete all the postings for the old version of the documents in the inverted index and insert the postings of the updated documents. With respect to the set of common documents, the document delete and insert method differs from index rebuild in that only documents that have changed are processed, whereas index rebuild re-indexes all documents whether they have changed or not.

For each document that has changed, the worst case number of postings deleted and inserted is $O(m + n)$, where $m$ and $n$ are the number of words in the old and new version of the document, respectively. The advantage of this method is that (1) query processing is not affected, and (2) web documents can be crawled at different sampling intervals depending on their rate of change. The document delete and insert method is recommended if the magnitude of change within an updated document is large.

### Forward Index Update

A forward index is a list of postings associated with a single document (see Figure 2.2) and differs from the inverted index in that the list of postings of the form $\langle word\_id, doc\_id, loc\_id\rangle$ is sorted first by $doc\_id$, then by $word\_id$ and $loc\_id$. The forward index data structure was introduced in Page and Brin's Google paper [64] as an auxiliary data structure to speedup inverted index construction. We deduce that it can be used for updating the index for common documents as described next. Given the forward indexes of the old and new document, we generate a list of postings operations (insert posting and delete posting) that will transform the forward index of the old document to that of the new document. Since the postings

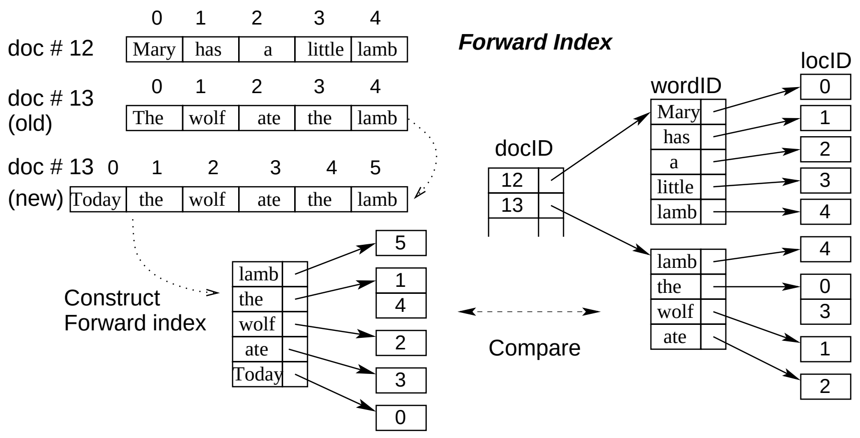


**Figure 2.2**: An example of a forward index. For the new version of the document #13 (with "Today" inserted to the beginning), a forward index representation is constructed. This forward index is then compared with the forward index of the old version stored by the system. Note how the location IDs of the words changes.

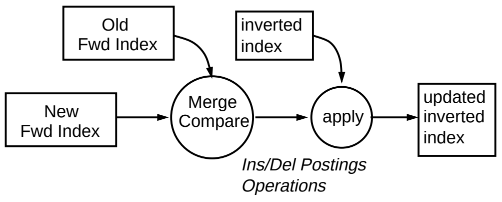


**Figure 2.3**: Updating an inverted index using forward indexes.

stored in a forward index are the same postings that are stored in the inverted index, we can use these posting operations to update the inverted index. The data flow diagram in Figure 2.3 illustrates the update procedure for a single document. To update the inverted index, we apply the procedure for every common document that has changed.

The advantages of using the forward index method are: (1) as with the document delete and insert method, incremental crawling can be used, (2) when the content change occurs near the end of the document, the postings corresponding to the words before the position of that change need not be deleted and re-inserted, (3) query processing is not affected. However, the disadvantages are: (1) an additional forward index needs to be stored for each document, (2) each forward index requires as much space as the document itself, and (3) the number of postings deleted and inserted for a document that has changed is still $\Theta(m+n)$ in the worst case, even if the difference between the two documents is small. For a pathological example, consider an insertion of one word to the beginning of the old document. All subsequent *loc_id*s will shift by one and hence all corresponding postings in the index will need to be updated.

### The Landmark-diff Method

The landmark-diff update method that we propose uses an encoding of positional information relative to landmarks in a document that specifically tackles the problem of *loc_id*s shifting due to small updates. Positional information is stored as a landmark-offset pair and the position of a landmark in a document is stored in a separate landmark directory for each document. Combining the use of landmark encoding and the `diff` of the old and updated documents, the landmark-diff method is able to reduce the number of inverted index update operations by leveraging on the "shift-invariant" property of the postings encoded using landmarks. For example, without landmarks, the insertion of just one word at the beginning of a document will shift the location IDs of all subsequent words by one. With landmarks, only the postings corresponding the text between the two landmarks straddling the edit need to change.

The advantages of the landmark-diff method are: (1) incremental crawling can be used, and (2) it is especially efficient for small and clustered changes in documents. The disadvantage is that processing of queries requiring positional information may require landmark directory lookups; however, the additional processing time is likely to be minimal compared to lazy updates, because the size of a landmark directory is significantly smaller than an update log.

## 2.3 Relation with Prior Work

Information retrieval using inverted indexes is a well studied field [4, 83]. Although most of the update techniques generalize to keyword searches in web documents (HTML and XML files), several assumptions made by those techniques need to be re-examined. First, web documents are more dynamic than the text document collections assumed by those techniques. Past work on incremental updates of inverted indexes [4, 11, 25, 77, 83] deals mostly with additions of new documents to a collection of static documents. Static documents are existing documents in the collection whose contents do not change over time. The problem addressed in this chapter deals with existing documents whose contents do change over time.

Second, previous work assumes that the inverted index resides on disk or in some persistent storage and try to optimize the propagation of in-memory update posting operations to the inverted file in persistent storage [11, 23, 24, 77]. Current search engines keep a large portion of the inverted index in memory[1], especially the most frequently queried portions of the inverted index. Even though the indexed collection is growing, we have three reasons to believe that a large portion of the inverted index will still be kept in memory:

1. query volume is increasing and there is a demand for increasing query processing speed;
2. the cost of memory chips is relatively low and still decreasing;

[1]For example, Google has thousands of machines with at least 8 GB of memory each.

3. the ease of scaling with parallel architecture.

Not only does the inverted index in persistent storage need to be updated, but the portion of the inverted index in memory needs to be updated as well. The update method we propose in this chapter addresses the problem of updating the inverted index as a data structure independent of whether it resides in memory or on disk. In many cases, our method could be used in conjunction with existing techniques to speedup the propagation of updates to the off-memory inverted index. Several of these existing techniques will be described in greater detail next.

For append-only inverted index updates, Tomasic et al. [77] proposed a dual inverted list data structure: the in-memory short list and the disk-based long list. New postings are appended to their corresponding short lists and the longest short list is migrated into a long list when the storage area for the short lists is full. Brown et al. [11] proposed another incremental append strategy that uses overflow ‘buckets’ to handle the new postings in the relevant inverted lists. These overflow buckets are chained together and have sizes that are powers of 2. Both [77] and [11] deal with append-only incremental updates and assume that a document never changes once indexed. Our method tackles the complementary problem of updating previously indexed documents that have changed.

Clarke et al. [23, 24] addressed the deficiency of these append-only inverted index techniques and proposed a block-based processing of the inverted index that supports deletion of documents as well. The entire document collection is concatenated into a long sequence of words. Positional information of a posting is reckoned from the beginning of the document collection (as opposed to the beginning of each document). Hence, even though Clarke's inverted index supports updates at the postings level, it does not solve the problem of small changes in documents causing a shift in the positional information of many postings unrelated to the change. In fact, the positional shift problem is exacerbated by the use of absolute positional information reckoned from the beginning of the collection.

Our landmark encoding scheme is based on the idea of *blocking*. The idea of partitioning a document or document collection into blocks has been used for various purposes in [5, 46, 58, 71, 91]. The Glimpse [58] system uses a block addressing inverted index in order to keep the index size small. A block addressing inverted index stores, for each keyword, pointers to the blocks of text containing the keyword. The actual occurrence of the keyword in each block is not stored and is found by scanning the actual text itself using a fast sequential scan algorithm, such as Knuth-Morris-Pratt [48] or Boyer-Moore [7]. In [46, 71, 91], the idea of block addressing is used to improve the quality or the relevance of the information retrieved for a given query. In our work, landmarks are used to provide relative addressing for efficient index update using `diff`. The inverted indexes we consider are full inverted indexes that differ from block addressing inverted indexes in that the position of the occurrences of each word in the text is stored.

## 2.4 An Analysis of Web Document Change

We analyzed a small set of web documents in order to understand the update characteristics of web documents. The results are summarized in this section as motivation for our landmark-diff index update method. Other than scale, our analysis differs from the study done by Fetterly et al. [32] in that our distance measure for two documents is based on exact edit distance whereas Fetterly et al. have used a shingle-based [10] approximate similarity measure. Because our landmark-diff index update method is based on edit transcripts, edit distance is a more appropriate distance measure in the context of this work; however, the edit distance is computationally more intensive and is not practical for large samples of the web such as those used in the study by Fetterly et al. We have chosen to restricted our study to a small sample of the web (relative to that used by Fetterly et al. and adopt the more relevant edit distance. Despite the difference in distance measure and scale, our results agree with those from the study by Fetterly et al.

We call the set of web documents associated with a snapshot of the web at a particular time a *sample*. The time between two consecutive samples is the *sampling interval*. A web document is defined to be the sequence of words contained in the HTML file that has been stripped of scripting code and HTML syntax. Each HTML file or each document is associated with an URL and is assigned a unique document ID (*doc_id*). Each word occurrence within a document encodes the information $\langle word\_id, doc\_id, loc\_id \rangle$, which is also known as a *posting*, where *word_id* denotes the unique ID identifying each word in the (English) vocabulary and *loc_id* denotes the position of that word occurrence in the document. Since each posting encodes all the information in a word occurrence on the web, the entire web can be encoded as a set of postings and a web information system can be viewed as a system maintaining the set of all postings (or a subset of it). For example, a web index is a web information system that stores this set of postings ($\langle word\_id, doc\_id, loc\_id \rangle$) sorted by *word_id*.

Consider two consecutive samples $S_n$ and $S_{n+1}$. Any document can only belong to one of the three partitions (see Figure 2.4 top diagram):

1. common documents $S_n \cap S_{n+1}$,
2. deleted documents $S_n - S_{n+1}$, and
3. inserted documents $S_{n+1} - S_n$.

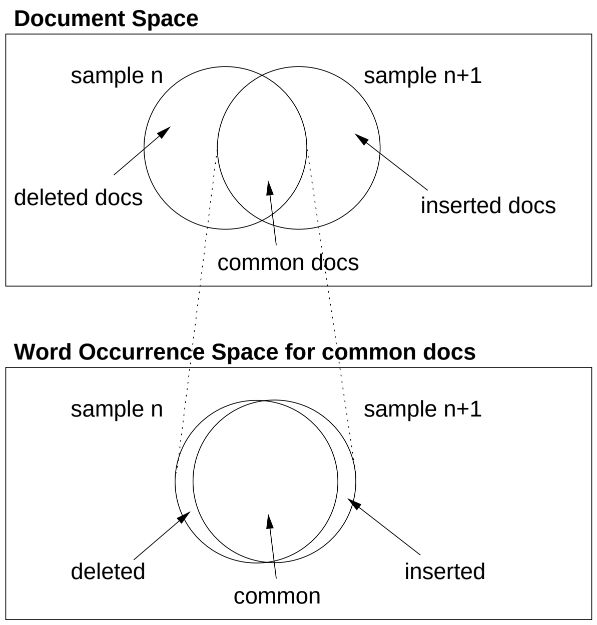


**Figure 2.4**: Types of changes at the document collection level and the word occurrence collection level.

In the Venn diagram at the top of Figure 2.4, a point represents a document and each circle represents a set of documents. Deleted documents are old documents that do not exist in the new sample anymore. Inserted documents are new documents that appear in the new sample only. Common documents are the documents that are common between the two consecutive crawlings of the web. Some of these common documents might be unchanged and some of it might contain changes. Corresponding these three partitions of documents, three operations are needed to update the index of $S_n$ so that it reflects $S_{n+1}$.

1. Postings corresponding to the deleted documents need to be removed from the current index;
2. Postings corresponding to the inserted documents need to be inserted to into the index,
3. Postings corresponding to the common documents that have changed need to

be updated in the index.

For our data analysis, we are primarily concerned with the changes in the common documents, because (1) techniques for inserting documents into information systems have already been well studied by the text retrieval community, (2) the common documents represents at least 50% of the documents currently maintained by the web information system at each update, (3) of these common documents, most of the changes at each update are small as we will see in Section 2.4.1. If we consider the set of common documents as a collection of word occurrences (defined next), then between two consecutive samples, the overlap is significantly large.

We can further examine the set of common documents at the granularity of a word occurrence. A word occurrence corresponds loosely to a posting (a $\langle word\_id, doc\_id, loc\_id \rangle$ tuple) without the limitation of a numeric location ID; that is, it is shift invariant in some sense. In the Venn diagram at the bottom of Figure 2.4, each point represents a word occurrence. Each circle represents the set of word occurrences corresponding to the set of common documents of a particular sample. If we consider each circle as a sequence of word occurrences, the set of common word occurrences loosely corresponds to the longest common subsequence of the two sequences. Since most web information systems keep track of word occurrences, the number of common word occurrences gives an upper bound on the number of postings in the system that can remain unchanged upon an update.

We analyze the changes occurring in the set of common documents in terms of the magnitude of change and the clusteredness of change. For our analysis, we used several samples of the web recursively crawled from several seed URLs at 12 hour intervals. The list of seed URLs consists of www.cnn.com, www.ebay.com, www.yahoo.com, espn.go.com, and www.duke.edu. We present our analysis using

| | |
|---|---|
| No. of docs at time $n$ | 6042 |
| No. of docs at time $n+1$ | 6248 |
| No. of deleted docs | 2788 |
| No. of inserted docs | 2994 |
| No. of common docs | 3254 |
| No. of common docs unchanged | 1940 |

**Table 2.1**: Summary of the characteristics of the representative data set used in the data analysis.

2 samples that are representative of the general update behavior. Other characteristics of our data are summarized in Table 2.1.

### 2.4.1 Magnitude of Document Change

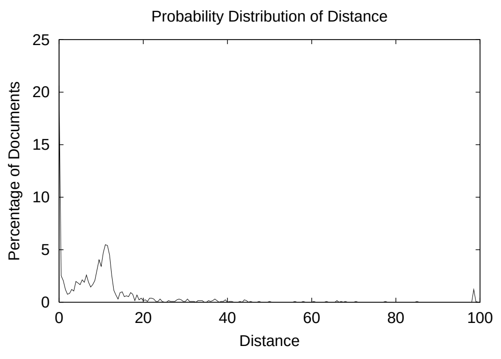
For common documents whose contents have changed, we are interested in how large the changes are. The magnitude of change between two versions of a document can be quantified by the *edit distance*. The edit distance of two documents is the minimum number of *edit operations* (word insertions or deletions) required to transform one document to the other. Let $\delta$ be the minimal number of words deleted or inserted to transform one document to another. We define the distance between two documents $A$ and $B$ as

$$d(A, B) = \frac{\delta}{m+n}, \tag{2.4.1}$$

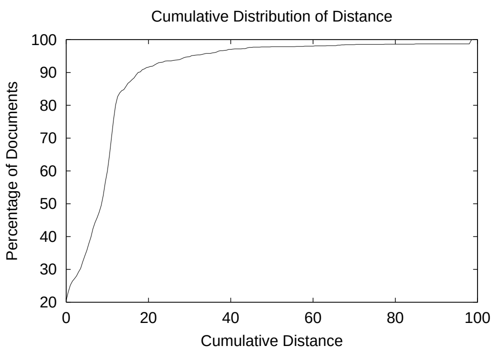
where $m$ and $n$ are the size (in words) of document $A$ and document $B$ respectively. Two identical documents will have zero distance and two completely different documents will have a distance of one.

For our empirical analysis, we measure the distance of each pair of old and updated documents (same document ID) in the set of common documents and put them into bins according to the distance. Each bin has a width of 0.5 %. The number of documents in each bin is normalized to a percentage and plotted in

Figure 2.5. We observe that most documents fall into the bins between distance



(a) Probability Distribution



(b) Cumulative Distribution

**Figure 2.5**: Distribution of documents with respect to our distance measure.

0 % and 20 % (Figure 2.5). In fact, the corresponding cumulative distribution plot shows that more than 90 % of the documents have changes smaller than a distance of 20 %. Moreover this behavior is consistent across the consecutive samples we crawled and we conjecture that the set of common word occurrences is very big in general. The implication is that a large portion of the information maintained by a web index can remain unchanged at an update.

### 2.4.2 Clusteredness of Change

Another important characteristic of change is how clustered the changes are. For example, from an update processing point of view, the insertion of a paragraph of 10 words to the beginning of a document is quite different from inserting the same 10 words at 10 different random and non-contiguous locations in the document. The former has locality that could be exploited in update processing.

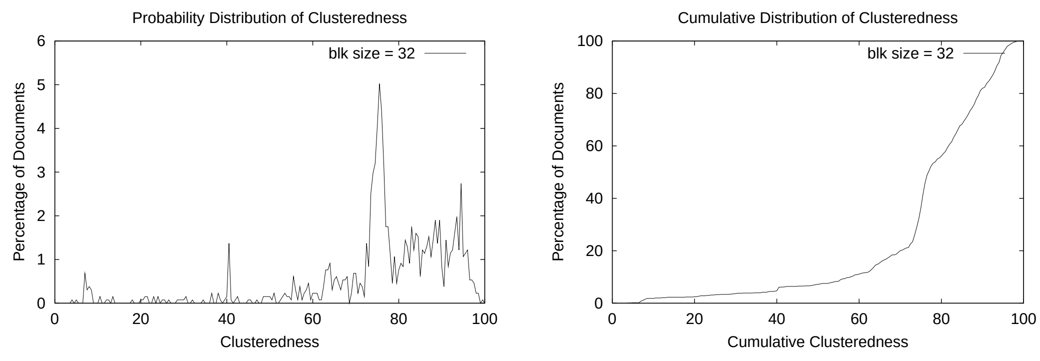
We measure how clustered the document changes are as follows. Choose a block size and partition the document into blocks according to the chosen block size. The fraction of the blocks affected by the changes is then used as a measure of how clustered the changes are. The *clusteredness* of the changes required to transform document $A$ to document $B$ is

$$c(A, B, b) = 1 - \frac{\Delta}{\lceil m/b \rceil}, \tag{2.4.2}$$

where $\Delta$ is the number of blocks affected by the change, $m$ is the size of the old documents in words and $b$ is the block size in words. If there are no changes, $\Delta$ will be zero and the clusteredness will measure one. If all the changes are clustered into one block and assuming that the block size $b$ is sufficiently small, the clusteredness will be close to one. If the changes are distributed over all the blocks, $\Delta$ will be equal to $\lceil m/b \rceil$ and clusteredness will be zero.

The block size $b$ must be chosen such that it is much smaller than the document size $m$ for this measure to be meaningful. Another possibility is to partition the document using HTML tags. One such tag is the paragraph tag `<p>`.

We study the empirical distribution of the common documents that have changed using this clusteredness measure. Figure 2.6 shows the clusteredness distribution for fixed size blocks. Figure 2.7 shows the clusteredness distribution for `<p>`-tag blocks. From the probability distribution plot with respect to our clusteredness measure $c(A, B, 32)$ (Figure 2.6), we observe that most document have changes that are more than 50 % clustered, that is, the changes affect less than half of the blocks. The corresponding cumulative distribution plot shows that only about 20 % of the documents have changes that are less than 70 % clustered. Using HTML paragraph tags, we observe in the probability distribution plot with respect to the clusteredness measure $c(A, B,$ `<p>`-tag$)$ (Figure 2.7) that many document have changes that are more than 50 % clustered; however significant spikes occur consistently at the 0-0.5



(a) Probability Distribution (b) Cumulative Distribution

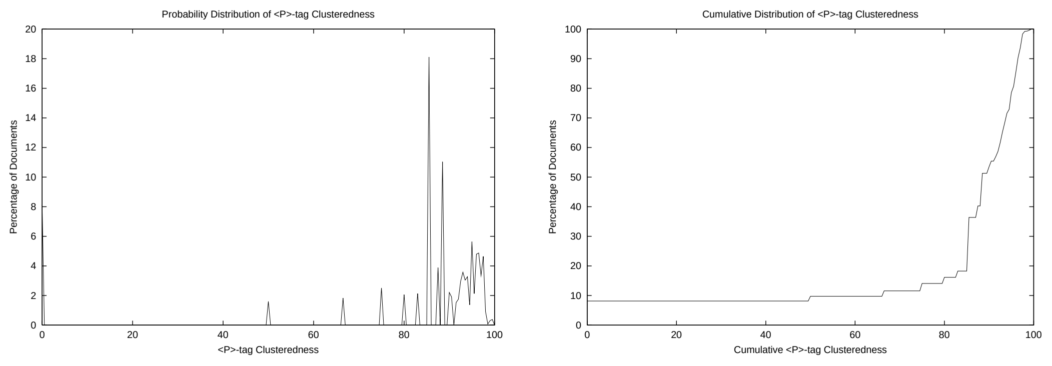
**Figure 2.6**: Distribution of documents with respect to clusteredness $c(A, B, 32)$.

% clusteredness bin. This is because some documents do not use the `<p>`-tag at all. For such documents there is only one block in total and any changes must occur in that block and hence $c(A, B,$ `<p>`-tag$)$ is zero. This is consistent with the previous $c(A, B, 32)$-distribution plot (Figure 2.6) where no documents have changes distributed to every block. Other observable artifacts are the spikes at the 50% and 66% clusteredness marks. These are mostly due to the documents with only two to three paragraphs in total.

We draw three conclusions from our empirical analysis that are true for the web samples we collected and we conjecture that they are true in general if the sampling interval is sufficiently small:

1. common documents represents at least 50 % of the currently indexed collection,
2. most of the changes in the common documents, are small, and
3. most of the changes are spatially localized within the document.

The landmark-diff update method that we propose in the next section is a scheme



(a) Probability Distribution (b) Cumulative Distribution

**Figure 2.7**: Distribution of documents with respect to clusteredness $c(A, B, \texttt{<p>}\text{-tag})$.

that exploits these properties.

## 2.5 The Landmark-diff Update Method

In this section, we present our landmark-diff index update method. Our method has the following desirable properties: (1) it is document based and incremental crawling can be used; (2) the number of update operations on the inverted index (the number of postings deleted and inserted) is independent of the document size, and depends on the size of the content change only; (3) the landmark-diff method is especially efficient when the set of common documents is large, and changes are small and clustered.

We first give an overview of our method and then discuss three specific issues: choosing landmarks, application scenarios and query processing with landmarks.

### 2.5.1 Overview

Our landmark-diff method combines two important ideas: encoding the position of words using landmarks in a document, and using the *edit transcript* or `diff` output of the old and new document to obtain the index update operations. An edit transcript is a list of operations that transform an old version of a document to the new version of the document. The landmark encoding scheme allows us to translate the edit transcript for the document into a very compact list of update operations on the inverted index.

We first introduce the landmark encoding scheme and then describe the landmark-diff update procedure. For the rest of this chapter, we reserve the term *edit transcript* for the `diff` output of a document and its updated version, and we use the term *update operation list* to refer to the list of inverted index update operations (delete or insert posting) that will update an inverted index.

#### Landmarks

The purpose of relative addressing using landmarks is to minimize the changes to the location IDs stored in the inverted index when documents change. Given a document, a number of positions in the document are chosen as landmarks. The position of each landmark in the document is recorded in a landmark directory structure for the document. The location ID for each word in the document is then encoded as a $\langle \text{landmark ID}, \text{offset} \rangle$ pair. Given a $\langle \text{landmark ID}, \text{offset} \rangle$ pair, the original location ID of the word occurrence is recovered by retrieving the position of the landmark from the associated landmark directory and adding the offset to that position. Each landmark acts as a reference point for the words between itself and the next landmark.

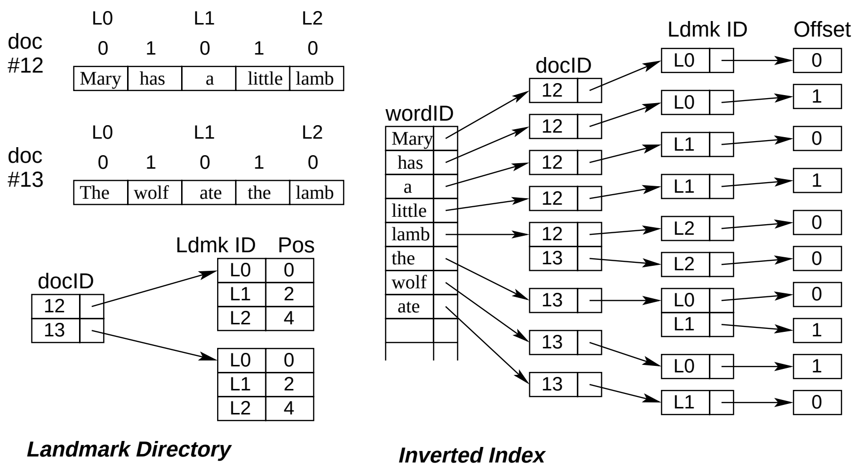


**Figure 2.8**: An inverted index with fixed sized landmarks. Note that this pointer-based representation is purely conceptual.

Putting landmarks in a document can also be thought of as partitioning a document into blocks. The starting position of a block corresponds to a landmark and the location of every word in a block is encoded as an offset from the start of that block. We use the term *block* to denote the words between consecutive landmarks. Each landmark corresponds to a block and vice versa. For the following discussion, we assume that a document is partitioned into fixed sized blocks. Other ways of choosing landmarks are discussed in Section 2.5.2. Figure 2.8 shows an inverted index with landmarks.

The landmark encoding does not increase the space requirement of the inverted index itself; however, auxiliary landmark directories need to be stored for each document. Suppose $k$ bits are allocated for each *loc_id* in an inverted index without landmark encoding. In an equivalent index using landmark encoding, the same $k$ bits in each posting can be used to encode a $\langle \text{landmark ID}, \text{offset} \rangle$ pair with the most significant $b < k$ bits used for storing the landmark ID and the least significant $k - b$ bits for the offset.

The landmark directory for a document is usually small. For a fixed block size of

$l$ words, the number of entries in a landmark directory for a document of $m$ words is $\lceil m/l \rceil$. For other ways of choosing landmarks, such as using HTML paragraph tags, the number of entries in a landmark directory is less dependent on the document size. Note also that a landmark directory lookup is not always required during query processing. Further analytical evaluation and implementation issues for the landmark directories are presented in Section 2.6.

### Update Procedure Using Landmarks

We show how to update an inverted index using our landmark-diff method in response to a content change in a document. The goal is to obtain an update operation list (an "edit transcript" for the inverted index) that can be applied to the inverted index to bring it up-to-date. The key idea is to obtain this update operation list from the edit transcript for the old and updated document. The landmark encoding scheme allows us to construct this update operation list using the edit transcript of the updated document, without increasing the number of delete or insert posting operations per document to $\Omega(m+n)$, where $m$ and $n$ are the number of words in the old and new version of a document. The details of the procedure for transforming a document edit transcript to an update operation list is lengthy but trivial, and we illustrate it by way of the example in Figure 2.9. The entire landmark-diff update procedure is also outlined in Figure 2.9. For each document that has changed, we obtain the edit transcript for updating that document using a `diff` procedure. The `diff` output is then transformed into corresponding entries in the update operation list for the inverted index using landmark information. The update operation list is then used to update the inverted index. All the procedures before the apply step require as input the old document, its landmark directory, and the new version of the document only; therefore these procedures lend themselves to parallel processing.

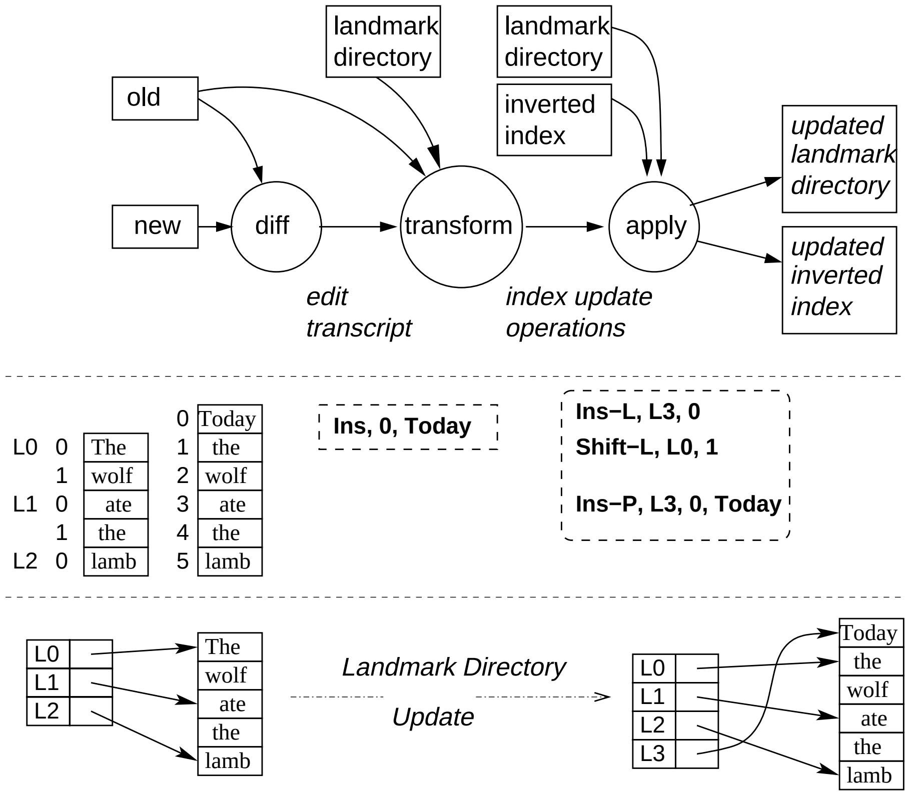


**Figure 2.9**: The inverted index update method using landmarks. The top diagram shows the data flow; the middle diagram shows what the edit transcripts look like with an example; the bottom diagram show how the landmark directory is updated. The landmark directory is represented here conceptually as a table.

In addition to updating the index, the landmark directory for each changed document needs to be updated as well, because insertion and/or deletion of words from the blocks within a document change the absolute position of landmarks within that document. Updating a landmark directory can be done very efficiently in a single sequential scan through the landmark directory data structure. This process is linear in the size of the landmark directory and the number of landmarks deleted and inserted. The overhead incurred for storing and maintaining landmark directories is not significant compared to the savings gained in the number of inverted update operations and in the update time as shown by our analytical and empirical evaluations in Section 2.6 and Section 2.7. Further analysis of landmark directories is given in Section 2.6.

### 2.5.2 Choosing Landmarks

A *landmarking policy* describes how landmarks are chosen in a document and also how these landmarks are to be processed during updates. Examples of landmarking policies are fixed size partitioning, HTML/XML tags, metadata, and semantic structure of document.

**Fixed size partitioning.** The fixed size partitioning policy partitions each document into blocks of a fixed size during index construction. During update processing, two ways of dealing with how an edit operation affects a block are possible and the landmarking policy needs to specify which one to use. Consider an edit operation that inserts a piece of text at some position within a block, the landmarking policy has to specify whether to make the block bigger or to split the block (Figure 2.9 shows the case when an insert always splits a block). Similarly, when a piece of text in the middle of a block is deleted, the landmark policy has to specify whether the remaining text forms one block or two blocks. A *split-block* policy is a policy where an edit operation always splits a block. A *grow-block* policy is a policy where an edit operation always makes a block larger. The split-block policy has the property that each block never exceeds the chosen size, while the grow-block policy has the property that the landmark directory never exceeds a certain size. In both cases, the variance in block size increases with the number of updates – a condition we term loosely as *fragmentation.* For the split-block policy fragmentation results in large landmark directories and hence increases the overhead in query and update processing that require accesses to the landmark directories. For the grow-block policy, fragmentation results in large blocks of text and increases the number of index update operations required. Defragmentation or index rebuild should be performed when update performance degrades. Since content changes in the updated documents are likely to be small, fragmentation would not occur frequently.

In fixed size landmarking, the landmarks are not inherent in the structure of the document. A brute-force `diff` on the old and updated documents is required to generate the edit transcript. Using the landmark information of the old document and the edit transcript, inverted index and landmark directory update operations are generated (see Figure 2.9 for an example). Each edit operation in the edit transcript is mapped to landmark directory update operations and index update operations in terms of landmark-offset pairs.

**HTML/XML tags.** HTML tags such as the paragraph tag (`<p>`) can also be used as landmarks. In contrast to the fixed size policy, the landmarks in this case are inherent in the document and the problem of whether an edit operation should split or grow a block is irrelevant. A more efficient, block-based, approximate `diff` procedure could be used instead of the brute force `diff`. For example, we could hash each block of the original and new versions and do the `diff` on the substantially smaller sequence of hashed values.

Which tags should be used as landmarks? Linear-time heuristics can be used to check which tags (or combinations of tags) that are suitable as landmarks. A brief description of the landmarking tags being used will then be stored together with the landmark directory. XML tags can be used in a similar way.

## 2.5.3 Some Application Scenarios

We have described the landmark-diff method for updating the index for common documents that have changed. We now give three example scenarios to illustrate how our landmark-diff method can be used to solve the general index update problem.

**Update Log.** If the inverted index is maintained using an update log (similar to "stop-press") in between complete index rebuilds, our landmark-diff method can be used to reduce the size of the update log. The naive update log corresponds to the list of inverted index update operations generated by the forward index update method (Section 2.2.3). Our landmark-diff update method can be used to significantly reduce the size of this update log and thus the query processing time.

**Partial Rebuild.** In contrast to a complete rebuild, a partial rebuild avoids re-indexing the common documents that did not change. Suppose that the inverted index is stored as a sorted array $A_{index}$ of postings, the *doc_id*s of the deleted documents are stored in a bitmap $\mathcal{B}_{deleted}$, and the postings of the inserted documents are stored in a sorted array $A_{inserted}$ (stop-press). The landmark-diff method can be used to maintain a reduced-size sorted update log $A_{update}$ for the common documents. The inverted index can then be updated in a single merging pass of the three sorted arrays $A_{index}$, $A_{update}$ and $A_{inserted}$ (checking $\mathcal{B}_{deleted}$ for deletes). Our experiments described in Section 2.7.3 show that partial rebuild can be twice as fast as complete rebuild.

**Distributed Index Update.** Suppose the document collection is partitioned among $M$ machines and indexed independently. At each update, each machine updates its index using a bitmap for the deleted documents and the landmark-diff method for the common documents. Inserted documents are always processed at a free machine that builds an index for them. When no free machines are available, the two machines with the smallest indexes are found and the two indexes are merged to free one machine.

### 2.5.4 Query Processing with Landmarks

Query processing using an inverted index with landmarks does not differ significantly from using a traditional inverted index. The main difference is that when positional information is required (either to check positional query constraints or to compute relevance rank), the landmark directories of all documents involved need to be retrieved to compute the location ID's of the postings. To minimize the number of landmark directories retrieved, the retrieval of landmark directories should be done after all document ID based filtering.

For phrase queries, the landmark directory retrieval and lookup can be avoided completely if positional information is not required. An additional field is associated with the last word occurrence of every block to store the ID of the next landmark. Given a phrase query ($key_1$ $key_2$), the postings for each keyword is retrieved from the index independently. Postings with the same document ID are grouped together and two postings, $\langle key_1, doc\_id, landmark\_id_1, offset_1\rangle$ and $\langle key_2, doc\_id, landmark\_id_2, offset_2\rangle$ form a phrase only if (1) $landmark\_id_1$ is equal to $landmark\_id_2$ and the difference between $offset_1$ and $offset_2$ is 1, or (2) $landmark\_id_1$ is not equal to $landmark\_id_2$, but $offset_2$ is zero and the posting for $key_1$ has the additional next landmark field that is equal to $landmark\_id_2$.

The cost of retrieving a landmark directory depends on the size of the landmark directory which depends on the landmarking policy. The cost of looking up the location ID of a landmark, depends on the implementation of the directory and we discuss the details in Section 2.6.3 and Section 2.6.7.

## 2.6 Analytical Evaluation

In this section we evaluate the performance of our landmark-diff update method analytically using the number of update operations as well as the more traditional running time complexity. We also show that the complexity of the `diff` operation is not a bottleneck, and that landmarking has minimal impact on query processing.

### 2.6.1 Update Performance in Number of Operations

Consider the number of updates (insert and delete postings) to an inverted index generated by our update method for a single edit operation on a single document. The following theorem states a key property of the landmark-diff method that all existing update methods lack: the number of updates generated by our method is independent of document size and dependent only on the size of the edit operation in the document and the maximum block size. The number of updates generated by existing update methods, on the other hand, is dependent on the size of both the original and the updated document.

**Theorem 1** *The number $U$ of updates to an inverted index caused by a single edit operation that deletes or inserts $\Delta$ words is at most*

$$U \leq \Delta + \epsilon,$$

*where $\epsilon$ is the number of words that are not affected by the edit operation in the last*

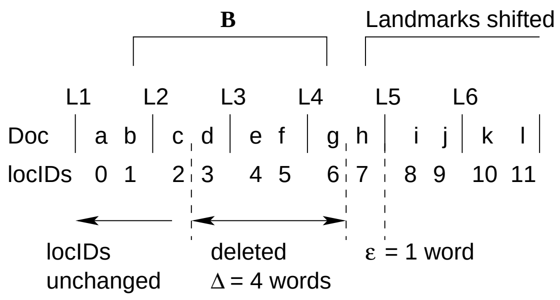


**Figure 2.10**: Consider a document with 12 words ('a' to 'l'). The block size is set at two words. Deletion of $\Delta = 4$ words generates $\Delta + \epsilon$ update posting operations on the inverted index. The set **B** contains all the segments affected by the change. Landmark ID $L3$ has to be deleted, the position of landmark $L4$ has to be changed to 3, the offset of the word 'h' has to be changed to 0, and the position of landmarks $L5$ and $L6$ has to be shifted by $-4$.

*modified block.*

*Proof*: Let **B** be the set of landmarks or blocks affected by this deletion or insertion of $\boldsymbol{\Delta}$ words in the document. For the example in Figure 2.10, $\mathbf{B} = \{L2, L3, L4\}$. All postings in the inverted index corresponding to the words in **B** will have to be updated except postings for the words occurring prior to the start of the deletion or insertion point. The number of updates is thus bounded by $\Delta + \epsilon$ words, where $\epsilon$ is the number of words following the change in the last block in **B**. In Figure 2.10, the posting for the word 'c' does not change, but the posting for the word 'h' changes, because its offset becomes zero after the deletion of four words. □

The following lemma follows from the analysis of the gap distribution given in [79]. The positions of the $L$ landmarks of a document are modeled as random, because the initially uniform landmark positions are no longer uniform after several random edit operations.

**Lemma 1** *If the $L$ landmarks of a particular document are randomly located, the number $U$ of updates to an inverted index caused by a single edit operation of size $\Delta$*

*on a document of size $m$ words satisfies*

$$U \leq \Delta + (m - L)/(L + 1).$$

## 2.6.2 The Complexity of the `diff` Operation

The use of edit transcripts (`diff` output) is a key idea in our method. The problem of computing an edit transcript given two documents is known formally as the longest common subsequence (LCS) problem. The worst case running time for a dynamic programming solution to the LCS problem is $\Theta(mn)$, where $m$ and $n$ are the number of words in the old and the new documents, respectively. A heuristic-based algorithm due to Ukkonen [78] achieves a running time of $O(D \min\{n, m\})$ using $O(D \min\{n, m\})$ space, where $D$ is the minimum edit distance. The UNIX `diff` program uses an output-sensitive heuristic algorithm similar to that of [78], so that the running time is near-linear when $D$ is small and quadratic if $D$ is linear.

Since `diff` is near-linear for small updates and the updates are usually small between two consecutive samples [56], `diff` is not a bottleneck in the processing in most cases. If the `diff` operation did form a bottleneck, we could represent each block in a document by its hashed value and perform the `diff` operation on the sequence of hash values rather than on the raw blocks directly.

**Theorem 2** *Let $L$ and $L'$ be the number of landmarks in the old and new documents, respectively. The block-based* `diff` *variant takes $O(D' \min\{L, L'\} + m + n)$ time to generate the edit transcript, where $L \leq m$, $L' \leq n$, and $D'$ is the block-wise minimum edit distance.*

False negatives (i.e., two identical blocks that are reported to be different) do not affect the correctness of the update method, but increases the size of the edit transcript and hence the number of update operations. False positives (i.e., two different blocks that are reported to match) could affect the correctness of our update method and need to be eliminated by doing a linear scan of the blocks that are reported to be identical to verify identity. All the blocks that are found different (both from the `diff` step and the linear scan step) will be the ones that need to be deleted and inserted into the system. The edit transcript would then consist of a list of delete landmark operations for blocks deleted from the old document and a list of insert landmark and insert postings operations for the blocks of text inserted into the new document.

### 2.6.3 Implementing the Landmark Directory

The running time complexity of the landmark-diff method is dependent on how the landmark directory is implemented. We briefly describe the operations that an implementation of the landmark directory must support.

**Insert(**$pos$**)** inserts a new landmark at position $pos$.

**Delete(**$landmark\_id$**)** deletes the landmark $landmark\_id$.

**DeleteRange(**$landmark\_id_1$, $landmark\_id_2$**)** deletes all landmarks occurring between $landmark\_id_1$ and $landmark\_id_2$.

**Shift(**$landmark\_id$, $value$**)** adds $value$ to the position of all the landmarks that occur after $landmark\_id$.

**Find_Pos(**$landmark\_id$**)** returns the position of the landmark $landmark\_id$.

**Find_Ldmk(**$pos$**)** returns the landmark corresponding to the position $pos$.

**Find_All_Ldmk(**$pos\_range$**)** returns all the landmarks corresponding to the positions in the range $pos\_range$.

| Operation | Offset Tree | Simple Array |
|---|---|---|
| Insert | $O(\log L)$ | $O(1)$ |
| Delete | $O(\log L)$ | $O(1)$ |
| DeleteRange | $O(\log L)$ | $O(B)$ |
| Shift | $O(\log L)$ | $O(L)$ |
| Find_Pos | $O(\log L)$ | $O(1)$ |
| Find_Ldmk | $O(\log L)$ | $O(L)$ |
| Find_All_Ldmk | $O(\log L + B)$ | $O(L)$ |
| Update Ops Generation | $O(\log L + B + \Delta)$ | $O(L + \Delta)$ |
| Query Overhead | $O((\sum_i s_{key_i}) \times \log L_{\max})$ | $O(\sum_i s_{key_i})$ |
| Space Overhead | $40L$ | $16L$ |

**Table 2.2**: Summary of the performance of the offset tree data structure versus that of the array data structure for landmark directories. The term $L$ is the number of landmarks in the document, $B$ is the number of landmarks affected by change or within a range, and $\Delta$ is the number of words deleted or inserted.

An offset tree [80] is the theoretically efficient data structure for a landmark directory. In practice, an array is used because of its compactness and good locality of memory reference. We analyze the offset tree data structure for landmark directories in Section 2.6.4, and the array data structure for landmark directories in Section 2.6.5. A summary of the analysis is given in Table 2.2. For the rest of the analysis we will assume that the landmark directories are implemented as arrays.

### 2.6.4 Offset Trees for Landmark Directories

Offset trees are binary trees that store an offset value in each node. They resemble the document representation data structures proposed in [30,33,80]. In the context of landmark-diff, we are not concerned with document representation, but with index representation together with novel index update techniques. Offset trees are frequently used in similar problems (such as in [80]) and they are theoretically very efficient. However, they require too much space as compared with arrays and exhibit poor locality of memory reference in practice.

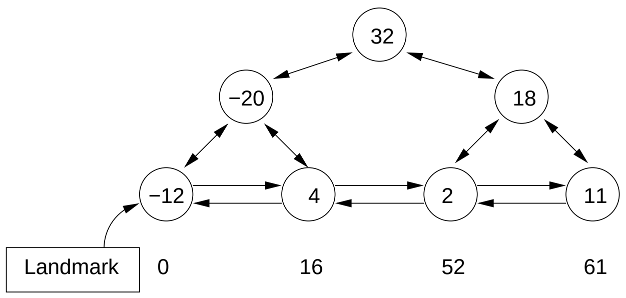


**Figure 2.11**: The offset tree data structure.

In our context, each node in the offset tree stores an additional parent pointer and each leaf node corresponds to a landmark of the document. The leaf nodes also store the following extra fields: the **size** field, the **byte position** field, the **left** and **right** pointers. The **size** field stores the extent (in number of words) of that landmark. The **byte position** field stores the byte position of the landmark in the document. The **left** and **right** pointers point to the adjacent leaf nodes and make the leaf nodes a doubly linked list.

The actual word position of a landmark is the sum of the offsets of the nodes in the path from the root to the leaf node corresponding to that landmark. Conceptually we can think of a landmark as a pointer to a leaf node without affecting the bounds we describe.

Offset trees can be height-balanced by using splay trees (amortized) or red-black trees (worst-case) [80]. Using offset trees, all the landmark directory operations (described in Section 2.6.3) take $O(\log L)$ time except for the operation Find_All_Ldmk(*pos_range*) which runs in $O(\log L+B)$ time, where $L$ is the number of landmarks and $B$ is the number of landmarks in the given range.

**Space Requirement.** For $L$ landmarks, how much space does this offset tree structure require? Each tree has $L$ leaf nodes and $L-1$ internal nodes. Each node stores an offset value and three pointers (parent, left and right). Each leaf node stores an additional two integers for the size and byte position fields. Assuming 4-byte pointers and integers, this structure requires $(L-1) \times 4 \times 4 + L \times 6 \times 4 \approx 40L$ bytes.

**Update Performance.** Suppose $\Delta$ contiguous words are deleted or inserted in an existing document. Using an offset tree, $O(\log L + B + \Delta)$ time is required to generate the inverted index update operations and to update the offset tree given the position(s) and the word IDs of the $\Delta$ words that are deleted or inserted.

**Query Performance.** The additional time required to process a conjunctive query of $k$ keywords $key_0 \wedge key_1 \wedge \ldots \wedge key_{k-1}$ is $O((\sum_{i=0}^{k-1} s_{key_i}) \times \log L_{\max})$, where $s_{key_i}$ is the selectivity of keyword $key_i$ and $L_{\max}$ is the maximum number of landmarks per document among all the inverted index entries involved in this query. The selectivity of a keyword is the number of inverted index entries with *word_id* equal to the given keyword.

## 2.6.5 Arrays for Landmark Directories

Although offset trees are theoretically efficient, they are not space-efficient for our application in practice. Arrays offer a more practical solution. Several array implementations of a landmark directory are possible depending on the space-time trade-off. We describe one simple implementation here as an illustration and analyze its update and query performance.

For a particular document, suppose there are at most $L$ landmarks and we allocate landmark IDs as integers in $[0, L-1]$. The landmark directory can be implemented as an array of size $L$ indexed by the landmark IDs. Each array entry stores the actual word position of that landmark, a previous landmark pointer, a next landmark pointer, and the actual byte position of the landmark in the document.

**Space Requirement.** Again assuming 4-byte integer/pointer for each of the four fields (word position, previous landmark, next landmark, and byte position), the space required by the array is $4 \times 4 \times L = 16L$ bytes. Some additional space could be used to accommodate future insertions, or else new space for the array can be allocated when $L$ changes. Thus, arrays provide significant saving in space over the use of offset trees.

**Update Performance.** The time complexity of the various update operations appears in Table 2.2. All lookups using a landmark ID takes constant time; hence Find_Pos(*landmark_id*) takes constant time. However, the Find_Ldmk(*pos*) operation requires $O(L)$ time, because the array is not indexed by word position and we have to scan through the array in order to find the corresponding landmark given its position. The Find_All_Ldmk(*pos_range*) operation also takes $O(L)$ time because we need one scan to locate the landmark ID corresponding to the starting position and then we can follow the next landmark pointers and check if the ending position has been reached. Since at most two passes through the array is required, it requires $O(L)$ time.

The insert, delete, shift, DeleteRange operations are not needed if a new updated landmark directory is generated together with the inverted index update operations. This is the update method of choice since only one linear scan is required per document to generate the update landmark directory. Nevertheless, for the sake of completeness, we give brief analysis of these operations.

If we maintain a free list of unused landmark IDs, the insert operation requires only $O(1)$ time. The delete operation requires a constant time lookup, a constant time nullify operation, and a constant time update previous landmark operation.

The shift operation requires changing the word position of all subsequent landmarks. The next landmark pointer is used to traverse through the subsequent landmarks and there are at most $L - 1$ subsequent landmarks.

For the DeleteRange operation, we lookup the starting landmark ID and traverse the range of landmark IDs using the next landmark field and nullify each entry; hence if $B$ is the number of landmarks in the given range, DeleteRange takes $O(B)$ time.

Note that the Delete and Shift operations are not required if the landmarking policy relies on the structure of the document (e.g., HTML tags), since we can rebuild the entire landmark directory from the new version of the document.

Compared with the logarithmic time lookup using offset trees, this simple array implementation offers constant time lookup of the word position given the landmark ID, an operation required during query processing. Query performance of arrays is therefore superior to offset trees; update performance requires the linear time reverse lookup operation (given position find the landmark ID) and hence is more expensive compared with offset trees.

### 2.6.6 Update Performance Using Arrays

**Theorem 3** *Let $\Delta$ be the number of contiguous words that were deleted or inserted and $L$ be the number of landmarks in the existing document. Using an array, $O(L+\Delta)$ time is required to generate the inverted index update operations and update the landmark directory given the position(s) and the word IDs of the $\Delta$ words that are deleted or inserted.*

*Proof*: In the case where UNIX `diff` is used, the edit transcripts consist of the positions of the words deleted or inserted and the words themselves. For delete, a position range of the words deleted is usually given. To generate the update entries for the deleted words, the operation Find_All_Ldmk(*pos_range*) is called. The operation Find_All_Ldmk(*pos_range*) outputs the landmark ID and offset of the starting word, the landmark ID and offset of the ending word and a list landmark IDs with their corresponding extents. Using this list and the information inside the edit transcript, we can generate the delete entries for the inverted file. The operation Find_All_Ldmk takes $O(L)$ time, where $L$ is the total number of landmarks for that document. Generating the delete landmark entries and applying them on the landmark directory take another $O(L)$, because at most $O(L)$ landmarks can lie in the given range and each delete landmark operation requires constant time. Generating the delete and insert posting entries for the words in the first and last block requires constant time assuming a constant block size. Hence to delete $\Delta$ contiguous words, $O(L + \Delta)$ time is required to generate the update entries for the inverted file and update the landmark directory given the edit transcript. A similar argument can be made for inserts. From our data analysis, we can deduce that $\Delta$ is typically very small.

In the case where a block-based variant of `diff` such as that described in Section 2.6.2 is used, an extra access to the old file or new file is required to obtain the words that are deleted or inserted. We show that this additional file access does not degrade the bound obtained in the previous paragraph.

Suppose that the variant `diff` returns an edit transcript in the following format. For delete, it returns the landmark ID of the block to be deleted. For insert it returns to the word position of the insertion point and the byte positions in the new document where the words to be inserted are located.

For deletion of a block of $\Delta$ words, we can obtain the word position, the byte position and the extent of the landmark in constant time. Using the byte position, we can access the starting point of the block of text to be deleted in the old document in $O(1)$ time. Reading the $\Delta$ words requires another $O(\Delta)$ time assuming a constant word length. We can now generate the delete entries for the inverted file and update the landmark directory as before. Since the file access requires only $O(\Delta)$ time the overall bound is still $O(L + \Delta)$ time. The argument for insert is analogous. $\square$

### 2.6.7 Query Performance Using Arrays

**Theorem 4** *The additional time required to process a conjunctive query of $k$ keywords $key_0 \wedge key_1 \wedge \ldots \wedge key_{k-1}$ is $O(\sum_{i=0}^{k-1} s_{key_i})$, where $s_{key_i}$ is the selectivity of keyword $key_i$.*

*Proof*: Recall how conjunctive queries are processed. For each keyword $key_i$, we obtain from the inverted index a list $l_i$ of inverted index entries of the form $\langle word\_id, doc\_id, loc\_id\rangle$. The selectivity of keyword $key_i$ is $s_{key_i} = |l_i|$, the number of inverted index entries with *word_id* equal to $key_i$. Entries from documents that do not contain all the keywords are filtered out of each $l_i$.

If the query post processing does not require the position of each occurrence of the word in a document, then we do not need to access the landmark directories and no additional cost is incurred. If positional information is needed (for example, to compute some relevant metric), then there is an additional $s_{key_i} \times O(1)$ time overhead to look up the actual position of each of the $s_{key_i}$ landmark IDs for a keyword $key_i$, assuming in the worst case that the the landmark IDs are all distinct. Hence, for a query consisting of a list of $k$ keywords $\{key_i \mid 0 \leq i < k\}$ joined by the boolean operator AND, the additional overhead is $O(\sum_i s_{key_i})$ time. Note that an offset tree implementation of the landmark directory is an order of magnitude slower, because each lookup requires $O(\log L)$ time (see Table 2.2). $\square$

## 2.7 Experimental Evaluation

In this section we describe the experiments used to evaluate our landmark-diff method. We compare the performance of the landmark-diff method with the forward index method, because both methods are designed for updating common documents whose contents have changed. For the performance on general index update, we compare the partial rebuild method using landmark-diff with the complete rebuild method, because the complete rebuild method is the industry workhorse.

Through our experiments, we answer five important questions about the performance of our landmark-diff method:

1. How does landmark or block size affect performance?
2. Does the landmark-diff method significantly reduce the number of edit operations on the inverted index compared to other methods (e.g., forward index method)?
3. Does the reduction in the number of inverted index update operations actually translate to savings in real execution time when applying these operations?
4. Does generating update operations using the landmark-diff method require more time than other methods?
5. Does the landmark-diff method provide a more efficient solution for the general inverted index maintenance problem than complete rebuild method, especially when the change between two consecutive samples is large?

**Our Implementation.** Our text indexing system is implemented in C. Two implementations of the inverted index have been used. The binary tree (of postings) implementation is used for measuring the time required to apply update operations on the index, because the binary tree represents the worst case tree-based data structure. We discuss a B-tree representation in Section 2.7.4 which will take better advantage of our landmark-diff method when updates are very frequent and therefore very small in magnitude. A b-way search structure implicitly represented by a linear array of postings is used for the comparison between the partial rebuild method and the complete rebuild method, because both methods rely on an external $k$-way merge sort and linear arrays give the complete rebuild method the most advantage. Landmark directories and forward indexes are implemented using linear arrays, because these are small data structures.

**Landmarking Policy and Block Size.** Fixed size partitioning is used to choose landmarks. Intuitively, the square root of the average document size is a good block size[2]. For the experiments we present, a default block size of 32 words is chosen since the average document size is roughly 1000 words. Insertion of words always split a block.

**Performance Measures.** We evaluate the performance of our landmark method using four measures:

1. the number of index update/edit operations (Table 2.3),
2. the time to perform those operations (Table 2.4),
3. the time to generate those operations (Table 2.5),

[2]The reasoning is similar to a 2-level B-tree where the number of blocks and the block size is 'balanced'.

4. the time to bring the inverted index up-to-date using the partial rebuild method (Table 2.6).

An edit operation is defined to be a delete posting or insert posting operation. The number of edit operations on the inverted index is a natural performance measure, since the goal of a good update method is to reduce the number of edit operations on the inverted index. Moreover, unlike the execution time, the number of edit operations depends neither on the implementation of the system nor on the hardware architecture. The number of edit operations is therefore a good measure of update performance across different search engine architectures and implementations.

**Data Set.** Two sets of data are crawled from the web. Data Set I consists of two samples of the web crawled from 100 seed web sites. The time between the start of the two web crawls is 71 hours and the recursion depth is limited to 5 levels. The first sample has 63,336 documents and the second has 64,639 documents with 37,641 documents common to both. Each sample contains about 1.5 GB worth of HTML files. Data Set II consists of 6 samples of the web crawled from 5 seed web sites (www.cnn.com, www.ebay.com, www.yahoo.com, espn.go.com, and www.duke.edu). The sampling interval is 12 hours and the recursion depth is 5 levels. Note that 4 out of the 5 listed web sites have fast changing contents.

**Document Preprocessing.** Every HTML file is preprocessed into a canonical form by stripping off HTML tags, scripting code, JAVA code, and extra white space. Every letter is capitalized so that different capitalizations do not result in different word IDs. If the canonical form of a file has less than 10 words, it is discarded.

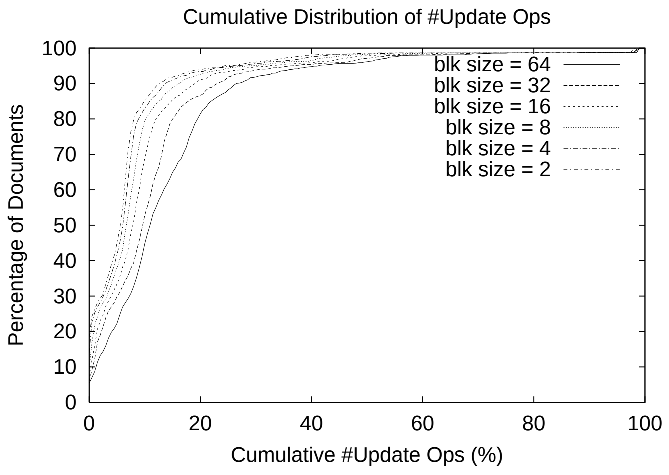


**Figure 2.12**: Cumulative distribution of documents with respect to the number of inverted index update operations for different block sizes.

### 2.7.1 Block Size and Performance

In this section, we address two questions: how does block size affect the number of inverted index update operations, and how does block size affect the time to generate these operations.

We measured the minimum number of index update operations required for different block sizes (using data set I described previously) and verified that as the block size gets smaller the number of index update operations decreases at the expense of increasing landmark directory size. Figure 2.12 shows the distribution of the documents with respect to the number of index update operations normalized by the sum of the sizes of the old and new documents. While small block sizes result in less update operations on the index, it also results in larger landmark directories, which translates to an increase in the cost of manipulating the landmark directory.

The increasing cost of manipulating the landmark directories as block sizes decreases can be seen in our experiment for investigating how the landmark size affects the time required to generate the update operations for the common documents

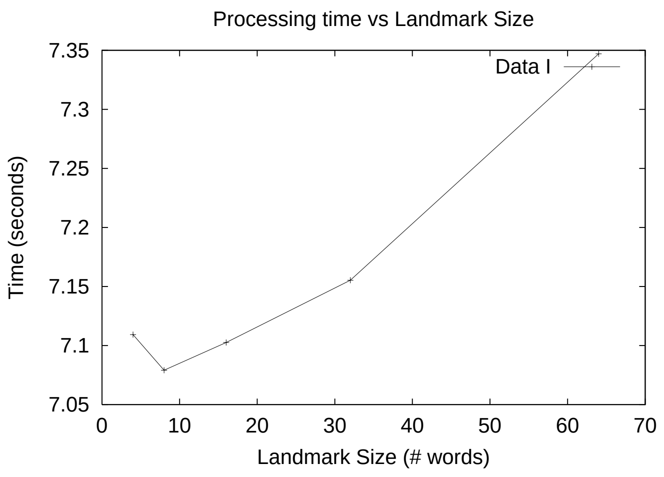


**Figure 2.13**: Landmark size versus time to generate update operations for common documents that have changed. Note that the time axis starts at 7.05 seconds.

(data set I) that have changed. Figure 2.13 shows that smaller landmark size results in less processing time to generate update operations for block sizes greater than eight words. For block sizes less than eight words, the cost of manipulating large landmark directories dominate and the processing time increases. Note that the block size that minimizes the time to generate update operations is dependent on the magnitude and clusteredness of change of the data as well as the implementation of the system.

In principle it is possible to determine the block size that minimizes the time for the entire update process for common documents given a data set and a particular system implementation. However, this is computationally intensive, and the computed optimal block size is not very useful, because it is dependent on the data, which change over time. One approach to reduce the computational overhead is to compute the optimal block size using a small subset (a statistical sample) of the data. A simple strategy to deal with changes in data characteristics is to recompute the block size during periodic index re-organization (defragmentation).

| Data | $\mathcal{C}$(docs) | $\Delta\mathcal{C}$(docs) | Fwd. Index | Landmark |
|---|---|---|---|---|
| I | 37,641 | 10,743 | 10,501,047 | 3,360,292 |
| IIa | 2,026 | 1,209 | 1,109,745 | 330,695 |
| IIb | 5,456 | 1,226 | 1,566,239 | 350,802 |
| IIc | 5,335 | 3,096 | 1,855,618 | 534,088 |
| IId | 5,394 | 1,278 | 1,426,163 | 378,661 |
| IIe | 5,783 | 1,605 | 1,762,018 | 539,594 |

**Table 2.3**: The number of update operations generated by the update methods in our experiments. The symbol $\mathcal{C}$ denotes the number of common documents ($|S_n \cap S_{n+1}|$) and the symbol $\Delta\mathcal{C}$ denotes the portion of $\mathcal{C}$ that has changed.

### 2.7.2 Performance on Common Documents

We compare the update performance of the landmark-diff method on common documents with the forward index method.

#### Number of Update Operations

We count the number of inverted index update operations generated by the forward index method and our landmark-diff method on two data sets. Fixed size landmarking policy is used with a block size of 32 words. Table 2.3 shows that the landmark-diff method generates significantly less update operations on the inverted index than the forward index method (which represents the best of the naive methods). The performance of the landmark-diff update method is consistent over a broad range of web sites including web sites with fast changing content. We indicate in Section 2.7.4 that this measure will mirror real-world performance when updates are frequent.

#### Update Time

Do the reductions in the number of inverted index update operations actually translate to savings in the time to apply these operations? We measure the execution

| Method | $\lvert S_n \rvert$ | No. Update Ops. | Time (s) |
|---|---|---|---|
| Fwd. Index | 63,336 | 10,501,047 | 28.3 |
| Landmark | 63,336 | 3,360,292 | 9.2 |

**Table 2.4**: Time required to apply the update operations on a binary tree implementation of the inverted index. The symbol $|S_n|$ denote the number of documents in the index and corresponds to the index size.

time for applying the update operations generated by the forward index method and the landmark-diff method to a binary tree implementation of the inverted index for data set I. The experiment is performed on a Sun Blade-1000 computer running SunOS 5.8 with 4 GB of RAM. Our results as summarized in Table 2.4 show that a reduction in the number of update operations does translate to a proportional reduction in the time required to update the inverted index. Note that the measured time does not include the time for updating the landmark directories. Updating a landmark directory requires only one linear pass and is done at the same time as generating the update operations (see Section 2.7.2).

### Time for Generating Update Operations

Does generating update operations using the landmark-diff method require prohibitively more time than the forward index method? We measured the time used to generate the update operations for data set I using the same configuration as in Section 2.7.2. For the forward index method, the measured time includes the time to read each pair of old and new document from disk, the time to create forward indexes for them, and the time to compare the forward indexes and generate update operations. For the landmark-diff method, the measured time includes the time to read each pair of old and new document from disk, the time to performing a `diff` on them, the time to translate the `diff` output to update operations, and the time to generate the updated landmark directory. Table 2.5 shows that for common docu-

| | | Generation Time (s) | |
|---|---|---|---|
| Method | No. Update Ops | $\mathcal{C}$ | $\Delta\mathcal{C}$ |
| Fwd. Index | 10,501,047 | 148.9 | 18.9 |
| Landmark | 3,360,292 | 17.5 | 7.5 |

**Table 2.5**: The time (in seconds) required to generate the inverted index update operations for the landmark-diff method and the forward index method. The $\mathcal{C}$ column gives the required time if all the common files (37,641) have to be checked and processed. The $\Delta\mathcal{C}$ column gives the required time, if incremental crawling is used and the system know *a priori* whether a common file has been modified since the last crawl.

ments that have changed the landmark-diff method is roughly three times as fast as the forward index method in generating update operations. This speedup is roughly the ratio of the output size (the number of update operations). Observe that `diff` is a more efficient method for detecting identical, i.e., unchanged, documents than the forward index method. In our experiments, the landmark directories are stored in memory, but even if the landmark directories are stored on disk, the additional time required to read a landmark directory into memory is still small compared with reading a document, since it is much smaller in size.

For common documents, our experiments have shown that the landmark-diff method does indeed reduce the number of update operations as compared to the forward index method. Moreover, the reduced number of update operations does translate to speedups of roughly the same ratio in update generation and application time. We next show how the landmark-diff method can be used for efficient general index update (not restricted to common documents).

### 2.7.3 Performance on General Index Update

We investigate the efficiency of using the landmark-diff method for solving the general index update problem. We compare the performance of the partial rebuild

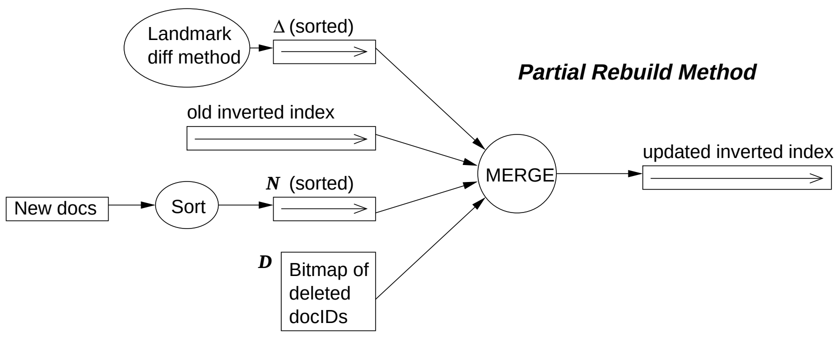


**Figure 2.14**: The schematic diagram for the partial rebuild method. Our landmark-diff method produces a list of update operations $\Delta$ which is then sorted. The new documents are processed into a sorted list of postings $\mathcal{D}$ to be inserted. The *doc_ids* of the deleted documents are stored in a bitmap. These three data structures are merged with the old inverted index in a linear pass.

method using landmark-diff (see Section 2.5.3) with that of the complete rebuild method.

In both methods, the inverted index is implemented as a sorted array of postings residing on disk (searching can be accomplished using an implicit complete b-way tree similar to a binary heap implemented as an array). For the complete rebuild method, the entire updated document collection is scanned into a list of postings and sorted using external $k$-way merge sort and written out to disk. For the partial rebuild method, the original and updated document collection is scanned to obtained (1) a list of document IDs of the deleted documents stored as a bit map in memory, (2) a sorted list of postings of the inserted documents stored as an array in memory, and (3) a sorted list of index update operations for the common documents, generated using our landmark-diff method, and stored as an array on disk. The old inverted index (on disk), the bit map, the array of new postings, and the sorted update operations are merged (in a fashion similar to mergesort) into an updated inverted index and written to disk (see Figure 2.14).

The two update methods are applied on data set I and the elapsed time mea-

| | No. of docs | No. of postings |
|---|---|---|
| $\mathcal{D}$ | 25,695 | - |
| $\mathcal{C}$ | 37,641 | 58,575,596 |
| $\Delta\mathcal{C}$ | 10,743 | - |
| $\Delta$ | - | 3,360,292 |
| $\mathcal{N}$ | 26,998 | 43,366,308 |
| Partial Rebuild | 513.8 s | |
| Complete Rebuild | 1113.7 s | |

**Table 2.6**: Running time performance of the partial rebuild method using landmark-diff and the complete rebuild method. The symbols $\mathcal{D}$, $\mathcal{C}$, $\Delta\mathcal{C}$, $\Delta$ and $\mathcal{N}$ denote the deleted documents, the common documents, the common documents that have changed, the update operations for the common documents and the new documents that have been inserted. We give the sizes of these sets in units of documents as well as postings.

sured. Table 2.6 shows that the partial rebuild using landmark-diff method is twice as fast as the complete rebuild method even for two web samples that are crawled 71 hours apart and thus have a large number of changed common documents and inserted documents. The partial rebuild method illustrates that the efficient update processing of the common documents can improve overall efficiency by avoiding the expensive re-indexing of the portion of the common documents that did not change. Hence, the partial rebuild method is more efficient because it only has to perform the expensive indexing operation on the new documents, whereas the complete rebuild method has to index both the updated common documents and the new documents.

Observe that the time required to process common documents (Table 2.4 and 2.5) is quite small compared to the overall update time of the partial rebuild method using landmark-diff (Table 2.6). A large portion of the elapsed time for the partial rebuild method is spent on the expensive indexing operation done on the new documents. One could argue that optimizing updates of common documents is meaningless because the bottleneck lies in indexing the new documents; however, the goal of this work is incremental updates. In the incremental update scenario, the time between updates is small and the new documents that arrive between updates are also fewer. Update processing of the common documents would account for the bulk of the processing time in incremental updates. If the set of new documents is significantly large, other ways of dealing with the new documents can be used. For example, a separate index can be built for the new documents as suggested in Section 2.5.3.

### 2.7.4 Discussion

Updating inverted indexes given a new sample of the web involves four sets of items:

1. the postings of deleted documents $\mathcal{D}$,
2. the postings for inserted/new documents $\mathcal{N}$,
3. the postings for original common documents $\mathcal{C}$, and
4. the update operations $\Delta$ for $\mathcal{C}$.

The existing inverted index (consisting of $\mathcal{C}$ and $\mathcal{D}$) has to be 'merged' with the sets $\mathcal{D}$, $\mathcal{N}$ and $\Delta$, so that $\mathcal{D}$ is deleted, $\mathcal{N}$ is inserted, and $\Delta$ is applied to the inverted index (the alternative is to rebuild from scratch). The set $\mathcal{D}$ can be processed very efficiently by storing the *doc_id*s using a bit map. Updating the index with $\Delta$ can be done very efficiently using our landmark-diff method (three times faster than the forward index method). The efficiency of incorporating $\mathcal{N}$ to the inverted index will depend on the size of $\mathcal{N}$.

The partial rebuild method using landmark-diff exploits the following three facts to achieve the factor of two speedup compared to the complete rebuild method: (1) $\mathcal{N}$ can be processed in a sequential manner very efficiently; (2) our landmark-diff method produces a very small $\Delta$ relative to $\mathcal{C}$ very efficiently; (3) the landmark-diff method avoids re-indexing the portion of $\mathcal{C}$ that did not change. Our goal is to investigate fast incremental update of inverted index. As the sampling interval decreases, the sizes of $\mathcal{N}$ and $\Delta$ also decrease relative to $\mathcal{C}$ [19]. In the limit, $\mathcal{N}$ will be very small and random access updates to an inverted index implemented as a B-tree will be faster than an array implementation. A similar trade-off between random access data structures and stream-based processing has been observed in the processing of spatial joins [3]. Therefore, the speedup in incremental update time using a B-tree implementation approaches

$$\frac{|\mathcal{C}| + |\mathcal{N}| + |\Delta|}{|\mathcal{N}| + |\Delta| + |\mathcal{D}|} \tag{2.7.3}$$

compared with a complete rebuild. We have shown in Table 2.6 that the array-based partial rebuild using landmark-diff is twice as fast as complete rebuild for a relatively large update interval of 71 hours. As the update interval decreases, the speedup will be more significant and even more dramatic if we use a B-tree implementation of the inverted index.

## 2.8 Conclusion

Keeping web indexes up-to-date is an important problem in research and in practice. Naive update methods such as index rebuild is inadequate in keeping the inverted index up-to-date especially in the context of in-place, incremental crawling. Incremental update methods are required to make newly crawled documents searchable.

Our analysis of change characteristics of web documents has shown that edits in web documents are generally small and clustered. These small and clustered change characteristics motivate the use of incremental update algorithms based on `diff`. However, positional information that is usually stored in the inverted index presents a problem in using the `diff` approach, because any edit can shift all subsequent word positions. We proposed the landmark-offset representation for encoding positional information to address this problem. The landmark-offset encoding has three advantages: it renders postings more "shift-invariant", it does not increase the index size, and it does not affect query processing (Section 2.5.4) significantly. The mapping of a landmark to its position in a document is maintained in a landmark directory. A landmark directory is very small compared with the size of the document and hence does not significantly affect query response time.

We have proposed the landmark-diff index update method that exploits the landmark-offset encoding with the `diff` approach to reduce the number of update operations on the inverted index. The landmark-diff method specifically addresses the problem of incremental index updates for previously indexed documents whose contents have changed – this problem has not been addressed before in the literature. We also showed how the landmark-diff method can be applied in a variety of ways to obtain new and more efficient solutions to the general index update problem and also to optimize many existing methods.

In our analysis, we showed that the number of inverted index update operations generated by our landmark-diff method is independent of the size of the document and is dependent only on the size of the document change and the maximum distance between two landmarks. In contrast, the number of inverted index update operations generated by previous methods are all dependent on the size of the document. Bounds on the update and query running time performance are also given.

In our experiments, we showed that our landmark-diff method is three times faster than the forward index method in updating the inverted index for common documents that have changed. We also showed how our landmark-diff method can be used in the partial rebuild method to solve the more general inverted index update problem. The partial rebuild method is twice as fast as a complete rebuild.

One important insight is that not re-indexing the portion of the document collection that did not change can result in more efficient update processing. The implication for incremental updates is significant, because smaller time intervals between updates mean that more of the document collection remains unchanged. Hence as the update frequency increases, the speedup from using the landmark-diff update method will be even more significant.

# Chapter 3

# Self-Adapting Set of Histograms.

## 3.1 Introduction

In this chapter, we study a particular selectivity estimation problem in the context of relational database systems: How to build and maintain a *set of histograms* in an on-line manner using query feedback information only. As explained in Section 1.2, selectivity estimation is an important problem in query optimization. Most relational database management systems (RDBMSs) maintain a set of histograms using a small amount of memory for selectivity estimation. While the problem of building an accurate histogram for a given attribute or attribute set has been well-studied [2, 12, 59, 68, 69, 81], little attention has been given to the problem of building and tuning a set of histograms collectively in a self-managed manner using only query feedback information.

A set of histograms is completely specified by

1. the sets of attributes on which each histogram is built,
2. the histogram technique (eg. equi-width, equi-depth) used for each histogram,
3. the statistics or parameters stored in each histogram, and
4. the memory (or buckets) allocated to each histogram.

For example, given a database with a single relation *Product(PID, Price, Sales)*, the singleton attribute sets $\{Price\}$ and $\{Sales\}$ to build histograms on, the equi-width histogram technique, 20 buckets worth of memory to be distributed uniformly among all the histograms, we can build a set of two 10-bucket histograms, one on attribute *Price* and one on attribute *Sales*, by scanning the tuples of the *Product* relation to obtain the width and the average frequency for each bucket.

In commercial database systems, the histogram technique is often fixed for a particular data type. The parameters stored in a histogram are dependent on the histogram type and in the off-line case, the parameters are usually obtained by scanning the database. In the on-line case, given a fixed attribute set and memory distribution, self-tuning histograms [2, 12] is a type of histogram that can learn its parameters from query feedback information.

The memory allocated to each histogram in most commercial DBMSs is a constant independent of the statistical properties of the data. Distributing memory to a set of histograms according to the statistical properties of the data has been addressed by [40] for one-dimensional histograms and [27] for multidimensional histograms. In this chapter, we address how to distribute memory to a set of multi-dimensional histograms according to the statistical properties of both the data and the query workload.

Most commercial database systems today use only one-dimensional histograms; therefore, the attribute sets are by default singleton sets. Selectivity estimates on multi-attribute queries are obtained by assuming statistical independence between attributes – this assumption is known as the attribute value independence (AVI). Using AVI, multi-attribute selectivity estimates are computed by multiplying single attribute estimates from the one-dimensional histogram. However, AVI rarely holds and estimates obtained using AVI are often wildly inaccurate [68]. To remove the AVI assumption, multidimensional histograms were proposed [68, 69]. Multi-dimensional histograms assumes a given attribute set on which to build the histogram – what attributes to include in the set is not addressed. Finding which attributes to group together for building multidimensional histograms is closely related to finding the correlation between attributes. Graphical models have been used to address this problem for the off-line case [27, 36] where the models can be computed with full access to the database relations. In this chapter, we address the on-line case where the graphical models are computed using query feedback information only.

In addition to building a set of histograms, we also want to tune the histograms for a particular query workload. Building and tuning a set of histograms using only query feedback information (without performing any off-line scan of the underlying database relations) present several unique challenges:

1. How do we decide which sets of attributes to build histograms on just by looking at query feedback?
2. How should the histograms be built and tuned using query feedback?
3. How should the fixed amount of memory be distributed among the histograms to achieve best overall accuracy for estimating the selectivities of a given workload?

We illustrate these issues with an example.

**Example 3.2** *Consider building a set of histograms for a database consisting of just the Product(PID, Price, Sales) relation (see Figure 3.1). The set of histograms would either consist of one 2-dimensional histogram on attribute set* $\{Price, Sales\}$

Product

| PID | Price | Sales |
|---|---|---|
| 1 | 10.50 | 120 |
| 2 | 13.00 | 1000 |
| 3 | 30.25 | 220 |
| 4 | 3.25 | 2020 |

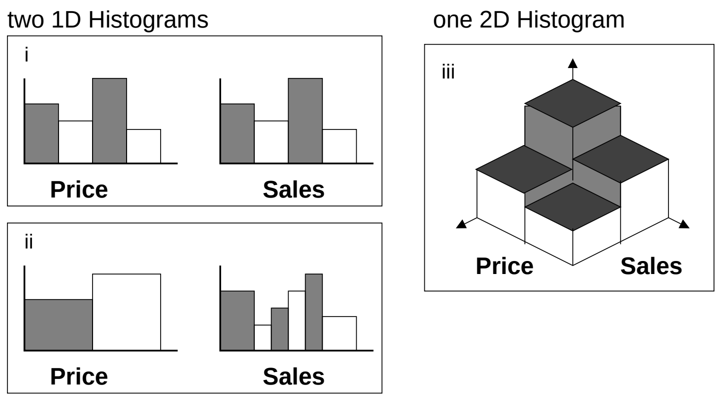


**Figure 3.1**: An example of the different sets of histograms that could be used for the product table.

*or two one-dimensional histograms, one on attribute Price and one on attribute Sales. The attribute PID is the primary key and does not require a histogram to be built for it. If two one-dimensional histograms are used, should the fixed amount of memory be allocated evenly between the two histograms (Figure 3.1, Set i), or should one histogram get more memory (Figure 3.1, Set ii) ? If there are many more queries on Sales than on Price, then we might consider allocating more memory to the histogram for attribute Sales, so that the histogram will be more accurate.*

In this chapter, we present SASH, a Self-Adaptive Set of Histograms that addresses the problem of building, maintaining and tuning a set of histograms. SASH uses a novel two-phase method to automatically build and tune itself using query feedback information only. In the on-line tuning phase, the current set of histograms is tuned in response to the estimation error of each query in an on-line manner. In the restructuring phase, a new and more accurate set of histograms replaces the current set of histograms. The new set of histograms (attribute sets and memory distribution) is found using information from a batch of query feedback. Graphical statistical models are used to model the sets of attributes on which histograms are built and the best model along with the best memory distribution among the histograms is computed using only query feedback information. The restructuring phase can be activated periodically or when performance degrades.

The rest of this chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 introduces basic notations. In Section 3.4 we describe SASH in detail: Section 3.4.2 describes the restructuring phase, Section 3.4.3 gives a brief introduction to graphical models, and Section 3.4.8 describes the on-line tuning phase. Section 3.5 presents an experimental evaluation of SASH. Section 3.6 summarizes this chapter and draws conclusions.

## 3.2 Related Work.

Previous work [2, 12, 27, 36, 40] addressed the issues of finding the attribute sets to build histograms on, allocating memory to the histograms, and tuning the histograms, by dealing with these issues independently and/or assuming full access to the underlying database relations.

The idea of using query feedback information to update the statistics kept by the query optimizer first appeared in [16] where coefficients of a curve representing the underlying data distribution are adapted using query feedback. Self-tuning histograms [2, 12] successfully used this idea to build and maintain individual histograms; however, neither [2] nor [12] addresses the issue of finding which attributes to build histograms on and the issue of distributing memory among the set of histograms. Our work addresses these two important issues. Moreover, SASH addresses how to update low dimensional histograms using high dimensional query feedback which has not been addressed in the literature before.

Graphical statistical models were first used for multidimensional query selectivity estimation in [27, 36]. Getoor et al. [36] use Probabilistic Relational Models (PRMs) for estimating the selectivities of point queries. PRMs [36] are based on directed graph models (Bayesian networks) and they are used in [36] to find which attribute sets to build conditional histograms on. However, the technique proposed in [36] is based on a complete off-line scan of the underlying data. It does not address the issues of on-line construction of histograms, memory distribution among multiple histograms, and workload-driven tuning of the existing histograms. Dependency-based (DB) histograms, proposed by Deshpande et al. [27], are based on undirected graph models (in particular, junction trees) and are used for estimating the selectivity of range queries. The technique proposed in [27] addresses the issue of which attribute sets to build histograms on and the issue of memory distribution among multiple histograms. However, it treats these two issues independently. Moreover, similar to the PRM-based technique in [36], it is based on a complete off-line scan of the underlying data and does not address on-line construction and workload-driven tuning.

Jagadish et al. [40] present several greedy and heuristic algorithms for distributing memory (buckets) among a set of single attribute histograms, but does not address the problem of finding the sets of attributes to build histograms on.

All three techniques [27, 36, 40] minimize some objective function that approximates the distance between the joint distribution associated with a set of histograms (or statistics) and the true data distribution of the database. The histograms (or statistics) that they maintain are obtained by scanning the database, and the objective functions that they minimize require accesses to the database to be computed. Because of the off-line nature of these techniques, they build histograms without considering how the histograms are being used (i.e., query workload) and assume that all queries are equally likely to be asked. This assumption is rarely true in practice. Ideally, more storage resource should be spent on storing the statistics that are relevant to the most frequently queried portions of the underlying relations. These techniques are oblivious to workload distribution and consequently waste precious storage space in storing statistics of infrequently queried portions of the base data. Another consequence of the off-line nature of these techniques is that the histograms they built are *static* in the sense that, after histograms are built, the histograms remain fixed regardless of any change in the data distribution. To ensure accuracy of the statistics when the base data change significantly, the histograms must be rebuilt by scanning the base data again. This rebuild approach is neither effective nor efficient because of the scanning cost associated with the size of the base data and the complexity associated with evaluating the objective functions that they minimize. Our work overcomes these drawbacks by building and maintaining histograms in a dynamic way based only on query workloads.

LEO by Stillger et al. [75] takes a different approach. It uses the actual selectivity feedback from each operator in the query execution plan to maintain *adjustments* that are used to correct the estimation errors from histograms. Note that LEO does not change the histograms themselves, while our work aims on building and maintaining better histograms using query feedback.

## 3.3 Preliminaries

A database consists of a set of relations. Each relation $R$ is a set of attributes. For the purpose of selectivity estimation, we can assume that relation $R$ contains only attributes of interest – attributes that are not relevant to selectivity estimation (e.g. primary and secondary keys) have been stripped off. Since we are not dealing with joins between relations, we simplify our presentation by considering a single relation $R$. Without loss of generality, let relation $R = \{A_1, A_2, \ldots, A_k\}$. Each attribute $A_i$ takes real values from the value domain $D_i$ (a discrete set of real numbers). Let $D_i$ be indexed by $\{1, 2, \ldots, |D_i|\}$ and let $\mathcal{D}(R)$ denote the domain for a set of attributes in relation $R$, i.e., $\mathcal{D}(R) = D_1 \times D_2 \times \ldots \times D_k$, where each $D_i$ is the domain for attribute $A_i \in R$. The normalized frequency distribution of relation $R$ is denoted by

$$P(R) = \frac{f(R)}{||R||} \tag{3.3.1}$$

where $f(R)$ is the frequency distribution for the attributes in $R$ and $||R||$ denotes the total number of tuples in relation $R$. Equation (3.3.1) allows us to treat frequency distributions as if they are probabilities. The frequency distribution $f(R)$ is a shorthand for

$$\begin{aligned} f(\mathbf{R}{=}\mathbf{a}) &= f(A_1{=}a_1, \ldots, A_k{=}a_k) \\ &= \text{no. of } \langle a_1, a_2, \ldots, a_k \rangle \\ &\quad\ \text{tuples in relation } R \end{aligned}$$

where $a_i$ is the value for attribute $A_i$. Geometrically, the tuple or vector $\mathbf{a}$ is a point in the $k$-dimensional domain space of $R$ (i.e., $\mathcal{D}(R)$). Let $\mathbf{X} = \{A_1, A_2, \ldots, A_j : j \leq k\} \subset R$. The frequency distribution for subset $\mathbf{X}$ can be obtained via *marginaliza-*

*tion*,

$$\begin{aligned} f(\mathbf{X}{=}\mathbf{x}) &= \text{no. of } \langle x_0, x_1, \ldots, x_j, \ldots\rangle \text{ tuples} \\ &\quad\ \text{in relation } R \\ &= \sum_{\mathbf{b}\in D(\mathbf{R}-\mathbf{X})} f(\mathbf{R}{=}\langle \mathbf{x}, \mathbf{b}\rangle). \end{aligned} \tag{3.3.2}$$

Let $\mathbf{r}$ be a set of real intervals constraining $\mathbf{X}$. We adopt the shorthand of $\mathbf{X}{\in}\mathbf{r}$ to represent a range query on the attributes of $\mathbf{X}$. For example, if $\mathbf{X}{=}\{A_i\}$ and $\mathbf{r}{=}\{[l,h]\}$, then $\mathbf{X}{\in}\mathbf{r}$ denotes the range query $l \leq A_i \leq h$. Geometrically, the $j$ intervals in a query range $\mathbf{r}$ form a hyper-rectangle in the $j$-dimensional domain space $\mathcal{D}(\mathbf{X})$. We denote the true selectivity of a query $q$ by $\sigma(q)$ and the estimated selectivity by $\hat{\sigma}(q)$. For example, the notation $\hat{\sigma}(\mathbf{X}{\in}\mathbf{r})$ denotes the estimated selectivity of the range query $\mathbf{X}{\in}\mathbf{r}$. Using the geometric interpretation, the selectivity of a range query $\mathbf{X}{\in}\mathbf{r}$ on relation $R$ can be computed as the sum of all the $j$-dimensional points $\mathbf{x} \in \mathcal{D}(\mathbf{X})$ that are enclosed by the hyper-rectangle defined by $\mathbf{r}$,

$$\sigma(\mathbf{X}{\in}\mathbf{r}) = \sum_{\substack{\mathbf{x}\in\mathcal{D}(\mathbf{X}) \\ \mathbf{x}\in\mathbf{r}}} f(\mathbf{X} = \mathbf{x}). \tag{3.3.3}$$

A *query feedback* is a tuple $(\mathbf{X}{\in}\mathbf{r}, \sigma(\mathbf{X}{\in}\mathbf{r}))$ consisting of a range query constraining the attributes in the set $\mathbf{X}$ to the ranges in $\mathbf{r}$ and the true selectivity (the number of tuples that satisfy that query) corresponding to this query.

A histogram $h$ is an approximation to a frequency distribution. In the following description of SASH and in our experiments, we have use partition-based histograms (and in particular MHIST [69]) in order to make comparisons with related work more meaningful. However, SASH is a general framework and does not assume any specific histogram type. A partition-based histogram $h$ is described by the set of attributes $attributes(h)$ and the set of buckets represented by the index set $B(h) = \{1, 2, \ldots, nbkts(h)\}$. Each bucket $i \in B(h)$ is associated with a particular partition $box(i)$ of the domain space $\mathcal{D}(attributes(h))$ and the frequency count of tuples that occur in that partition $freq(i)$.

## 3.4 SASH : Self-Adapting Set of Histograms

### 3.4.1 Overview

SASH is a general framework aimed at building, maintaining and tuning a set of histograms that

1. covers all the attributes of interest,
2. optimizes some performance criteria,
3. fits in a given amount of memory, and
4. require only query feedback information for construction and maintenance.

SASH consists of two phases: the *restructuring phase*, and the *on-line tuning phase*. A schematic of our approach is shown Figure 3.2.

The *restructuring phase* takes as input a batch of query feedback and performs a search for the optimal set of histograms and the corresponding memory allocation for each histogram. The restructuring phase is used to reorganize the set of histograms either periodically or when performance degrades. The restructuring phase can also be used to obtain an initial set of histograms when a batch of query feedback is available[1]. The restructuring phase can be activated when sufficient query feedback

[1]If query feedback is not available, the set of histograms can be initialized to a set of single attribute histograms constructed from the base relations.

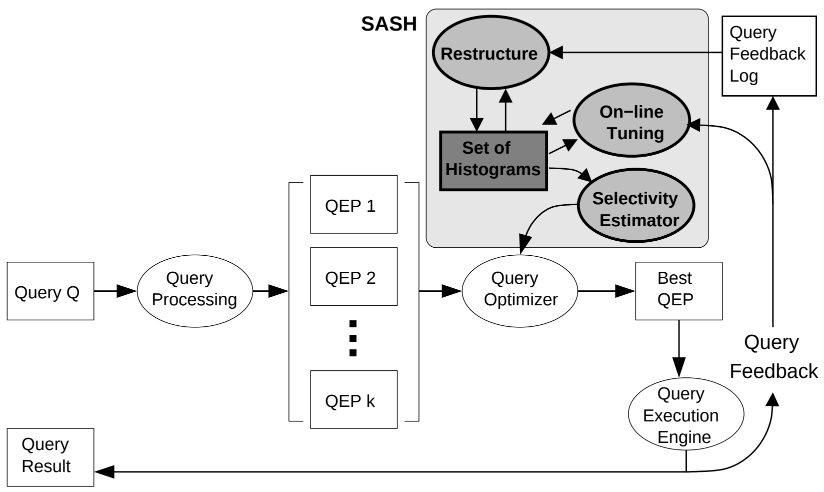


**Figure 3.2**: Schematic of SASH in relation to the query optimization and execution process.

has been collected. It is not feasible to run the restructuring phase after each query, because the restructuring phase involves the building of new histograms.

The *on-line tuning phase* tunes the frequencies of a set of histograms obtained from restructuring phase using query feedback in an on-line manner without changing the structure of the histogram set and the memory allocation among histograms.

We describe the restructuring phase in Sections 3.4.2–3.4.7. The on-line tuning phase will be described in Section 3.4.8.

## 3.4.2 The Restructuring Phase

The purpose of the restructuring phase is to find the best set of histograms with respect to a given multidimensional query workload. To that end, the restructuring phase takes a batch of recently seen query feedback (queries and their true selectivities) and searches for a set of histograms that best approximates the batch of queries.

Once the type of histograms (e.g., equi-depth, equi-width, MHIST, etc.) is chosen, a set of histograms is characterized by

1. the attribute sets on which the histograms are built,
2. the frequency or count stored in each bucket of each histogram, and
3. the amount of memory (or number of buckets) allocated to each histogram in the set.

We describe how SASH configures these parameters automatically in the next few sections.

There are two frequently used ways of choosing attribute sets: (a) each set contains a single attribute (also called the attribute-value independence or AVI assumption), and (b) each set contains all the attributes of interest from a single relation (also known as the saturated model[2]). In the first case, one histogram is built for each attribute of interest in the database, and in the latter, one multidimensional histogram is built for each relation in the database. Under tight memory constraints, both ways are error-prone and perform poorly for high-dimensional data with complicated correlations [27, 31, 36, 68].

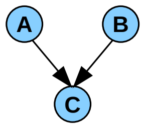
SASH's restructuring phase models a set of histograms using a *junction tree* graphical model with memory constraints. Given the query feedback information of a workload, SASH searches for the best model that fits in the specified amount of memory and that best approximates the given query workload.

We first introduce the junction tree graphical model before describing SASH's search algorithm for the best set of histograms.

[2]It is called the saturated model, because the histogram on this set of attributes captures all possible statistical correlations among all the attributes in that relation.

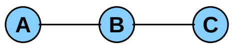
### 3.4.3 Graphical Models

Graphical models [45] are compact representations of high-dimensional joint data distributions. A graphical model $GM = (S, \Theta)$ consists of a graph $S = (V, E)$ and a set of parameters $\Theta$ encoding the associated distributions. The graph $S$ encodes the statistical dependence relationships between attributes and is often called the *structure* of the data. Each vertex represents an attribute and each edge represents a statistical dependence between two attributes. Using the graph $S$ and the associated distributions $\Theta$, we can reconstruct the joint distribution of all the attributes losslessly. Examples of graphical models include Bayesian networks,



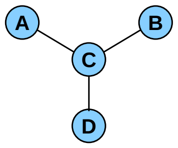
$$P(A, B, C) = P(C|A, B)P(A)P(B)$$

Markov chains,



$$\begin{aligned} P(A, B, C) &= P(C|B)P(B|A)P(A) \\ &= P(A|B)P(B|C)P(C) \\ &= P(A, B)P(B, C)/P(B) \end{aligned}$$

Markov networks, and junction trees:



$$P(A, B, C, D) = \frac{P(A, C)P(B, C)P(C, D)}{P(C)P(C)}.$$

For the rest of this chapter, we will use the junction tree representation; however, the techniques described in this chapter can be applied to other types of decomposable graphical models.

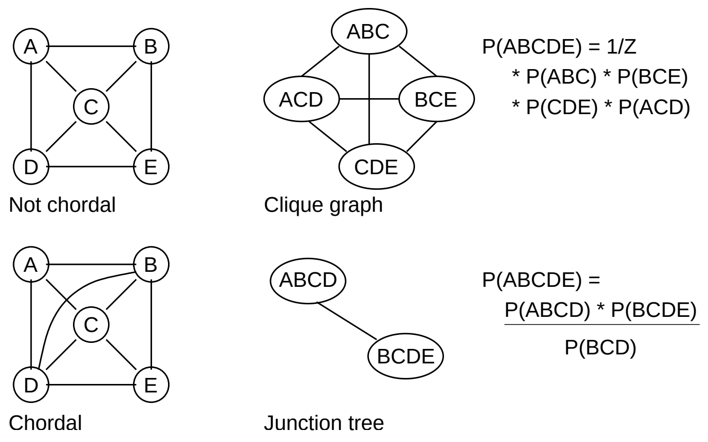


**Figure 3.3**: An example of a non-chordal graph (top left) and a chordal graph (bottom left) for five attributes $\{A, B, C, D, E\}$. The clique graph of the non-chordal and chordal graphs are shown on the right. Each of the graphical models on the left has five nodes corresponding to the five attributes. Each edge in the graphical model represents a statistical dependency between two nodes. The non-chordal graph differs from the chordal graph in the absence of the edge $(B, D)$.

In the junction tree representation, the graph structure $S$ is restricted to chordal graphs. A *chordal graph* is an undirected graph where, for every simple cycle of more than three vertices, there is some edge that is not involved in the cycle but joins two vertices in the cycle (a chord). A *clique graph* $CG(S)$ of any graph $S$ is a graph where where each node corresponds to a clique (maximal complete subgraph) in $S$ and each edge corresponds to a non-empty intersection between two cliques in $S$. If $S$ is chordal, its clique graph $CG(S)$ is a tree [26]. A *junction tree (or forest)* $J(S) = (V_J, E_J)$ of a chordal graph $S$ is the clique graph of chordal graph $S$. An example of a chordal graph, a graph that is not chordal, and their corresponding clique graphs is shown in Figure 3.3. Each node in a junction tree represents a set of attributes in the chordal graph $S$. We will use the notation $C_u$ to denote the set of attributes associated with a node $u \in V_J$. An edge $(u, v) \in E_J$ exists iff $C_u$ and $C_v$ has non-empty intersection. The set of parameters $\Theta$ associated with a junction tree consists of a set of distributions $\{P(C_v) : v \in V_J\}$, one for each node in the

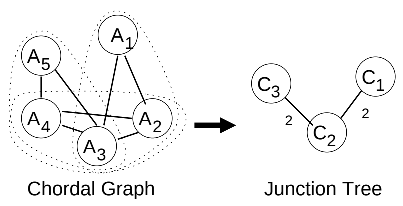


**Figure 3.4**: An example of a graphical model for five attributes $\{A_1, A_2, \ldots, A_5\}$. The junction tree has three cliques: $C_1 = \{A_1, A_2, A_3\}$, $C_2 = \{A_2, A_3, A_4\}$, and $C_3 = \{A_3, A_4, A_5\}$. The weight of each edge in the junction tree is the number of shared attributes between the end nodes of that edge.

junction tree. Each $P(C_v)$ is called a *clique distribution*.

Given a junction tree model $M = (J(S), \Theta)$ for a set of attributes $R = \{A_1, \ldots, A_k\}$, the joint distribution can be computed as

$$P(\mathbf{R}{=}\mathbf{a}) = \frac{\prod_{u \in V_J} P(C_u)}{\prod_{(u,v) \in E_J} P(C_u \cap C_v)}. \tag{3.4.4}$$

Each numerator term $P(C_u)$ is obtained directly from the clique distributions in the model parameter set $\Theta$. Each denominator term $P(C_u \cap C_v)$ is computed from the clique distribution for $C_u$ by marginalizing out the attributes in $C_u - C_v$ using Equation (3.3.2). Note that the attributes in the both the numerator terms $P(C_u)$ and the denominator terms $P(C_u \cap C_v)$ need to be instantiated with the values from the attribute value vector $\mathbf{a}$.

**Example 3.3** *Consider the chordal graph and junction tree model in Figure 3.4 for five attributes $\{A_1, A_2, \ldots, A_5\}$. Since the junction tree has three cliques, $C_1 = \{A_1, A_2, A_3\}$, $C_2 = \{A_2, A_3, A_4\}$, and $C_3 = \{A_3, A_4, A_5\}$, only three probability distributions, $P(A_1, A_2, A_3)$, $P(A_2, A_3, A_4)$, and $P(A_3, A_4, A_5)$, need to be stored. The joint probability distribution can be recovered using,*

$$P(A_1, A_2, \ldots, A_5) = \frac{P(A_1, A_2, A_3)P(A_2, A_3, A_4)P(A_3, A_4, A_5)}{P(A_2, A_3)P(A_3, A_4)}.$$

In the case where the clique distributions have a common normalizing constant, for example, if all the attributes come from the same database relation, Equation (3.4.4) can be rewritten in terms of frequency counts using Equation (3.3.1),

$$f(\mathbf{R}) = \frac{\prod_{u \in V_J} f(C_u)}{\prod_{(u,v) \in E_J} f(C_u \cap C_v)}. \tag{3.4.5}$$

We have described the junction tree model in the context of a single set of attributes. Recall that a database consists of many sets of attributes called relations. Since we do not consider joins, a junction tree/forest model of an entire database is simply a collection of junction trees for each relation. The parameter set $\Theta$ for this junction forest model is the collection of frequency distribution for each clique in this forest.

### 3.4.4 Estimating Selectivity with a Set of Histograms.

Graphical models can be used to describe the joint distribution of a set of attributes concisely. A set of histograms that approximates the joint data distribution in a database can likewise be modeled using graphical models with memory constraints. In particular, we extend the junction tree representation to describe a set of histograms (the sets of attributes to build histograms on, the parameters stored in the histograms and the memory distribution among the histograms).

**Definition 1** *A histogram set model (HSM) for a set of attributes of interest $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ is a triple $\langle J(S), \mathcal{H}, B\rangle$, where $J(S)$ is a junction tree or forest representation for the underlying chordal graph $S$, $\mathcal{H}$ is the set of clique histograms for the vertices in $J(S)$, $B$ is a set of bucket allocations that specify the number of buckets allocated to each histogram in $\mathcal{H}$, and the attributes in all the cliques cover* *all the attributes of interest (i.e.,* $\cup_{u \in V_J} C_u = \mathcal{A}$*).*

A histogram set model differs from a junction forest model $(J(S), \Theta)$ in that the parameters $\Theta$ of the junction forest, i.e., the clique distributions, are stored *as frequency counts* and the count distributions are stored *approximately.* In the context of selectivity estimation in database systems, a very limited amount of memory is typically allocated for storing the statistics (e.g. in the form of histograms) used in selectivity estimation; therefore it is impractical to store the count distribution for each clique $C_u$ exactly. These count distributions are approximated by the histograms in $\mathcal{H}$ using the memory constraints specified in $B$. We call the histograms that approximate the clique distributions *clique histograms.*

**Example 3.4** *Consider a database containing only the Product(PID, Price, Sales) relation (the same relation as in Figure 3.1). Suppose the database system has allocated 3 buckets worth of memory for storing statistics and builds a one-bucket equi-width histogram for attribute Price, and a two-bucket equi-width histogram for attribute Sales. Note that PID is a primary key and therefore not an attribute of interest. The histogram set model for the set of histograms in this database is,*

$$\begin{aligned}
S &= (\{Price, Sales\})\,,\ \emptyset),\\
J(S) &= (\{p, s : C_p{=}\{Price\}, C_s{=}\{Sales\}\},\ \emptyset)\,,\\
\mathcal{H} &= \{h_p{=}\{\langle[10.5, 30.25], 4\rangle\},\ h_s{=}\{\langle[120, 1070), 3\rangle, \langle[1070, 2020], 1\rangle\}\}\,,\\
B &= \langle 1, 2\rangle,
\end{aligned}$$

*where* $h_p$ *and* $h_s$ *represents the histogram for Price and Sales respectively, and the memory distribution* $\langle 1, 2\rangle$ *specifies that histograms* $h_p$ *and* $h_s$ *are allocated* 1 *and* 2 *buckets respectively. Each histogram is denoted as a set of buckets. Each bucket is* *denoted as a pair consisting of an interval and a count, so $h_p$ consists of one bucket that covers the interval* $[10.5, 30.25]$ *in the domain space of attribute Price and there are* 4 *tuples contained in this interval.*

Given a set of histograms described by the HSM $\langle J(S)=(V_J, E_J), \mathcal{H}, B\rangle$ and a range query $q = \mathbf{X}\in\mathbf{r}$ on a relation $\mathbf{R}$, we estimate the selectivity $\sigma(q)$ as follows.

1. If the query attributes form a subset of some clique, $\mathbf{X} \subseteq C_u, u \in V_J$, then compute the selectivity estimate directly using the clique histogram corresponding $C_u$. Note that we can always compute the selectivity of a query on attribute set $\mathbf{X}$ using a multidimensional histogram for attribute set $C_u$ using marginalization (Equation (3.3.2)), if $\mathbf{X} \subseteq C_u$.

2. Otherwise use the junction tree $J(S)$ and the clique histograms $\mathcal{H}$ to construct a histogram for a joint distribution $f(\mathbf{R}')$ that contains the required estimate.

   Let $V' = \{v \in V_J : C_v \cap \mathbf{X} \neq \emptyset\}$ be the set of cliques that contain some of the query attributes and $E' = \{(u, v) \in E : v, u \in V'\}$ be the set of edges that have the nodes in $V'$ as endpoints. Define $\mathbf{R}' = \cup_{v\in V'} C_v$ to be the union of all the cliques that contain some of the query attributes. Since the HSM cover all attributes of interest, the set $\mathbf{R}'$ contains all the query attributes, $\mathbf{X} \subseteq \mathbf{R}' \subseteq \mathbf{R}$. The histogram for $f(\mathbf{R}')$ is then constructed using a specialization of Equation (3.4.4),

$$f(\mathbf{R}') = \frac{\prod_{u\in V'} f(C_u)}{\prod_{(u,v)\in E'} f(C_u \cap C_v)}, \tag{3.4.6}$$

   Each numerator term $f(C_u)$ and denominator term $f(C_u \cap C_v)$ can be approximated from the clique histogram for attribute set $C_u$.

3. Compute $\hat{\sigma}(q)$ directly using the histogram for $f(\mathbf{R}')$.

For MHIST histograms [68], efficient algorithms for computing $f(\mathbf{R}')$ from the junction tree model and the clique histograms exist and are described in [27]. For the rest of the chapter we will assume the MHIST histogram type, even though other histogram techniques can be used with minor modification.

**Example 3.5** *Consider a relation with five attributes of interest* $\{A_1, A_2, \ldots, A_5\}$ *and a HSM that specifies a junction tree model with cliques*

$$\{\{A_1, A_2\}, \{A_2, A_3\}, \{A_3, A_4\}, \{A_4, A_5\}\},$$

*and clique histograms* $\{h_i : \textit{attributes}(h_i)=\{A_i, A_{i+1}\}\}$*. Compute the selectivity of range query* $q = A_1{\in}r_1, A_3{\in}r_3$*.*

*Since only the cliques* $\{\{A_1, A_2\}, \{A_2, A_3\}\}$ *intersect the query attributes, we construct a histogram* $h'$ *for*

$$f(A_1, A_2, A_3) = \frac{f(A_1, A_2) f(A_2, A_3)}{f(A_2)},$$

*using* $h_1$ *and* $h_2$*. The selectivity* $\sigma(q)$ *is then computed directly from the histogram* $h'$*.*

*Note that conceptually, attribute* $A_1$ *is statistically dependent on* $A_3$ *via* $A_2$*. Therefore the required selectivity needs to be computed from* $f(A_1, A_2, A_3)$ *by summing out the unwanted attribute* $A_2$*,*

$$f(A_1, A_3) = \sum_{a \in \mathcal{D}(A_2)} f(A_1, A_2{=}a, A_3).$$

### 3.4.5 Finding the Best Set of Histograms

Having defined a histogram set model (HSM), we now tackle the main problem in the restructuring phase: how to find the best set of histograms given a batch of query feedback. Since a set of histograms can be formally described by a HSM, we will use a HSM to refer to a set of histograms.

Suppose the quality or goodness of a HSM can be evaluated using some scoring criterion (Section 3.4.6 discusses different scoring criteria), the problem of finding the best set of histograms for a database can be cast as a search for the HSM that maximizes or minimizes the scoring criterion.

Since a HSM is parameterized by a graphical structure, a set of histograms, and a vector of memory allocation, the search space is the cross product of all possible graphical structures, all possible distribution of the fixed amount of memory to the histograms, and all possible histogram configurations. In general, for $n$ attributes, there are $O(2^{n^2})$ different (undirected) graphs, and for $m$ buckets to be allocated to $k$ histograms, there are $\binom{m-1}{k-1}$ different distribution patterns. The number of possible configurations of a histogram is dependent on the histogram technique. In the case of SASH, the graphical structure is constrained to chordal graphs (or equivalently junction trees) and the histogram configurations are constrained once the graphical structure and memory distribution are fixed (see Section 3.4.7). Even with these constraints, it is not feasible to perform an exhaustive search. Therefore, we use a greedy search strategy.

Our search algorithm is outlined in Algorithm 3.1. We initialize the current model $m$ to the simplest model (lines 1–3) and iteratively improve it. The simplest model assumes attribute value independence and builds a one-bucket histogram for each attribute (see Figure 3.5 for an example). At each iteration of the `while`-loop

**Algorithm 3.1** RESTRUCTURINGPHASE(*Qfb*, *C*)

```
INPUT: memory constraint C, histogram set model $m = \langle J(S), \mathcal{H}, B \rangle$
1: $J(S) \leftarrow$AVI
2: $\mathcal{H} \leftarrow$one bucket histograms
3: $B \leftarrow$vector of one's
4: $c \leftarrow bytesize(m)$
5: while $c < C$ do
6:    for all $m_{struct} \in S\_Candidates(m)$ do
7:       score($m_{struct}$, $Qfb$)
8:    end for
9:    $m^*_{struct} \leftarrow m_{struct}$with best score
10:   for all $m_{bucket} \in B\_Candidates(m)$ do
11:      score($m_{bucket}$, $Qfb$)
12:   end for
13:   $m^*_{bucket} \leftarrow m_{bucket}$with best score
14:   $m \leftarrow$best scoring model between ($m^*_{bucket}$, $m^*_{struct}$)
15:   $c \leftarrow bytesize(m)$
16: end while
```

in line 5, the algorithm explores two options:

1. making the best local change in structure, i.e., the junction tree (`for`-loop of line 6), or

2. adding one or more buckets to a histogram (`for`-loop of line 10), but keeping the current structure (no changes to the current structure).

The notation *S_Candidates(m)* refers to the set of candidate models obtained by making a local structural change on model $m$ and *B_Candidates(m)* refers to the set of candidate models obtained by adding one bucket to some histogram in model *m*. The option that gives the best scoring model is taken (line 14) and the algorithm proceeds to the next iteration. A local change in structure is defined to be the addition of an edge to the underlying chordal graph $S$ that does not violate the chordal graph property. Computing the set of candidate junction trees that differ from $J(S)$ by one edge in the chordal graph $S$ has been described in [26]. Making

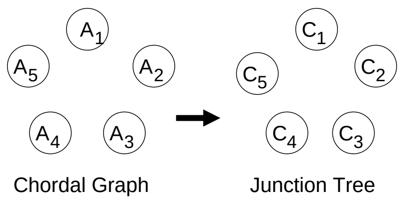


**Figure 3.5**: An example of the AVI graphical model for five attributes $\{A_1, A_2, \ldots, A_5\}$. The mutual independence of the attributes is represented by the lack of edges in the chordal graph (left). The junction tree (right) reduces to singleton cliques: $C_i = \{A_i\}$ for $i = 1, \ldots, 5$.

a structural change always results in some old clique histograms being discarded and one new clique histogram of higher dimension being created. The buckets of the discarded histograms are assigned to the new histogram and in addition, the algorithm also explores giving more buckets to the new histogram, because it has higher dimensionality than the discarded ones.

At each iteration of the search (line 5), we add buckets to either existing clique histograms or the new one. When the (memory) size of the model reaches the desired threshold, the search algorithm terminates and outputs the best model that it finds.

### A Few Optimizations

**Caching histograms to avoid recomputation.** Histograms that are constructed for scoring a candidate model in the current (`while`-loop) iteration should be kept in a cache in order to avoid recomputation when they are needed again to score candidate models in future iterations. For example, suppose the current iteration considered the candidate model that includes the clique $\{A_1, A_2\}$. In order to score this candidate model, a histogram for $\{A_1, A_2\}$ is constructed from the query workload. Suppose this candidate model is not the best scoring option and the search algorithm selected another option (e.g. adding one more bucket to the histogram for $\{A_3\}$). At the next iteration, the algorithm will consider a candidate model that includes the clique $\{A_1, A_2\}$ again. If the histogram constructed for $\{A_1, A_2\}$ is kept in a cache instead of being discarded, the algorithm can retrieve it from the cache instead of constructing it from scratch again.

**Persistent histograms for precomputing histograms of different sizes.** In combination with caching a histogram associated with a particular clique, histograms for the same clique but with different bucket/memory allocations should be precomputed (and cached). For example, suppose the current iteration is considering adding one or two buckets to the histogram for attribute $\{A_3\}$ which currently has four buckets. In order to score these two options, a five-bucket histogram for $\{A_3\}$ and a six-bucket histogram for $\{A_3\}$ needs to be constructed. Again, these candidate histograms may be needed in the next iteration and should be cached. A persistent histogram data structure that can store the history of a clique histogram as it grows in number of buckets would enable more efficient caching. In the case of MHIST, a single binary space partition tree can be used to efficiently store multiple versions of the MHIST for the same clique, but with different number of buckets.

**Pruning edges.** When generating *S_Candidates(m)*, the set of candidate (chordal graph) models that differ from the current model by one edge, not all edges (that do not violate the chordal graph property) need to be considered. We say that an edge $(A_1, A_2)$ between attributes $A_1$ and $A_2$ *occurs* in a query workload if there exists at least one query that specifies range constraints on both attributes $A_1$ and $A_2$. Some edges may not occur in the query workload at all and hence will never affect the score. For example, if the attribute *Price* and *Sales* never occur together in the same query in the query workload, then correlation statistics between the two attributes will never be needed for that query workload. Generalizing this idea, we can find all the edges that occur in the query workload and rank them according to how frequent they occur in the workload. Low frequency edges can be ignored for greater efficiency.

**Bounding the histogram dimensionality.** Resources for constructing and storing high-dimensional histograms are limited in most commercial databases. Moreover, most user generated queries are not likely to involve a high number of attributes. Hence, we can place a bound on the dimensionality of each clique histogram that are considered in the search algorithm: candidate graphical models that contains a clique that has size greater than a given size (typically three) need not be considered [27].

### 3.4.6 Scoring Criteria

In this section we describe some of the scoring functions we have used with the search algorithm outlined in Algorithm 3.1. Previous work on selectivity estimation using graphical models [27,36] evaluated the candidate models against the true distribution, i.e., the database itself. Since only feedback information for a query workload is available in our case, scoring functions that are based on the Kullback-Liebler divergence [50] from the true distribution cannot be used. Candidate models will have to be evaluated based on the information from the batch of query feedback. We describe two scoring techniques that require only information from query feedback: standard error measures and the minimum length description criterion.

### Standard Error Measures

A candidate model $m'$ can be evaluated based on its selectivity estimation performance on the queries in the batch of query feedback. Selectivity estimation performance can be measured using a variety of error measures [81]. Two examples are the 2-norm of the absolute error,

$$score_{abs2n}(m', \mathit{Qfb}) = \left[ \frac{1}{N} \sum_{(q,\sigma) \in \mathit{Qfb}} |\hat{\sigma}_{m'}(q) - \sigma|^2 \right]^{\frac{1}{2}},$$

where $N$ is the number of query feedback tuples in the query feedback batch *Qfb*, and the 1-norm of the relative error,

$$score_{rel1n}(m', \mathit{Qfb}) = \frac{1}{N} \sum_{(q,\sigma) \in \mathit{Qfb}} \frac{|\hat{\sigma}_{m'}(q) - \sigma|}{\max(1, \sigma)}.$$

### Minimum Description Length (MDL)

Using the MDL principle [6], we can evaluate a candidate model $m'$ by the number of bits required to encode the selectivity information in the batch of query feedback using $m'$. Given this encoding, the true selectivities in the batch of query feedback can be losslessly reconstructed. Since $m'$ provides estimates to the true selectivities, we only need to encode the estimation errors and the model $m'$ itself. The decoder can use the model $m'$ to estimate the selectivity of each query in the batch and use the estimation errors to recover the true selectivity. Let $e_1, e_2, \ldots, e_N$ be the estimation errors of model $m'$ for the batch of query feedback *Qfb* (where $N$ is the number of query feedback tuples in *Qfb*), and let $p(e)$ be the empirical probability distribution of the estimation errors,

$$p(e) = \frac{\text{no. of } e_i \text{ with value } e}{N}.$$

A reasonable encoding of the estimation errors is to encode the empirical distribution $p(e)$ of the errors and encode each error $e_i$ using $\log 1/p(e_i)$ bits. Hence, our MDL score is

$$\begin{aligned} score_{mdl}(m', Qfb) &= k \times bytesize(m') \\ &\quad + \sum_{e \in E} [1 + \log(|e| + 1) + \log(p(e) \times N + 1)] \\ &\quad + \sum_{i=1}^{N} \log \frac{1}{p(e_i)}, \end{aligned}$$

where $E$ is the set of distinct error values (distinct $e_i$'s). The quantity $bytesize(m')$ denotes the amount of memory used to store the histogram set model $m'$ without any compression. The MDL encoding of the model $m'$ therefore requires less space and we approximate the encoding length by $k \times bytesize(m')$, where $k$ is some constant. We set $k$ to 1/4 in our experiments[3]. The distribution $p(e)$ is encoded as a sequence of $\langle e, p(e) \rangle$ pairs. Each error value $e$ requires one bit to encode the sign and roughly $\log(|e| + 1)$ bits to encode the magnitude, since $|e|$ can be zero. Each probability value $p(e)$ requires roughly $\log(p(e) \times N + 1)$ bits.

## 3.4.7 Learning Clique Histograms from Query Feedback

We have described our search algorithm for the best histogram set model. In this section we address the problem of building a set of histograms from a batch of

[3]This value of $k$ is obtained via experimentation.

query feedback given a fixed structure, i.e., the sets of attributes on which to build histograms are fixed. This problem differs from that addressed in [2, 12] in that we consider multi-attribute queries whose query attributes are not covered by a single histogram, whereas [2, 12] assumes that the attributes of each query is covered by exactly one multidimensional histogram. For example, [2, 12] do not address how to build two one-dimensional histograms for attributes $A_1$ and $A_2$ given two dimensional queries on attributes $A_1$ and $A_2$. We show how to build low-dimensional histograms from high-dimensional queries using the delta rule [70].

A histogram $h$ that approximates the frequency distribution of a set of attributes $\mathit{attributes}(h)$ consists of a set of buckets with index set $B(h) = \{1, 2, \ldots, \mathit{nbkts}(h)\}$. Each bucket $i$ corresponds to a particular partition $\mathit{box}(i)$ of the domain space for $\mathit{attributes}(h)$ and is associated with $freq(i)$, the frequency count of tuples that occur in that partition. The shorthand $c_i$ for $freq(i)$ will be used in mathematical formulas.

The selectivity of a range query $q = \mathbf{X}{\in}\mathbf{r}$, such that $\mathbf{X} \subseteq \mathit{attributes}(h)$, is given by the sum of the counts of all the buckets that overlap with the range constraints,

$$\hat{\sigma}(\mathbf{X}{\in}\mathbf{r}) = \sum_{i=1}^{\mathit{nbkts}(h)} \alpha_i \times c_i,$$

where $\{1, \ldots, |B(h)|\}$ is the set of bucket indices for histogram $h$, and each $\alpha_i$ is the fraction of overlap between the bucket $i$ and the query range,

$$\alpha_i = \frac{\mathit{box}(i) \cap \mathbf{r}}{\mathit{box}(i)}$$

If the query attributes $\mathbf{X}$ is a proper subset of $\mathit{attributes}(h)$, the attributes that are not in the query are marginalized out in the same way as computing a marginal

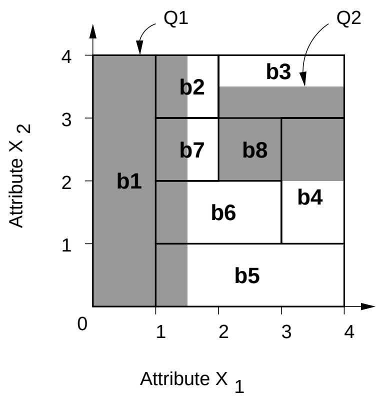


**Figure 3.6**: An example of a two dimensional histogram.

probability distribution (Equation (3.3.2)).

**Example 3.6** *Consider the one-dimensional query, $Q_1 = X_1{\in}[0, 1.5]$, the two-dimen-sional query, $Q_2 = X_1{\in}[2, 4], X_2{\in}[2, 3.5]$, and the two-dimensional histogram for the attributes $\{X_1, X_2\}$ in Figure 3.6. The shaded regions in the figure denote the query regions for $Q_1$ and $Q_2$.*

*We compute the selectivity of the query $Q_1$ as,*

$$\hat{\sigma}(X_1{\in}[0, 1.5]) = c_1 + \frac{1}{2}c_2 + \frac{1}{2}c_7 + \frac{1}{4}c_6 + \frac{1}{6}c_5.$$

*The selectivity of $Q_2$ is computed as,*

$$\hat{\sigma}(X_1{\in}[2, 4], X_2{\in}[2, 3.5]) = c_8 + \frac{1}{2}c_4 + \frac{1}{2}c_3.$$

If the set of query attributes is not a subset of any histogram attributes, the query selectivity needs to be computed using the graphical model. In our case, a junction tree model $J(S) = (V_J, E_J)$ is used. Let $V' = \{v \in V_J : v \cap \mathbf{X} \neq \emptyset\}$ be the set of cliques that contain some of the query attributes and $E' = \{(u, v) \in E : v, u \in V'\}$ be the set of edges that have the nodes in $V'$ as endpoints. The selectivity of the range query $\mathbf{X} \in \mathbf{r}$ can be computed as,

$$\hat{\sigma}(\mathbf{X} \in \mathbf{r}) = \frac{\prod_{u \in V'} \hat{\sigma}(C_u)}{\prod_{(u,v) \in E'} \hat{\sigma}(C_u \cap C_v)}. \tag{3.4.7}$$

Each numerator term $\hat{\sigma}(C_u)$ is computed from the clique histogram for attribute set $C_u$ at the values specified in $\mathbf{r}$, and each denominator term $\hat{\sigma}(C_u \cap C_v)$ is computed from the clique histogram for $C_u$ by marginalizing out the attributes in $C_u - C_v$ and setting the values of the attributes in $C_u \cap C_v$ with corresponding values in $\mathbf{r}$. Since each term is computed from a histogram, it is a sum of bucket frequencies. The buckets that are included in the sums are those that overlap with the query range $\mathbf{r}$.

Suppose that the partitioning of the domain space by a clique histogram is fixed and we want to update the frequency count $c_b$ of a particular bucket $b$ of that histogram in response to an estimation error in a query $\mathbf{X} \in \mathbf{r}$. Let $\sigma$ be the true selectivity of this query and $\hat{\sigma}$ be the selectivity of this query estimated by the model. The frequency for bucket $b$ can appear at most once in the numerator and possibly many times in the denominator (in the following formula $l$ times); hence, the estimated selectivity (Equation (3.4.7)) can be expressed in terms of $c_b$ as

$$\begin{aligned} \hat{\sigma} &= \frac{\alpha_0 c_b + k_0}{\prod_{i=1}^{l} (\alpha_i c_b + k_i)} \times \rho & (3.4.8) \\ &= \frac{p_0}{\prod_{i=1}^{l} p_i} \times \rho & (3.4.9) \end{aligned}$$

where $\rho$ denotes the rest of the numerators and denominator terms that do not involve the bucket $b$, each $\alpha_j$ denotes the fraction of overlap of bucket $b$ with the query range, each $k_j$ denotes the rest of the bucket frequencies being summed, and each $p_j$ is a shorthand for $\alpha_j c_b + k_j$. We use the delta rule [70] (gradient descent method) to update the frequency count $c_b$ of bucket $b$ in response to an estimation error $\hat{\sigma} - \sigma$. The delta rule states that for an error function $E(c_b)$, the update to the variable $c_b$ should be proportional to the negative gradient of $E(c_b)$ with respect to $c_b$. Using the delta rule to minimize the squared error,

$$E(c_b) = (\hat{\sigma} - \sigma)^2,$$

we update the bucket frequency $c_b$ in proportion to

$$\frac{\partial E(c_b)}{\partial c_b} = 2(\hat{\sigma} - \sigma)\frac{\partial \hat{\sigma}}{\partial c_b} \tag{3.4.10}$$

where

$$\frac{\partial \hat{\sigma}}{\partial c_b} = \frac{\alpha_0 \hat{\sigma}}{p_0} - \sum_{i=1}^{l} \frac{\alpha_i \hat{\sigma}}{p_i}. \tag{3.4.11}$$

Hence, given a query feedback with true selectivity $\sigma$ and estimated selectivity $\hat{\sigma}$, we can update each bucket $b$ that is involved in the selectivity estimation computation. For each bucket $b$, we compute $l$, the number of times the frequency of $b$ is used in the denominator, and we update the frequency of $b$ using

$$\begin{aligned} c_b^{(n+1)} &= c_b^{(n)} - \gamma \frac{\partial E(c_b)}{\partial c_b} \\ &= c_b^{(n)} - 2\gamma(\hat{\sigma} - \sigma)\left[\frac{\alpha_0 \hat{\sigma}}{p_0} - \sum_{i=1}^{l} \frac{\alpha_i \hat{\sigma}}{p_i}\right], \end{aligned} \tag{3.4.12}$$

where $\gamma$ is a tunable parameter called the learning rate, each $p_0$ corresponds to a $\hat{\sigma}(C)$ term (in Equation (3.4.7)) that uses bucket $b$ and each $p_j, j > 0$ corresponds to a $\hat{\sigma}(C_u \cap C_v)$ term that uses bucket $b$ in (3.4.7). For a batch of query feedback, we iterate through the batch a fixed number of times, applying the update rule in (3.4.12) (see Algorithm 3.2).

**Algorithm 3.2** LEARNBUCKETFREQ($\mathcal{B}$, *Qfb*)

```
INPUT: set of buckets $\mathcal{B}$, set of query feedback Qfb, number of update iterations niter
1: for all $i = 1 \dots niter$ do
2:   for all $q \in Qfb$ do
3:     for all $b \in \mathcal{B}$ do
4:       update $freq(b)$ using Equation (3.4.12)
5:     end for
6:   end for
7: end for
```

The update rule in (3.4.12) learns the bucket frequencies of a clique histogram assuming that the set of buckets (i.e., the partition) is fixed. To find a suitable partition, we search in a restricted set of partition schemes for the partition that minimizes the squared error (with respect to a batch of query feedback). In the case of MHIST, we always split a bucket into two equal halves. The region covered by a bucket is $d$-dimensional, and hence there are $d$ possible splits. The algorithm for learning a MHIST histogram (both partitioning and bucket frequencies) from a batch of query feedback is outlined in Algorithm 3.3. Basically, in each iteration of

**Algorithm 3.3** LEARNMHIST($h$, $nB$, *Qfb*)

```
INPUT: a one bucket clique histogram h, the target number of buckets nB, set of query feedback Qfb
1: while $nbkts(h) < nB$ do
2:   for all bucket $b \in h$ do
3:     for all dimension $d$ do
4:       $(b', b'') \leftarrow$ split $b$ at the mid point along $d$
5:       LEARNBUCKETFREQ($\{b', b''\}$, $Qfb$)
6:       $score(b, d) \leftarrow$ score this split
7:     end for
8:   end for
9: end while
10: $(b*, d*) \leftarrow \arg\max_{(b,d)} score(b, d)$
11: apply the split $(b*, d*)$ on $h$
12: $nbkts(h) \leftarrow nbkts(h) + 1$
```

the `while`-loop (line 1), we score each candidate split (line 4-6) and choose the best candidate split to apply on the histogram. A candidate split is completely specified by the bucket to split and the dimension to split. Line 4 does not actually perform the split in the histogram, but simulates a split by returning the two buckets that would have resulted from the split.

## 3.4.8 The Online Tuning Phase

Once the restructuring phase has established a set of histograms that is locally optimal based on the previous batch of query workload, the on-line phase continually tunes the bucket frequencies of this set of histograms to ensure adaptivity to changes in the current query workload.

The on-line update algorithm is based on the update rule given in (3.4.12) and is outlined in Algorithm 3.4. Given a query feedback $\langle q, \sigma(q) \rangle$, we update all the histograms that are involved in the computation of the selectivity estimate $\hat{\sigma}(q)$. The complexity of the on-line update algorithm is bounded by the size of the set

**Algorithm 3.4** OnlineUpdate($\langle J(S), \mathcal{H}, B \rangle$, $\langle q, \sigma(q) \rangle$)

```
INPUT: Query feedback $\langle q, \sigma(q) \rangle$, current histogram set model $\langle J(S), \mathcal{H}, B \rangle$
1: for all $h \in \mathcal{H}$ involved in q do
2:     for all bucket $b \in h$ involved in q do
3:         update $freq(b)$ using Equation 3.4.12
4:     end for
5: end for
```

of histograms, which reside in a small constant amount of memory space. The overhead of the on-line update is therefore not a significant cost compared to the potential improvement in estimation accuracy.

## 3.5 Experimental Evaluation

In this section, we describe our experiments and present experimental results. We present results for a real data set from the US census.

**Real Data.** We use data from the Current Population Survey (CPS) of the US Census Bureau (www.census.gov) as in [26]. This data set consists of the following attributes from the Person Data Files of the March (2001) Supplement: race, native country of sample person, native country of mother, native country of father, citizenship and age. There are 128821 tuples, of which 13824 are distinct.

**Query Generation.** Our query workload generator takes as input a pair of normal distribution parameters $(\mu, \sigma)$ for each attribute $A_i$, where $\mu$ is the most frequently queried value of $A_i$ and $\sigma$ is the standard deviation that controls how skewed the query workload is going to be. For our experiments, each $\mu$ is randomly chosen. A query is generated by randomly picking the relation $R$ to be queried, the number of attributes $d$ to be queried, and the $d$ attributes from relation $R$. For each of the $d$ attributes, the low endpoint is picked randomly according to the normal distribution specified for the attribute. The high endpoint is picked using the uniform distribution from between the low endpoint and the maximum value for that attribute. The true selectivity of the query is evaluated using the actual data distribution. For our experiments, we generated workloads of 5000 queries each using the skew parameters from Table 3.1. All the queries have positive selectivity.

**Comparisons.** We compare the performance of five algorithms. **DBHist** is the method proposed in [27] that we extended to optimize the bucket allocation over all the clique histograms in the database. **SASH-avi** assumes attribute value independence and builds one MHIST histogram for each attribute. **SASH-sat** ('saturated') builds one MHIST histogram for each relation. Both **SASH-avi** and **SASH-sat** use our method to learn the MHIST and optimize the bucket allocation using the 2-norm absolute error score except that they assume a fixed structure $S$. In the case of **SASH-avi**, each clique contains one attribute, and in the case of **SASH-sat**, each clique contains all the attributes in a relation. **SASH-mdl** uses our search algorithm with the MDL scoring function. **SASH-abs2** uses our search algorithm with the 2-norm absolute error scoring criterion. In our implementation of the restructuring phase of the SASH algorithms, we have used a grid histogram to extract information from the query workload first. The LearnMHIST algorithm (Figure 3.3) then uses the grid histograms to compute the bucket frequencies of the split buckets.

**Error Measures.** We measure the performance of a set of histograms by their estimation errors on a query workload. We use the 1-norm and 2-norm of the absolute errors, where the p-norm absolute error is defined as,

$$||e^{abs}||_p = \left( \frac{1}{N} \sum_{i=1}^{N} |\hat{\sigma}_i - \sigma_i|^p \right)^{1/p}.$$

Absolute errors vary over different datasets. To make the results more meaningful we normalize the {1,2}-norm absolute errors using the largest selectivity in the query workload [81].

## 3.5.1 Restructuring Phase Performance

We evaluate how well the restructuring phase is able to learn the data distribution from a batch of query feedback. For a given query workload, we run the restruc-

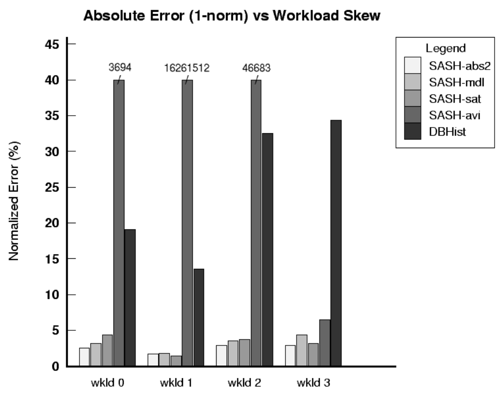
| Census Data Attribute | domain size | wkld 0 | wkld 1 | wkld 2 | wkld 3 |
|---|---|---|---|---|---|
| race | 4 | Uniform | 0.5 | 0.25 | 0.125 |
| native country of person | 113 | Uniform | 10 | 5 | 2.5 |
| mother's native country | 113 | Uniform | 10 | 5 | 2.5 |
| father's native country | 113 | Uniform | 10 | 5 | 2.5 |
| citizenship | 5 | Uniform | 0.5 | 0.25 | 0.125 |
| age | 91 | Uniform | 9 | 4.5 | 2.25 |

**Table 3.1**: The skew parameters (standard deviation) used for generating the four multidimensional query workloads for the restructuring phase.

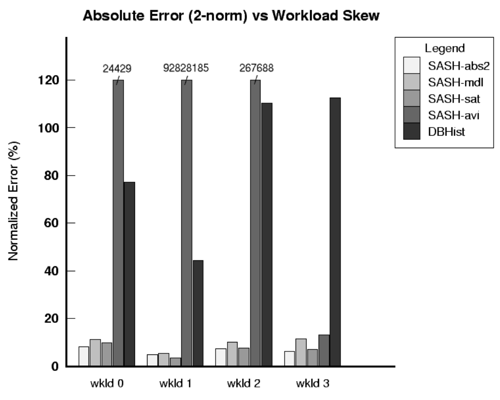
turing phase and measure the estimation errors for that given workload when the restructuring phase has completed.

**Varying workload skew.** We vary the skew levels (the standard deviation parameter in workload generation) of the workload to determine the performance of the restructuring phase over workloads of different skew levels. Workload 0 has query ranges drawn uniformly from the domain space. Workloads 1, 2, and 3 have query ranges drawn using a Gaussian distribution with decreasing standard deviation (hence increasing skew). Figure 3.7 shows the performance of SASH with 1500 bytes of memory over the four different workloads. SASH is consistently more accurate than the off-line **DBHist** over the different workload skew levels. Moreover, **SASH-abs2** is always better than using the AVI assumption (**SASH-avi**).

**Varying memory constraint.** For workload 3, Figure 3.8 shows the performance of SASH as the memory constraint is varied. SASH is consistently better than **DBHist** regardless of memory constraints.



(a) 1-norm



(b) 2-norm

**Figure 3.7**: Normalized absolute error performance for four query workloads with increasing workload skew. All methods are given 1500 bytes of memory. Workload 0 has queries drawn uniformly from the census data and workload 1, 2, and 3 have queries that are increasingly skewed.

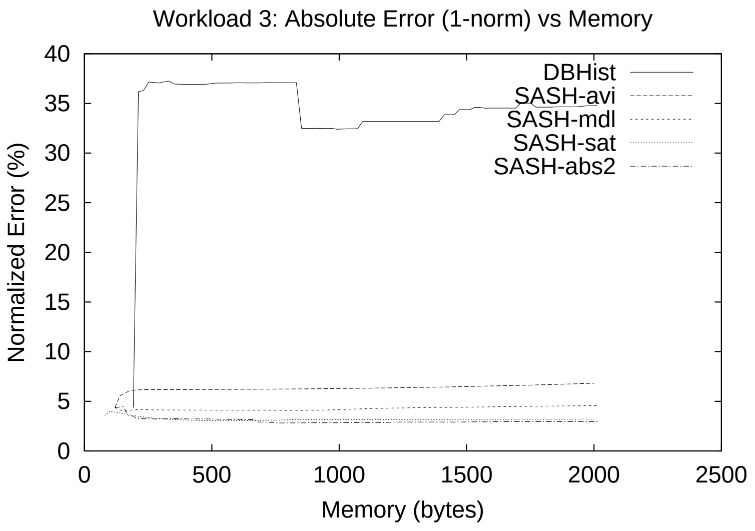
## 3.5.2 Online Tuning Phase Performance.

We evaluate the on-line tuning phase by the estimation errors of the set of histograms over workloads that differ in varying degrees from the workload used in the restructuring phase. The similarity of two workloads $Q_1$ and $Q_2$ are measured by the volume of intersection of the minimum bounding (hyper-)rectangles (MBRs) of the two workloads divided by the volume of the union of the two MBRs,

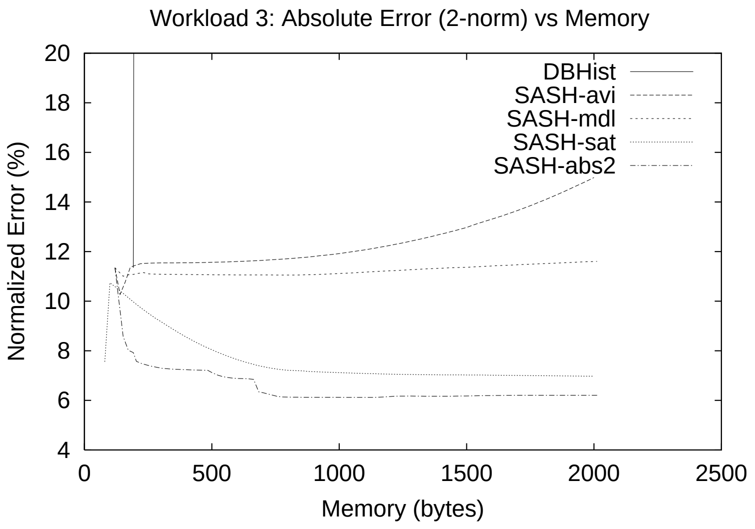
$$similarity(Q_1, Q_2) = \frac{vol(MBR(Q_1) \cap MBR(Q_2))}{vol(MBR(Q_1) \cup MBR(Q_2))}.$$

We can think of a multi-attribute range query as a hyper-rectangle in high dimensional space. The MBR of a workload is therefore the smallest hyper-rectangle that encloses all the query hyper-rectangles in the workload.

We run the on-line tuning phase using the histogram sets that were obtained



(a) 1-norm



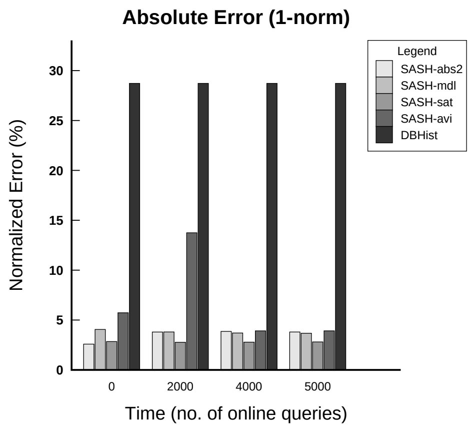
(b) 2-norm (zoomed in)

**Figure 3.8**: Normalized absolute error performance on workload 3 for different memory constraints. Note that the vertical ordering of labels in the legend is the same as the vertical ordering of the curves for memory greater than 200 bytes. We have truncated the **DBHist** curve and zoomed in on the SASH curves in the 2-norm plot to show the relative performance of the SASH methods.

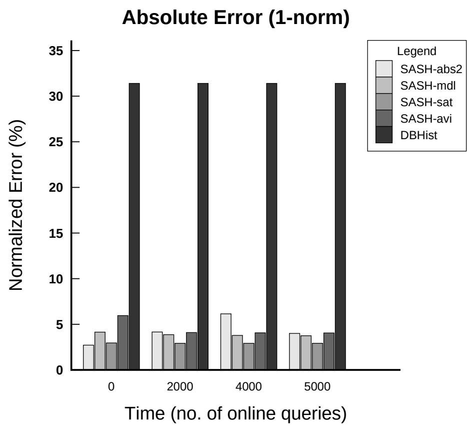
by running the restructuring phase on workload 2 of our restructuring phase experiments. All the histogram sets are roughly 2000 bytes in size. The learning rate used is 0.0001. Three on-line workloads of 5000 queries each were generated with the same skew parameters as the training workload. The similarity scores of the on-line workloads with respect to the training workload are 66% for workload 1, 89% for workload 2, and 91% workload 3. We measure the 1-norm of the absolute errors (average absolute errors) over the *entire* on-line workload as SASH performs on-line tuning in response to the estimation error of each query in the on-line workload. Figure 3.9 shows the on-line performance of SASH for four time slices. Time slice 0 gives the performance before any on-line tuning has been performed. Time slices 2000, 4000, and 5000 give the performance after the on-line tuning phase has seen 2000, 4000, and 5000 queries respectively. Note that since **DBHist** is static, its on-line performance does not improve with time. SASH consistently outperforms **DBHist** over the different on-line workloads. We have found that the histogram sets obtained from **SASH-mdl** restructuring phase tend to perform better than those from **SASH-abs2** during on-line tuning. The on-line performance of **SASH-abs2** tends to fluctuate more (as evidenced in Figure 3.9) compared to **SASH-mdl**. **SASH-mdl** seems to be better at capturing the correlations between attributes. Once correlations are sufficiently captured in the multidimensional histograms, on-line tuning is very effective in tuning the histograms to decrease the overall estimation errors (see Figure 3.10).

## 3.6 Conclusion

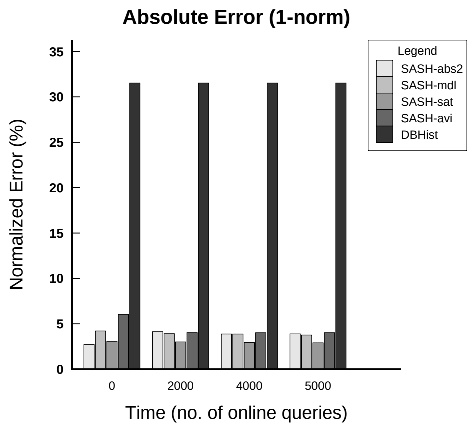
In this chapter we have proposed SASH, a self-adaptive set of histograms, that addresses the problem of which attribute sets to build histograms on, the problem



(a) Online workload 1 (66% similar)



(b) Online workload 2 (89% similar)



(c) Online workload 3 (91% similar)

**Figure 3.9**: Online performance of SASH for three on-line multidimensional query workloads that differ in varying degrees from the workload used in the restructuring phase.

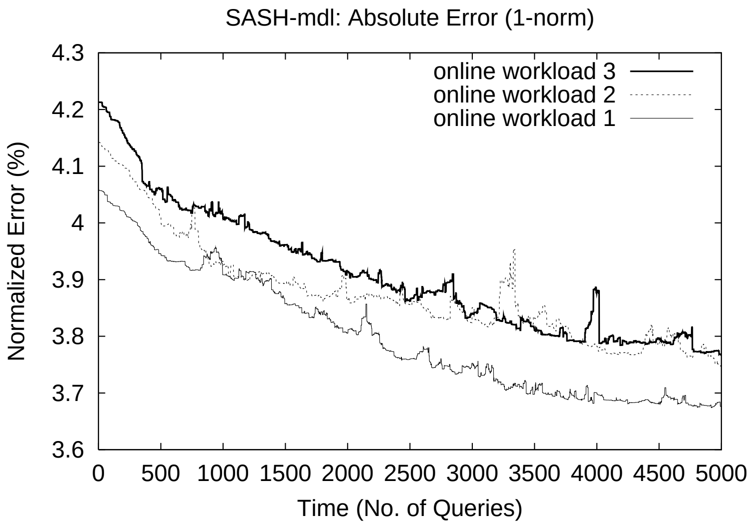


**Figure 3.10**: Online performance of **SASH-mdl** for the three on-line multidimensional query workloads. Online tuning is very effective for decreasing the overall errors, when correlations are adequately captured by the multidimensional histograms.

of allocating memory to a set of histograms, and the problem of tuning a set of histograms to the query workload simultaneously.

SASH is a two-phase method for the on-line construction and maintenance of a set of histograms for multidimensional queries. In the on-line tuning phase, SASH uses the *delta rule* [70] to tune the current set of histograms in response to the estimation error of each query. The estimation error is computed from the true selectivity of a query obtained from the query execution engine, a *query feedback*. In the restructuring phase, SASH searches for a new and more accurate set of histograms to replace the current set of histograms. We have used graphical statistical models to model a set of histograms with memory constraints and search for the best model given a batch of query feedback. The best model found by SASH includes both the optimal set of histograms and the corresponding optimal memory allocation for each histogram. In other words, SASH addresses both the problem of finding the best attribute sets to build histograms on and the problem of finding the best memory distribution (of the given amount of memory) among the histograms. In contrast to previous model search methods [27, 36], SASH does not require access to the database relations to evaluate the candidate models (sets of histograms), but evaluates each candidate using only query feedback information from a query workload. The restructuring phase can be activated periodically or as needed when performance degrades. In summary, our contributions are:

- We develop a new method to build and maintain an optimal set of histograms using only query feedback information from a query workload, without accessing the base data. Because our method is dependent only on query feedback, it is able to adapt to workload and data distribution changes.
- We propose a unified framework that addresses the problem of which attribute sets to build histograms on, the problem of allocating memory to a set of histograms, and the problem of tuning a set of histograms to the query workload.
- For a multidimensional query involving attributes spanning several histograms, we show how to perform on-line updates of the relevant histograms in a principled manner using the delta rule [70].

# Chapter 4

# XPathLearner for Selectivity Estimation of XPath Queries.

## 4.1 Introduction

The extensible mark-up language (XML) [8] is becoming ubiquitous as a data exchange and storage format especially on the World Wide Web. Almost all commercial RDBMSs include some support for XML data; other systems such as Xyleme [86], Niagara [63] and Lore [37] are specially designed to store and query XML data on the web. Similar to relational data, queries over XML data require accurate selectivity estimation in order to optimize query execution plans. In this chapter, we study a particular selectivity estimation problem for XML database systems : How to estimate the selectivity of XML path expressions using statistics collected from query feedback.

### 4.1.1 Query Optimization in XML Databases

XML data are conceptually trees where each internal node is labeled with a *tag name* and leaf nodes are associated with *data values*. Nodes in an XML tree are often located using *paths*. A path is a sequence of tag names that specify a navigational trajectory to a set of nodes in the XML tree. A rooted path is one that begins at the root node of the XML tree. Paths are described using the XPath [22] language. Using XPath, tags in a path are delimited using the symbol '`/`' and the special delimitor '`//`' denotes a wildcard that can match any subpaths. The relationship between a data value at a leaf node and the tag of the parent of the leaf node is often denoted using the symbol '`=`' in path expressions. In addition to the *linear* path expressions we have described, XPath can also be used to describe *branching* path expressions (also known as *twigs*).

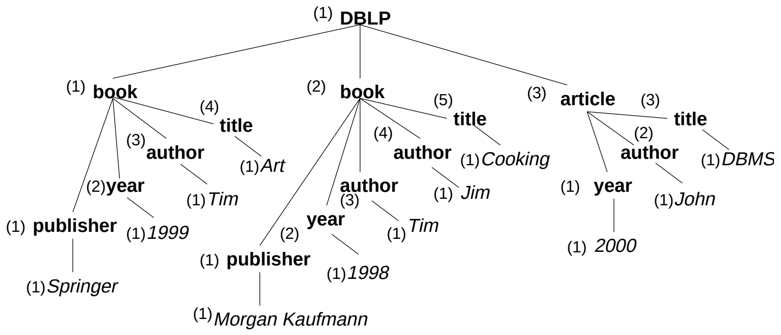


**Figure 4.1**: An example XML data tree. Node identifiers (unique among siblings) are in parenthesis, tag names are in bold and data values are in italics.

**Example 4.7** ***(XML tree and paths.)*** *Consider the XML tree in Figure 4.1. For our example XML tree, we adopt the Dewey scheme [76] for assigning identifiers to each node: the dewey identifier (ID) of a node consists of the identifiers of all its ancestors. Each node has an unique dewey identifier and is labeled with a tag or associated with a data value. Node* (1.1.1) *is an internal node labeled with tag* ***`publisher`*** *and node* (1.1.1.1) *is a leaf node associated with data value* ***`Springer`****. The path expression* ***`/DBLP/book/author`*** *is a rooted path that resolves to the node set* $\{(1.1.3), (1.2.3), (1.2.4)\}$*, while the path expression* ***`//author`*** *is not a rooted path and resolves to the node set* $\{(1.1.3), (1.2.3), (1.2.4), (1.3.2)\}$*. The path expression* ***`//author=Tim`*** *resolves to the leaf nodes* $\{(1.1.3.1), (1.2.3.1)\}$*. The branching* *path expression* ***/DBLP/book[author=Jim]/title*** *resolves to* $\{(1.2.5)\}$ *and specifies a node reachable via* ***/DBLP/book/title*** *whose parent node with tag* ***book*** *has a branch matching the path expression* ***author=Jim****.*

XML databases store XML data using tree-like data structures (e.g., Lore [37]). Queries are expressed using an XML query language such as XQuery [13] and are usually composed of constraints on path expressions specified by XPath. For efficient processing, complex path expressions in XML queries are often preprocessed into a set of candidate $\langle path, pred\rangle$ query pairs, where *path* is a linear rooted path and *pred* is a string predicate on the leaf value reachable via *path*. Consequently, an XML query (such as XQuery) can be mapped to several retrieval operations using $\langle path, pred\rangle$ query pairs [44, 76, 87, 89]. These retrieval operations using $\langle path, pred\rangle$ query pairs form the set of basic query processing operators. Accurate estimation of the selectivity of such $\langle path, pred\rangle$ query pairs is therefore crucial for choosing an optimal execution plan in cost-based query optimization [61]. In this chapter, we will deal only with $\langle path, pred\rangle$ query pairs containing the exact match string predicate and we will use the shorthand, *path=string* to denote the query pair. Chapter 5 presents a method that will handle substring predicates as well.

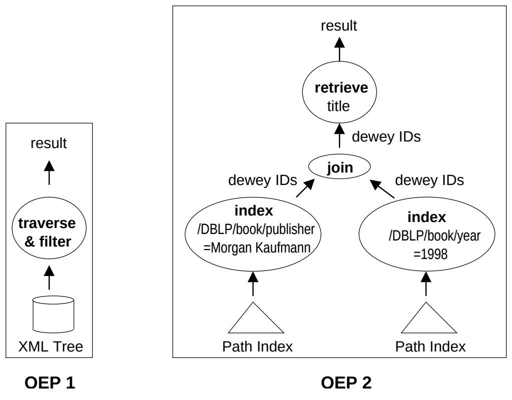
**Example 4.8** ***(XML query processing.)*** *Consider the XML tree in Figure 4.1. Suppose we want to find the titles of all books published by Morgan Kaufmann in the year 1998. (The answer is the book title "Cooking".) Our query can be expressed in the XQuery language [13] as:*

```
FOR $b IN document("*")//book
WHERE $b/publisher = "Morgan Kaufmann"
  AND $b/year = "1998"
RETURN $b/title
```

*The XQuery function* `document("*")` *indicates that all XML documents in the repository should be searched for the path expression* `//book`*. The XQuery specifies a particular set of nodes (represented by variable* `$b`*) be found and the data value associated with the leaf node descending from each child labeled with* `title` *be returned. Each node in the required set satisfies the following constraints:*

1. *it is reachable by the path expression* `//book`
2. *it has a child labeled* `publisher` *with leaf node having value* `Morgan Kaufmann`*.*
3. *it has a child labeled* `year` *with leaf node having value* `1998`*.*

*We sketch two possible query execution plans (QEPs) for this query:*



*In both QEPs the path expressions have been resolved to rooted paths in a preprocessor. QEP 1 traverses to all nodes reachable by the rooted path* `/DBLP/book` *in the XML tree and checks for the other constraints. QEP 2 assumes a path index that returns a set of Dewey IDs of nodes reachable via a given* $\langle path, pred\rangle$ *query pair. To evaluate the cost of these QEPs, estimates for the cardinality of the node sets reachable by linear path expressions such as,*

- `/DBLP/book`*,*
- `/DBLP/book/publisher=Morgan Kaufmann`*, and*

- */DBLP/book/year=1998*

*will be required.*

### 4.1.2 Selectivity Estimation in XML Databases

As in relational databases, cost-based query optimization in XML also requires cost evaluation of QEPs and cost evaluation of QEPs in XML databases require estimating the selectivity of path expressions. This chapter deals with the selectivity estimation of three commonly used types of path expressions in XML queries.

1. *Simple path expressions* are linear path expressions consisting of tags only (e.g., `//book/title`).
2. *Single-value path expressions* are Linear path expressions ending in a data value (e.g., `//book/year="1998"`).
3. *multi-value path expressions* are (semi-)linear path expressions with multiple tag-value bindings (e.g., `//chapter="2"/section="3"`).

Selectivity estimates of path expressions are usually computed using statistics about the structure of the XML data. The main challenges in collecting and storing these statistics are:

- How to obtain the structure of the XML data? All previous work scans the entire XML repository in an off-line manner [1,61]. However, off-line scans are often not possible or feasible in Internet-scale applications since Internet-scale repositories are either inaccessible or too large to be scanned entirely.
- How to capture the statistics for the selectivities of different types of XML path expressions using a small amount of memory? State-of-the-art techniques proposed in [1, 17] are unsatisfactory either because they are limited

to the selectivity of simple path expressions only [1] or they are not space efficient [17].

- How to use the limited storage space in the most effective way? Ideally, more storage resource should be spent on storing the statistics that are relevant to the most frequently queried portions of the XML repository. All previous work are oblivious to workload distribution and consequently waste precious storage space in storing statistics of infrequently queried portions of the repository.

- How to incrementally update the statistics when the underlying XML data change? The XML repositories in Internet scale applications are constantly changing. To ensure accurate XML path selectivity estimation, the statistics must keep up with the change. However, the off-line periodic scan used by previous work to obtain new statistics is neither effective not efficient because of the scanning cost associated with the size of the repositories.

We propose XPathLearner to address these challenges. XPathLearner is a novel on-line learning method for estimating the selectivity of XML path expressions. Our XPathLearner assumes a Markov model [1, 61] of path selectivities and learns this model from query feedback using error reduction strategies. Two such strategies are presented: the heavy-tail rule and the delta rule. XPathLearner overcomes the limitations of previous work and has the following properties:

- Instead of scanning the XML data, XPathLearner collects the required statistics in an on-line manner from *query feedback* information (see Figure 4.2).

- XPathLearner learns both tag distribution and value distribution from query feedback. It is designed to estimate the selectivity of all three types of common path expressions. It can also estimate the selectivity of a path expression

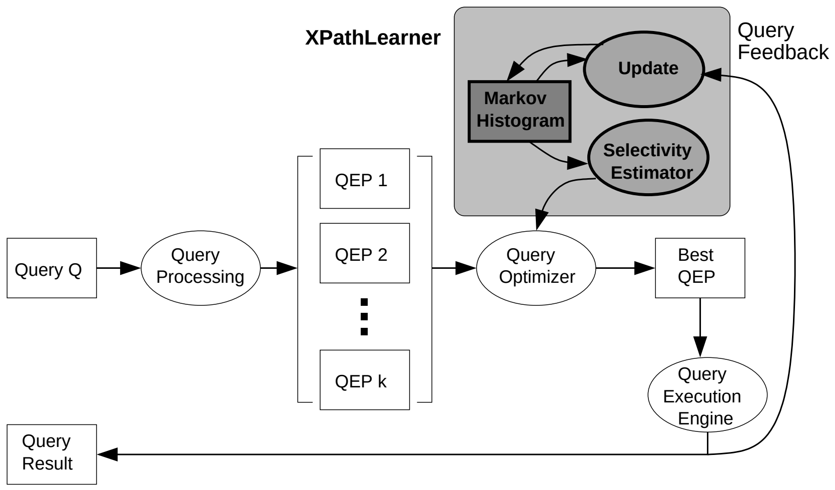


**Figure 4.2**: Schematic of XPathLearner. The top path is for query processing and feedback loop is for workload-driven selectivity estimation processing.

containing a simple wildcard.

- XPathLearner is workload-aware in collecting the required statistics. The allocated storage space is used in the most effective way since more statistics are collected for more frequently queried portions of the XML data.

- XPathLearner automatically adapts to changing XML data, because the statistics are refined on-line according to the most current query feedback. XPathLearner incurs a small overhead in updating the statistics using query feedback, but this cost is offset by an increase in estimation accuracy.

Since query feedback provides only partial information about the path selectivity distribution, we would expect that an on-line method using query feedback to be less accurate than an off-line method. In our experiments we show that not only does XPathLearner come close in accuracy to the off-line method in general, but sometimes surpasses the off-line method because of its workload-driven nature.

The rest of the chapter is organized as follows. In the next section, we review related work. Section 4.3 introduces the terms used in our description of the on-line XML path selectivity estimation problem. Section 4.4 presents the XPathLearner: the Markov chain model for path expressions, the Markov histogram for storing the model parameters, two approaches for dealing with single-value path counts, and two learning strategies. We present our experimental evaluation in Section 4.5 and draw conclusions in Section 4.6.

## 4.2 Related Work

To estimate the selectivities of XML path expressions, the Lore system stores statistics of all distinct paths up to length $m$, where $m$ is a tunable parameter [61]. Selectivity of paths longer than $m$ are estimated assuming the Markov property (see Equation (4.4.5) of Section 4.4.1). The paths stored include both tags and data values and no further summarization is performed. The space requirement of the statistics used in the Lore system is therefore prohibitively large, because the number of possible data values in a big XML repository can be extremely large and the number of distinct paths with data values can therefore be even larger. Our XPathLearner presents two different approaches to address this problem: storing the counts of paths with data values in a compressed histogram [73], or, alternatively, evicting stored entries based on certain criteria (similar to a cache).

Aboulnaga et al. extended the idea used by the Lore system in their *Markov table* method [1]. The Markov table method consists of a set of pruning and aggregation techniques on the statistics used in the Lore system. One limitation of the Markov table method is that paths with data values are not considered (i.e., it can only estimate the selectivity of simple path expressions). This limitation is serious, because the selectivities of paths with data values are crucial in optimizing XML queries that have large top-down search space and highly selective data values. For such queries, a bottom-up search plan is more cost-effective than a top-down search [61]. For the XML data in Figure 4.1, the query “find the titles of all books authored by Jim” is an example. The path expression `//author="Jim"` is more selective than `//book`. Our method aims to overcome this limitation by storing statistics for paths with data values while keeping the space requirement low.

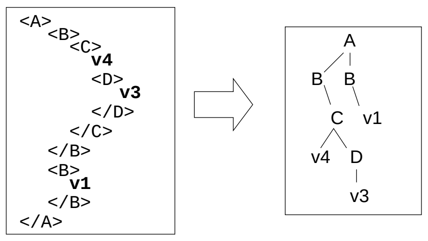
Aboulnaga et al. also proposed a tree-based method known as the *path tree* method [1] for estimating the selectivity of XML paths without data values. A path tree is a summarized form of the XML data tree where sibling nodes of the same tag are aggregated. Tree pruning and aggregation techniques are proposed to reduce the space requirement of path trees. Their experiments, however, have shown that the path tree method is inferior to the Markov table method for real data sets.

Chen et al. proposed another off-line tree-based method for estimating XML subtree selectivity [17]. A suffix tree based data structure is used to store the statistics of the XML data obtained from scanning the repository. Pruning and aggregation techniques are proposed to compress this data structure. However, the space requirement of their summarized data structure is especially large for XML data with long data values. Subtree selectivity estimation involves estimating the selectivity of a query that matches a subtree in the XML data tree as opposed to matching a single path. The problem of subtree selectivity estimation is significantly more general than the path selectivity problem that our method addresses, but even if the technique in [17] is modified for path expressions containing tags only, Aboulnaga et al. have shown that their Markov table method is superior in accuracy [1].

Polyzotis et al. have recently proposed XSketch a more general model based framework [66, 67], however, the construction of an XSketch model requires a greedy heuristic search that evaluates a candidate model against the XML data (or a summarized form of it). XPathLearner is much more lightweight, and builds and adapts itself using only query feedback information.

The XML path selectivity estimation methods proposed in [1, 17, 61, 66, 67] all require information from scanning the repository. These off-line methods share several limitations. The requirement of an off-line scan limits the use of these methods on large (especially Internet-scale) repositories. They are not tuned to the workload distribution: the workload may only query a small portion of the XML data and hence a small portion of the statistics stored in the allocated space. The repository needs to be rescanned whenever the data in the repository change sufficiently. Our XPathLearner overcomes these limitations by learning the statistics from query feedback in an on-line manner. Keeping statistics gathered from query feedback ensures that the allocated space is used to store statistics that are up-to-date and relevant to the query workload.

Selectivity estimation using statistics gathered from query has been proposed in [2, 12] for relational data. Tree-structured data such as XML present new challenges. Whereas the self-tuning histograms of [2,12] capture continuous distribution over numeric attributes, a corresponding self-tuning XML path selectivity estimator needs to capture a discrete distribution over a set of non-numeric path labels. In the continuous case, self-tuning histograms (such as [2, 12]) can start with a uniform distribution over a large interval and *refine* this distribution by creating finer partitions of this interval. In contrast, a self-tuning XML path selectivity estimator does not have an interval to start with and needs to *learn* each and every path label



```
<A>
   <B>
     <C>
       v4
       <D>
         v3
       </D>
     </C>
   </B>
   <B>
     v1
   </B>
</A>
```

**Figure 4.3**: An XML document and its corresponding tree representation.

in the data. Even if a Markov model is imposed on the tree data to simplify the distribution entailed by the tree data, it has been shown that learning a Markov model can be hard [47].

In the off-line XML path selectivity estimation domain, Aboulnaga et al. [1] and McHugh et al. [61] use estimation techniques based on the Markov model. An order $m - 1$ Markov model assumes that the selectivities of all the paths whose lengths are less than or equal to $m$ capture all the required statistics. The experiments in [1] show that, in practice, first and second order Markov models are sufficient to capture the path selectivity statistics with little loss in information. Our method assumes the Markov model, but differs from previous work in that our method (1) gathers statistics in an on-line manner without scanning the repository, (2) handles the three types of common query path expressions, and (3) is workload-aware.

## 4.3 Preliminaries

In this section, we introduce basic terms and notation used in describing the XML path selectivity problem. In particular we introduce the different types of path expressions and define their corresponding selectivities.

An XML document is structurally a tree (we ignore IDREFs) where each node is associated with a tag or a value. In practice, values are almost always associated

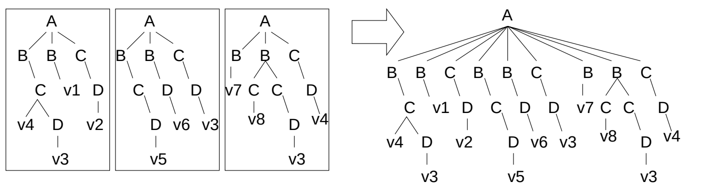


**Figure 4.4**: The XML data tree (right) is constructed from a repository of three XML documents (left). Alphabets in upper case denote tag names, 'v' followed by a number denotes a data value. The selectivity of the simple path expression //B/C/D is 3, the selectivity of the single-value path expression //B/C/D=v3 is 2, the selectivity of the multi-value path expression //B/C=v4/D=v3 is 1, and the selectivity of //A/*/D is the sum of the selectivities of //A/B/D and //A/C/D, which yields 4.

with leaf nodes. An XML *data tree* is a huge tree constructed either by merging the roots of all the XML documents if the tag associated with the root of each document is the same or by introducing a *super root* as the parent of the root node of each XML document. An XML data tree represents a repository of XML documents (see Figure 4.4).

A *simple path expression* $p$ of length $n$ is a sequence of tags $\langle t_1, t_2, \ldots, t_n \rangle$, $t_i \in \Sigma$, where $\Sigma$ is the set of all possible tag names. The tag sequence in a path expression encodes a navigational path through the XML data tree where each pair in the sequence $(t_i, t_{i+1})$ correspond to a (directed) edge with the tags $(t_i, t_{i+1})$ in the XML data tree. Using XPath notation, such a navigational sequence can also be written as $//t_1/t_2/\ldots/t_n$ (e.g., `//book/title`). We will use as shorthand, where there is no confusion, the string $t_1 t_2 \ldots t_n$ to represent $//t_1/t_2/\ldots/t_n$.

A *multi-value path expression* is a simple path expression where values are associated with one or more tags in the path expression[1]. A special case of a multi-value path expression is a *single-value path expression* where only the last tag in the path is associated with a value. The length of a single-value path expression

[1]A multi-value path expression is a special case of a twig [17].

$//t_1/t_2/\ldots/t_n{=}v_n$ is $n+1$.

Consider the XML data tree shown in Figure 4.4. The path `//B/C/D` is a simple path expression; the path `//B/C=v4/D=v3` is a multi-value path expression, and the path `//B/C/D=v3` is a single-value path expression.

We denote the selectivity of a (simple, multi-value or single-value) path expression $p$ as $\sigma(p)$. The selectivity of a simple path expression $p$ is defined to be the number of paths in the XML data tree that match the tag sequence in $p$. The selectivity of a single-value path expression is similar to that of simple path expressions except that the navigational path ends in a value instead of a tag in the XML data tree. The selectivity of a multi-value path expression $p$ is defined to be the number of subtrees that matches the tag and value sequence in $p$.

The path expressions that we consider in this chapter are allowed to contain one wildcard and we restrict each wildcard ('*') to match a single tag. Moreover we do not consider path expressions beginning or ending with a wildcard[2]. The selectivity of a path expression $p$ with a single wildcard is the sum of the selectivities of all the (non-wildcard) path expressions that are possible matches to $p$. Example selectivities are given in Figure 4.4.

A *query feedback* is a tuple $(p, \sigma(p))$ consisting of a path expression and its corresponding true selectivity. Our definition of query feedback assumes minimal information about the query execution engine. It is possible to obtain more feedback information from the query execution engine. The amount of information we can obtain depends upon the underlying data storage model and the query plan used by the execution engine. For example, using the Lore model and a top-down plan, the query execution engine can provide as feedback the selectivities of all prefixes

[2]We are investigating the extensions for more complicated wildcards in ongoing work.

of the given path. Since XML storage and retrieval technology is still evolving, we assume minimal feedback information in this dissertation.

## 4.4 XPathLearner

We propose XPathLearner, an on-line method for estimating the selectivity of a given path expression (simple, single-value, multi-value), using statistics that are obtained from query feedback only and reside in a given amount of space (memory).

XPathLearner models the selectivity of path expressions as an order $(m-1)$ Markov chain or network. The Markov model has been used by [1, 61] to model the selectivities of all possible paths in the XML data tree. XPathLearner differs from [1, 61] in that only the selectivities of paths that occur in the query workload are modeled. Moreover, in contrast to previous work, XPathLearner learns the parameters associated with the Markov model in an on-line manner using query feedback information only. XPathLearner stores these parameters using a *Markov histogram* and updates the Markov histogram using query feedback.

In this section we describe in detail the Markov model for path expressions, the associated Markov histogram, two approaches to dealing with single-value path counts, the on-line update algorithm, and two update strategies.

### 4.4.1 Modeling Path Expression Selectivity

A simple path expression query of length $n$ can be modeled as a sequence of random variables $\langle X_1 X_2 \ldots X_n \rangle$ (in XPath notation $//X_1/X_2/\ldots/X_n$), where each $X_i$ is instantiated with some tag name from $\Sigma$ the set of all possible tags. The selectivity of an instance of a simple path expression $t_1 t_2 \dots t_n$ can be computed from

$$\sigma(t_1 t_2 \dots t_n) = P(X_1 X_2 \dots X_n) \times N, \tag{4.4.1}$$

where $N$ is the total number of nodes in the XML data tree and $P(X_1 X_2 \dots X_n)$ is a shorthand for $P(\langle X_1 X_2 \dots X_n \rangle = \langle t_1 t_2 \dots t_n \rangle)$, the probability that each $X_i = t_i$ for $i = 1, 2, \dots, n$. An order $(m-1)$ Markov model for the sequence of random variables $X_1 X_2 \dots X_n$ assumes that each random variable $X_i$ is only dependent on its $m-1$ predecessors in the sequence,

$$P(X_1 X_2 \dots X_n) \approx \frac{\prod_{j=1}^{n-m} P(X_j X_{j+1} \dots X_{j+m-1})}{\prod_{i=2}^{n-m} P(X_i X_{i+1} \dots X_{i+m-2})}. \tag{4.4.2}$$

Moreover, the Markov model makes the stationarity assumption,

$$\begin{aligned} &P(\langle X_j X_{j+1} \dots X_{j+m-1} \rangle {=} \langle t_j t_{j+1} \dots t_{j+m-1} \rangle) \\ &\qquad = P(\langle X_i X_{i+1} \dots X_{i+m-1} \rangle {=} \langle t_j t_{j+1} \dots t_{j+m-1} \rangle), \end{aligned} \tag{4.4.3}$$

for any integers $i, j \geq 1$. Using the $m-1$ order Markov model, it is sufficient to store $P(X_1 X_2 \dots X_l)$, for $l = 1, 2, \dots m$, in order to approximate the selectivity of any simple path expression. In practice, each $P(\langle X_1 X_2 \dots X_l \rangle = \langle t_1 t_2 \dots t_l \rangle)$ is stored in a table of frequency counts $f(t_1 t_2 \dots t_l)$, where

$$P(X_1 X_2 \dots X_l) = f(t_1 t_2 \dots t_l)/N. \tag{4.4.4}$$

Each $f(t_1 t_2 \dots t_l)$ counts the number of occurrences of the path $t_1 t_2 \dots t_l$ in the XML data tree. Combining Equations (4.4.1), (4.4.2), (4.4.3) and (4.4.4), the selectivity of a simple path expression can be approximated by

$$\hat{\sigma}(t_1 t_2 \ldots t_n) = \frac{\prod_{j=1}^{n-m} f(t_j t_{j+1} \ldots t_{j+m-1})}{\prod_{i=2}^{n-m} f(t_i t_{i+1} \ldots t_{i+m-2})}. \tag{4.4.5}$$

In particular, when $m = 2$, the approximation formula becomes,

$$\hat{\sigma}(t_1 t_2 \ldots t_n) = \frac{f(t_1 t_2) f(t_2 t_3) \ldots f(t_{n-1} t_n)}{f(t_2) f(t_3) \ldots f(t_{n-1})}. \tag{4.4.6}$$

Previous work [1,17,43] has shown that the Markov model works well for many real XML data sets for $m = 2$ and $m = 3$.

The selectivity of single-value path expressions of the form $p = //t_1/t_2/\ldots/t_n = v_n$ can be approximated analogously by associating an additional random variable $X_{n'}$ with the value $v_n$ and instantiating the $X_{n'}$ with the value $v_n$,

$$\hat{\sigma}(//t_1/t_2/\ldots/t_n{=}v_n) = \hat{\sigma}(t_1 t_2 \ldots t_n) \frac{f(t_{n-m+2} t_{n-m+3} \ldots t_n v_n)}{f(t_{n-m+2} t_{n-m+3} \ldots t_n)}. \tag{4.4.7}$$

Similarly, for multi-value path expressions,

$$\hat{\sigma}(//t_1{=}v_1/t_2{=}v_2/\ldots/t_n{=}v_n) = \hat{\sigma}(t_1 t_2 \ldots t_n) \prod_{j=1}^{n} \frac{f(t_{j-m+2} t_{j-m+3} \ldots t_j v_j)}{f(t_{j-m+2} t_{j-m+3} \ldots t_j)}. \tag{4.4.8}$$

## 4.4.2 Markov Histograms

An order $(m-1)$ Markov model requires storing the counts of all length-$l$ simple or single-value paths, i.e., all $f(t_1 t_2 \ldots t_l)$ and $f(t_1 t_2 \ldots t_{l-1} v_{l-1})$, for $l = 1, 2, \ldots m$, in order to approximate the selectivity of any path expression. In the off-line case, storing just the counts of the length-$m$ paths are sufficient, because the count of a length-$l$ path, where $l < m$, can be computed from the counts of all length-$(l+1)$

| | | | |
|---|---|---|---|
| B | 6 | v4 | 2 |
| C | 7 | v5 | 1 |
| D | 7 | v6 | 1 |
| v1 | 1 | v7 | 1 |
| v2 | 1 | v8 | 1 |
| v3 | 3 | | |

Length 1 paths

| | | | | | |
|---|---|---|---|---|---|
| AB | 6 | C=v4 | 1 | D=v4 | 1 |
| AC | 3 | C=v8 | 1 | D=v5 | 1 |
| BC | 4 | B=v1 | 1 | D=v6 | 1 |
| BD | 1 | D=v2 | 1 | B=v7 | 1 |
| CD | 6 | D=v3 | 3 | | |

Length 2 paths

**Figure 4.5**: The first order Markov histogram corresponding the XML data tree in Figure 4.4.

paths,

$$f(t_1 t_2 \ldots t_l) = \sum_{\tau \in \Sigma} f(\tau t_1 t_2 \ldots t_l) \tag{4.4.9}$$

by assuming that the XML data is a tree and that the counts of all the length-$(l+1)$ paths are known. In the on-line case, not all the length-$(l+1)$ path counts have been learnt at a given time and for those that have been learnt, the corresponding counts may not have converged to the true counts. Hence, the counts of paths with length $l < m$, have to be stored for on-line selectivity estimation using a Markov model.

An order $(m-1)$ *Markov histogram* is a table storing a set of distinct paths, with length $l < m$, along with their associated occurrence counts or selectivities. For simplicity of presentation we consider $m = 2$ for the rest of this chapter. An example of a first order Markov histogram is shown in Figure 4.5. Markov histograms approximate the selectivity of simple, single-value, and multi-value path expressions using the formulas in Equations (4.4.5), (4.4.7), and (4.4.8) respectively. When the count of a length-$(l)$ path $(l < m)$ is needed and is not stored in the Markov histogram, a default count of 1 is used.

Storing the counts of single-value path (of length $l < m$) efficiently presents a unique challenge, because the number of distinct data values is typically very large compared to the number of distinct tags in an XML data tree. For example, the

DBLP data set contains 91,878 unique values and only 29 unique tag names. For $m = 2$, this translates to 841 possible tag-tag pairs and 2,664,462 possible tag-value pairs. Given a small amount of memory, it is not possible to store the counts of all length ($l < m$) paths exactly. Two approaches are possible. First, store the all simple path counts exactly and use a form of compressed histogram [73] to store the counts of single value paths. The second approach is not to differentiate between the Markov histogram entries for simple and single-value path expressions, but to delete entries based on some criteria when the Markov histogram runs short of memory (much like in a cache).

### 4.4.3 Compressed Histogram Approach

The large number of XML data values prohibits storing the counts of single-value path expressions with length ($l < m$) exactly. A similar problem is addressed in [49]; however, we adopt a simpler approach. Motivated by the fact that most of the probability mass (of counts) is concentrated in a very small number of single-value paths in many real XML data sets, we use a compressed histogram approach similar to [73].

1. Store the $k$ single-value paths with the largest counts exactly. The parameter $k$ can be tuned. Each entry in the 'top $k$' data structure stores the tuple $\langle \textit{single-value path}, \textit{count} \rangle$.

2. Single-value paths with a count smaller than the minimum count among the top $k$ single-value paths are aggregated into buckets. A bucket is a tuple $\langle \textit{simple path}, \textit{feature}, \textit{sum}, \textit{num} \rangle$, where *simple path* is the tag-only prefix of the single-value path, the field *num* is the number of single-value paths it represents, the field *sum* is the sum of the counts of those single-value paths

assigned to that bucket, and the field *feature* is the feature of the data value corresponding to that bucket. A single-value path is assigned to a bucket based on some feature of the data value in the single-value path.

For example, consider a first-order Markov histogram that uses a compressed histograms to deal with the data values. Further assume that the data values are case insensitive alphanumeric strings and that the first letter of the data value in a single-value path is used as the bucket assignment feature. The compressed histogram for the single-value path counts will then consist of at most $k$ tuples of the form $\langle \textit{single-value path}, \textit{count} \rangle$ and at most $36 \times |\Sigma|$ tuples (where $\Sigma$ is the set of tags) of the form $\langle \textit{simple path}, \textit{feature}, \textit{sum}, \textit{num} \rangle$. Ideally, the feature should be chosen so that the counts in each bucket are as uniform as possible, that is, the variance of the counts represented in a particular bucket should be minimized.

**Retrieval.** Accessing the count of a given single-value path requires searching through the top $k$ entries first. If the required single-value path is not found, the feature of the given single-value path is used to locate the corresponding bucket and the count is computed as $sum/num$.

**Update.** Given a single-value path and an updated count, the top $k$ entries are searched first and if a matching single-value path is found, its count is updated. Otherwise, we check if the updated count of the given single-value path is larger than the minimum count in the top $k$ entries. If it is larger, the minimum entry in the top $k$ is displaced into the bucket corresponding to the displaced entry. If it is smaller, the count of the given single-value path is added to the *sum* field of the corresponding bucket, and the *num* field of the same bucket is incremented. Each bucket therefore encodes the average selectivity of all the current instances of the single-value paths belonging to that bucket.

**Compress.** When memory is scarce, the tag-feature histogram can be further compressed by aggregating buckets with similar selectivities.

### 4.4.4 Cache-based Approach

A different and even simpler approach to keeping the Markov histogram small is to remove or evict entries based on some criteria. Each entry in a Markov histogram stores a simple or single-value path $p$ and its associated count $f(p)$.

The first criterion is to evict entries with counts smaller than a threshold parameter. Recall that our Markov histogram assigns a default value of 1 to entries that are not stored, so removing entries whose counts are close to 1 represents little loss of information.

The second criterion is to evict entries that are seldom used (similar to the least frequently used policy in caching). To keep track of the frequency of use, a counter needs to be associated with each entry in the Markov histogram. To minimize memory overhead, one-byte counters can be used. All the counters in the Markov histogram will need to be reset after every 256 units of use. Each Markov histogram entry will be of the form $\langle \textit{single-value path}, \textit{count}, \textit{counter} \rangle$

In our experiments, whenever memory is short, the small count criterion is first applied, followed by the least frequently used criterion, until sufficient memory has been freed up.

| tag | count |
|---|---|
| A | 1 |
| B | 6 |
| C | 7 |
| D | 7 |

(a) tag counts

| tag-tag | count |
|---|---|
| AB | 6 |
| AC | 3 |
| BC | 4 |
| BD | 1 |
| CD | 6 |

(b) tag-tag counts

| tag=value | count |
|---|---|
| D=v3 | 3 |

(c) top $k$ tag-value counts

| tag | feat. | sum | #pairs |
|---|---|---|---|
| B | a | 1 | 1 |
| B | b | 1 | 1 |
| D | a | 2 | 2 |
| D | b | 2 | 2 |
| C | a | 1 | 1 |
| C | b | 1 | 1 |

(d) tag-feature histogram

**Figure 4.6**: A first order Markov histogram using $k = 1$ and the first letter of the data value as the bucketing feature. Further suppose that the data values {v1, v2, v3, v4} all begin with the letter 'a' and {v5, v6, v7, v8} with the letter 'b'.

## 4.4.5 An Example

We illustrate how selectivity estimation can be done using our Markov histogram with the compressed histogram approach. Consider our earlier example in Figure 4.5. The corresponding representation using our Markov histogram with $k = 1$ is shown in Figure 4.6.

The selectivity of the simple path expression //B/C/D can be estimated by

$$\hat{\sigma}(BCD) \quad = \quad \frac{f(BC)}{f(C)} \times f(CD) = \frac{4}{7} \times 6 = 3.43,$$

which has an absolute error of 0.43. The selectivity of the single-value path expression //B/C/D=v3 can be estimated by

$$\hat{\sigma}(BCD = v3) = \frac{f(BC)}{f(C)} \times \frac{f(CD)}{f(D)} \times f(D = v3) = 1.47,$$

which has an absolute error of 0.53, since the real selectivity is 2. The selectivity of the multi-value path expression //B/C=v4/D=v3 can be estimated by

$$\begin{aligned} &\hat{\sigma}(//B/C = v4/D = v3) \\ &= \frac{f(BC)}{f(C)} \times \frac{f(CD)}{f(D)} \times f(D = v3) \times \frac{f(C = v4)}{\sum_v f(C = v)} = 0.735, \end{aligned}$$

which has an absolute error of 0.265. Note that the value v4 has feature "a". The selectivity of the simple path expression //A/*/D (with wildcard) can be estimated by

$$\hat{\sigma}(A * D) \;\; = \;\; \sum_{\alpha} \hat{\sigma}(A\alpha D) = 3.57,$$

which has an absolute error of 0.43, since $\sigma(ABD) + \sigma(ACD) = 1 + 3 = 4$.

### 4.4.6 On-line Update Algorithms

We describe the two update algorithms that our XPathLearner uses to learn a Markov histogram from query feedback: the heavy-tail rule and the delta rule.

Our two update algorithms follow the high-level steps outlined in Algorithm 4.1 and differ in the update equations used in line 12. The Markov Histogram is assumed to be initially empty. The function `compress-add entry` adds any unknown length-2 path to the Markov histogram: it learns the set of labels of a discrete distribution. A physical entry is not necessarily added to the histogram whenever `compress-add entry` is called. When memory is scarce, `compress-add entry` can trigger pruning or aggregation techniques (such as those in [1]) to compress the histogram. In contrast to learning the labels, the update equation in the algorithm learns the frequency counts of the discrete distribution.

**Algorithm 4.1** UPDATE ($f$, $(p, \sigma)$, $\hat{\sigma}$)

```
INPUT: Markov histogram $f$, Feedback $(p, \sigma)$, Estimate $\hat{\sigma}$
 1: if $|p| \leq 2$ then
 2:    if not exists $f(p)$ then
 3:       compress-add entry $f(p) = \sigma$
 4:    else
 5:       $f(p) \leftarrow \sigma$
 6:    end if
 7: else
 8:    for all $(t_i, t_{i+1}) \in p$ do
 9:       if not exists $f(t_i t_{i+1})$ then
10:          compress-add entry $f(t_i t_{i+1}) = 1$
11:       end if
12:       $f(t_i t_{i+1}) \leftarrow$ update {depends on update strategy}
13:    end for
14: end if
15: for all $t_i \in p, i \neq 1$ do
16:    if not exists $f(t_i)$ then
17:       compress-add entry $f(t_i)$
18:    end if
19:    $f(t_i) \leftarrow \max\{f(t_i), \sum_\alpha f(\alpha t_i)\}$
20: end for
```

## 4.4.7 The Heavy-tail Rule Update Method

Given estimated selectivity $\hat{\sigma}(p)$ and query feedback $(p, \sigma(p))$, where $p = t_1 t_2 \ldots t_n$ and $\sigma(p)$ is the real selectivity, we first compute the observed error

$$\epsilon = \sigma(p) - \hat{\sigma}(p). \tag{4.4.10}$$

Recall that the selectivity of path $p$ is computed as

$$\hat{\sigma}(t_1 t_2 \dots t_n) = \left( \prod_{i=1}^{n-2} \frac{f(t_i t_{i+1})}{f(t_{i+1})} \right) \times f(t_{n-1} t_n). \tag{4.4.11}$$

We need to refine all the $f(t_i t_{i+1})$ terms in this product based on the observed error $\epsilon$. The updates to the $f(t_{i+1})$ terms are dependent on the $f(t_i t_{i+1})$ terms through Equation (4.4.9) and will be described later. We may also want to attribute more of the estimation error to the terms associated with the end of the path $p$. There are two reasons for this: First, the terms closer to the end of the path $p$ are naturally more relevant to the selectivity of path $p$. Second, attributing more of the estimation error to them also minimizes the effect on other paths sharing the same prefix as $p$. We therefore assign weights to the $f(t_i t_{i+1})$ terms that increase with $i$.

Let $w_i$ be the (unnormalized) weight associated with $t_i$ in path $p$. We update the $f(t_i t_{i+1})$ terms as follows:

$$f_{k+1}(t_i t_{i+1}) \leftarrow f_k(t_i t_{i+1}) + sign(\epsilon)\,(\gamma|\epsilon|)^{w_i / \sum_{j<n} w_j}\,, \tag{4.4.12}$$

where $\sum_{j<n} w_j$ is the normalization factor, $\gamma$ is the learning rate or discount factor, $t_i$ is the $i$th tag in the query path $p$, and $f_k(\cdot)$ and $f_{k+1}(\cdot)$ are the counts before and after the update, respectively. The weights we used are

$$w_i = 2^i, \qquad i = 1, 2, \dots \tag{4.4.13}$$

If the last element $t_n$ in the query path is a data value, the weight for $f(t_{n-1} t_n)$ is defined to be the same as that for $f(t_{n-2} t_{n-1})$. The intuition for this is that an instance of a tag cannot take more than one data value in an XML data tree. Note that for query path $p = t_1 \dots t_n$, the updates to the relevant Markov histogram entries have the following property:

$$\prod_{i=1}^{n-1} (\gamma|\epsilon|)^{w_i / \sum_j w_j} = \gamma|\epsilon|. \tag{4.4.14}$$

In general, the discount factor $\gamma$ is set to be less than one and therefore it makes the error correction smaller. This prevents XPathLearner from overreacting to an estimation error and smoothens the error reduction process. The $f(t_i)$ terms are updated as

$$f_{k+1}(t_i) \leftarrow \max\{\sum_j f_{k+1}(t_j t_i),\ f_k(t_i)\}, \tag{4.4.15}$$

since the sum of the counts for all length-2 paths ending in $t_i$ must be a lower bound on the true $f(t_i)$ (by Equation (4.4.9)). The term $f_k(t_i)$ could be greater than $\sum_j f_{k+1}(t_j t_i)$, if XPathLearner has previously encountered a query feedback with a length-1 query path $t_i$.

**Example 4.9** *Consider the Markov histogram in Figure 4.6. Suppose the query path expression is $ACD$ and the feedback is $(ACD, 6)$. The estimated selectivity $\hat{\sigma}(ACD) = 3 \times 6 \div 7 \approx 3$. The observed error is $\epsilon = 6 - 3 = 3$ and using $\gamma = 1$, XPathLearner makes the following updates using the heavy-tail rule:*

$$\begin{aligned} f_{k+1}(AC) &\leftarrow \mathit{round}(3 + 3^{1/3}) = 4, \\ f_{k+1}(CD) &\leftarrow \mathit{round}(6 + 3^{2/3}) = 8, \\ f_{k+1}(C) &\leftarrow \max\{4 + 4,\ 7\} = 8, \\ f_{k+1}(D) &\leftarrow \max\{1 + 8,\ 7\} = 9, \end{aligned}$$

*The estimated selectivity of the path $ACD$ after the update is $4 \times 8 \div 8 = 4$. The* *estimation error has been reduced.*

### 4.4.8 The Delta Rule Update Method

A more principled way of updating the Markov histogram using query feedback is to attribute the estimation error to the relevant edge counts using the delta rule. The delta rule is an error reduction learning technique first proposed by Rumelhart et al. [70].

The learning scenario is the same as that in the heavy-tail method. The histogram learner is given query path $p = t_1 \ldots t_n$, with estimated selectivity $\hat{\sigma}(p)$ and query feedback $(p, \sigma(p))$, where $\sigma(p)$ is the real selectivity. We compute the observed error for query path $p$ as before,

$$\epsilon(p) = \sigma(p) - \hat{\sigma}(p). \tag{4.4.16}$$

The delta rule minimizes an error function. We choose our error function for the query path $p$ to be the squared error,

$$E(p) = [\epsilon(p)]^2 = [\sigma(p) - \hat{\sigma}(p)]^2 . \tag{4.4.17}$$

For a particular query path, we often need to express the error function as a function of a histogram entry. Let $w$ denote some entry in our Markov histogram. For example, $w$ could denote the count of the length-2 path AB in the Markov histogram of Figure 4.6. Where it is clear from context which query path is being referred to, we will use the notation $E(w)$ to denote the error as a function of $w$. The following shorthand notation is also adopted to make the equations more readable:

$$\begin{aligned} w_{\alpha\beta}^{(k)} &= f_k(\alpha,\beta), && (4.4.18)\\ W_{\beta}^{(k)} &= f_k(\beta) = \sum_{x\in\Sigma} w_{\alpha\beta}^{(k)} \\ &= w_{\alpha\beta} + \sum_{\substack{x\in\Sigma\\ x\neq\alpha}} w_{\alpha\beta}^{(k)}. && (4.4.19)\end{aligned}$$

The superscript $^{(k)}$ will be dropped if there is no confusion over the time of the variable. Note that $W_\beta$ is dependent on $w_{\alpha\beta}$.

The delta rule states that for an error function $E(w_{\alpha\beta})$ the update to term $w_{\alpha\beta}$ should be proportional to the negative gradient of $E(w_{\alpha\beta})$ with respect to $w_{\alpha\beta}$ evaluated at time $k$,

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} - \gamma \frac{\partial E(w_{\alpha\beta}^{(k)})}{\partial w_{\alpha\beta}}, \tag{4.4.20}$$

where $\gamma$ is the proportionality constant or learning rate. Simplifying the derivative using Equation (4.4.16),

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = 2\epsilon(p)\frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}} = -2\epsilon(p)\frac{\partial \hat{\sigma}(p)}{\partial w_{\alpha\beta}}$$

Using the Equation (4.4.5) for computing $\hat{\sigma}(p)$, the term $w_{\alpha\beta}$ can occur multiple times in the numerator and in the denominator. For example, in $\hat{\sigma}(ABCBAB)$, the term $w_{AB}$ would appear twice in the numerator, and twice in the denominator, because $W_B$ appears twice in the denominator. Consider the general case that $w_{\alpha\beta}$ appears $u$ times in the numerator and $v$ times in the denominator of the expression for $\hat{\sigma}(p)$,

$$\frac{\partial \hat{\sigma}(p)}{\partial w_{\alpha\beta}} = \frac{\partial}{\partial w_{\alpha\beta}} \left( \frac{w_{\alpha\beta}^u}{W_\beta^v} \times r \right), \tag{4.4.21}$$

where $r = \hat{\sigma}(p) \times W_{\beta}^{v}/w_{\alpha\beta}^{u}$ contains the rest of the terms that do not contain $w_{\alpha\beta}$. Differentiating using the quotient rule and simplifying,

$$\frac{\partial \hat{\sigma}(p)}{\partial w_{\alpha\beta}} = \hat{\sigma}(p) \left( \frac{uW_{\beta} - vw_{\alpha\beta}}{w_{\alpha\beta}W_{\beta}} \right). \tag{4.4.22}$$

Hence, our update equation for the Markov histogram term $w_{\alpha\beta} = f(\alpha, \beta)$ that occurs $u$ in the numerator and $v$ times in the denominator of the formula for $\hat{\sigma}(p)$ is

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} + 2\gamma\epsilon(p)\hat{\sigma}(p) \left( \frac{uW_{\beta}^{(k)} - vw_{\alpha\beta}^{(k)}}{w_{\alpha\beta}^{(k)}W_{\beta}^{(k)}} \right). \tag{4.4.23}$$

The learning rate parameter $\gamma$ is usually chosen by experimentation. A learning rate that is too small may result in slow convergence to the minimum error and a learning rate that is too big may result in oscillations between non-optimal error values.

**Example 4.10** *Consider the Markov histogram in Figure 4.6. Suppose the query path expression is ACD and the feedback is* $(ACD, 6)$. *The estimated selectivity* $\hat{\sigma}(ACD) = 3 \times 6 \div 7 \approx 3$. *The observed error is* $\epsilon = 6 - 3 = 3$ *and using* $\gamma = 0.5$, *XPathLearner makes the following updates using the delta rule:*

$$\begin{aligned} f_{k+1}(AC) &\leftarrow \mathit{round}(3 + 2 \times 0.5 \times 3 \times 3 \times \frac{7-3}{3 \times 7}) = 5, \\ f_{k+1}(CD) &\leftarrow \mathit{round}(6 + 2 \times 0.5 \times 3 \times \frac{3}{6}) = 8, \\ f_{k+1}(C) &\leftarrow \max\{5 + 4,\ 7\} = 9, \\ f_{k+1}(D) &\leftarrow \max\{1 + 8,\ 7\} = 9, \end{aligned}$$

*The estimated selectivity for path ACD after the updates is* $5 \times 8 \div 9 \approx 4$. *The estimation error has been reduced.*

### 4.4.9 On-line Update Overhead

Let the time needed to access an entry in the Markov histogram be $O(l)$, where $l$ is the size of the data structure used to implement the Markov histogram. Let the query path in question be $p = t_1 t_2 \ldots t_n$. The update equations for the heavy-tail rule method (Equation (4.4.12)) and the delta rule method (Equation (4.4.23)) both take $O(l)$ time. There are $O(n)$ iterations of the two loops starting on line 8 and line 15 of the update algorithm (Algorithm 4.1). Each iteration of the loop starting at line 15 requires $O(l)$ time, because the summation in line 19 is performed over at most all the length-$m$ paths in the order $(m-1)$ Markov histogram that has size $l$. Therefore each update takes

$$O(nl) \tag{4.4.24}$$

time, where $n$ is the query path length and $l$ is the size of the Markov histogram. Since $n$ is bounded by the height of the XML data tree and $l$ by the small amount memory allocated to store the Markov histogram, $l$ and $n$ are practically constants. Therefore the update overhead is effectively a constant.

### 4.4.10 Batch Update Strategies

While the emphasis of this dissertation is on-line methods, we briefly outline two batch processing strategies that can be used with XPathLearner. Batch updates assume that a buffer is available to hold $b$ query feedback tuples until the batch update procedure is activated. Batch updates are useful for two reasons.

1. When it is not feasible or possible to update the Markov histograms after each query, a batch of query feedback can be collected and used to update the Markov histogram periodically.

2. When a batch of query feedback is available during initialization, the Markov histogram can be initialize using a batch update strategy instead of being initialized to empty histograms. This may reduce the estimation errors associated with a ‘cold start’.

### Subpath Elimination

The subpath elimination strategy for batch updates is based on three assumptions:

1. the set of query paths in the batch contains some paths that are subpaths of other paths in the batch,
2. the underlying data remained the same for the entire batch of query paths, and
3. the Markov assumption holds reasonably well for the path selectivities

The implication of these assumptions is that the feedback for the subpath can give additional statistical information about the longer path. For example, if the query paths `ABCD` and `ABC` both occur in the batch, then using their true selectivities from query feedback, we know that $f(CD)/f(C)$ should be equal to $\sigma(ABCD)/\sigma(ABC)$. We can now use the on-line delta rule method to update $f(CD)$ and $f(C)$ using $\sigma(ABCD)/\sigma(ABC)$ as the true value for $f(CD)/f(C)$. The subpath elimination strategy assumes the delta rule method for updating the Markov histogram.

We sketch the algorithm for the subpath elimination batch update method in Algorithm 4.2. The batch of query feedback is first sorted (line 1) according to increasing query path length and according to lexicographic order for paths with the same length. Each iteration of the `while`-loop (line 2) then applies the subpath elimination batch update method until some stopping criterion is true. Exactly one

**Algorithm 4.2** SUBPATHELIMINATION($f$, *Qfb*)

```
INPUT: Markov histogram $f$, Batch of Query Feedback $Qfb$
1: Sort $Qfb$ in increasing path length and lexicographic order.
2: while stopping criterion $\neq$ true do
3:   for all query path $p \in Qfb$ do
4:     Search for longest query path $z \in Qfb$ that is a subpath of $p$
5:     if $z$ does not exist then
6:       UPDATE$(f, (p, \sigma(p)), \hat{\sigma}(p))$ using delta rule.
7:     else
8:       $\sigma \leftarrow \sigma(p)/\sigma(z)$
9:       $\hat{\sigma} \leftarrow \hat{\sigma}(p)/\hat{\sigma}(z)$
10:      for all $w_{\alpha\beta}$ in $\hat{\sigma}(p)$, but not in $\hat{\sigma}(z)$ do
11:        Update $w_{\alpha\beta}$ using $\sigma, \hat{\sigma}$, and delta rule Equation (4.4.23).
12:      end for
13:      Update all affected $W_\beta$ using Equation (4.4.15).
14:    end if
15:  end for
16: end while
```

scan through the batch of query feedback is made in each iteration. The sorted batch of query feedback facilitates the search for the longest query path that is a subpath of the current query path $p$ that is being processed (line 4). If no subpath is found, the on-line delta rule update procedure is used to perform the update (line 6). If a subpath is found, the true selectivity of the subpath is eliminated from the true selectivity of the current query path $p$. The delta rule is then applied to the histogram entries that are used in the computation of $\hat{\sigma}(p)$ and not used in the subpath (line 11).

The `while`-loop applies the subpath elimination batch update method until some stopping criterion is true. The `while`-loop can terminate when

- the error has reached a given error threshold, or
- the decrease in error between consecutive iterations has fallen below a given threshold, or

- the number of iterations has exceeded a given constant.

### Batch Delta Rule

Let $B$ be the set of $b$ query paths whose query feedback has been buffered. We can use delta rule to update the Markov histogram by defining a sum of squared error function for the set of query paths $B$,

$$E(B) = \sum_{p \in B} [\epsilon(p)]^2 = \sum_{p \in B} [\sigma(p) - \hat{\sigma}(p)]^2 .$$

To update a particular histogram entry $w_{\alpha\beta}$, we compute the derivative,

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = \sum_{p \in B} \frac{\partial [\epsilon(p)]^2}{\partial w_{\alpha\beta}} = \sum_{p \in B} 2\epsilon(p) \frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}}$$

Let $B'$ be the subset of $B$ containing paths that require the histogram entry $w_{\alpha\beta}$ in the computation of their selectivity estimates. The selectivity estimates of all the paths in $B - B'$ do not depend on $w_{\alpha\beta}$ and their derivative is zero,

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = \sum_{p \in B'} 2\epsilon(p) \frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}} = - \sum_{p \in B'} 2\epsilon(p) \frac{\partial \hat{\sigma}(p)}{\partial w_{\alpha\beta}}. \tag{4.4.25}$$

Using Equation (4.4.21) and the delta rule, we can update a Markov histogram entry $w_{\alpha\beta} = f(\alpha, \beta)$ using the query feedback for a set of query paths $B'$ using,

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} + \frac{2\gamma}{w_{\alpha\beta}^{(k)} W_{\beta}^{(k)}} \sum_{p \in B'} \left\{ \epsilon(p) \hat{\sigma}(p) \left[ u(p) W_{\beta}^{(k)} - v(p) w_{\alpha\beta}^{(k)} \right] \right\}, \tag{4.4.26}$$

where $\gamma$ is the learning rate parameter, and $u(p)$ and $v(p)$ are the number of times that $w_{\alpha\beta}$ occurs in the numerator and denominator of the formula for $\hat{\sigma}(p)$ respectively.

We sketch the algorithm for the delta rule batch update method in Algorithm 4.3. The first `for`-loop (line 1) initializes all non-existent entries in the histogram that

**Algorithm 4.3** BatchDeltaRuleUpdate($f$, *Qfb*)

```
INPUT: Markov histogram f, Batch of Query Feedback Qfb
 1: for all $w_{\alpha\beta} = f(\alpha\beta)$ used in Qfb do
 2:   Initialize $w_{\alpha\beta} = 1$, if $w_{\alpha\beta}$ is not in histogram.
 3:   Update $W_\beta$ using Equation (4.4.15).
 4: end for
 5: while stopping criterion $\neq$ true do
 6:   Initialize array $U$ to zeros.
 7:   for all query path $p \in Qfb$ do
 8:     for all $w_{\alpha\beta}$ involved in $\hat{\sigma}(p)$ do
 9:       $U[w_{\alpha\beta}] \leftarrow U[w_{\alpha\beta}] + \epsilon(p)\hat{\sigma}(p)\left\{u(p)W_\beta^{(k)} - v(p)w_{\alpha\beta}^{(k)}\right\}$
10:     end for
11:   end for
12:   for all $w_{\alpha\beta} = f(\alpha\beta)$ used in Qfb do
13:     $w_{\alpha\beta} \leftarrow w_{\alpha\beta} + \frac{2\gamma}{w_{\alpha\beta}^{(k)}W_\beta^{(k)}}U[w_{\alpha\beta}]$
14:   end for
15:   Update all affected $W_\beta$ using Equation (4.4.26).
16: end while
```

are required for the batch of queries. The `while`-loop (line 5) implements the delta rule batch update method as derived in Equation (4.4.26). Our implementation is optimized using the assumption that the size of the batch of query feedback is much larger than the total number of histogram entries updated. Hence each iteration of the `while`-loop scans through the batch of query feedback only once. If this assumption is not true, the order of the two `for`-loops (line 7 and 8) needs to be reversed.

The `while`-loop applies the delta rule until some stopping criterion is true. Reasonable stopping criteria includes stopping when the error has reached a given error threshold, or when the decrease in error between consecutive iterations has fallen below a given threshold, or when the number of iterations has exceeded a given constant.

## 4.5 Experiments

We implemented our XPathLearner in C/C++ using the XML Parser Toolkit [21]. We investigate the following issues in our experiments:

1. the accuracy of XPathLearner under varying memory constraints when it is trained on one query workload and evaluated using a different workload,
2. the convergence properties associated with XPathLearner,
3. the adaptivity of XPathLearner when the workload changes from one distribution to another,
4. the on-line accuracy of XPathLearner, i.e., the estimation accuracy on one workload, as XPathLearner updates itself after each query,
5. the distribution of on-line errors according to the true selectivity of the queries, and
6. the occurrence frequency of on-line errors.

We describe briefly the data sets used, the query workloads, the performance measures, and the methods used in the comparisons, before describing each experiment in greater detail.

**Performance Measures.** We have used the average relative error and the average absolute error to measure the accuracy of XPathLearner. The average relative error (*a.r.e.*) and the average absolute error (*a.a.e.*) with respect to a set of queries

| Characteristic | DBLP | XMark |
|---|---|---|
| Size (MBytes) | 10 | 116 |
| No. of nodes | 261,256 | 1,479,327 |
| No. of distinct tags | 29 | 74 |
| No. of distinct values | 91,878 | 415,262 |
| Path tree depth | 5 | 13 |
| No. of path tree tag nodes | 57 | 514 |
| No. of path tree value nodes | 109,741 | 675,844 |

**Table 4.1**: Characteristics of the DBLP data set and the XMark data set.

$Q$ of size $n$ are defined as

$$a.r.e. = \frac{1}{n}\sum_{q\in Q}\frac{|\sigma(q)-\hat{\sigma}(q)|}{\sigma(q)}, \quad a.a.e. = \frac{1}{n}\sum_{q\in Q}|\sigma(q)-\hat{\sigma}(q)|, \tag{4.5.27}$$

where $\sigma(q)$ is the selectivity of query path $q$ in the workload $Q$ and $\hat{\sigma}(q)$ is the corresponding estimated selectivity. The state of the selectivity estimation method is assumed to remain unchanged for all the queries in the workload $Q$. In an on-line setting, we would also like to measure the on-line or dynamic performance of an estimation method. The *on-line a.a.e.* and the *on-line a.r.e.* are defined as in (4.5.27), except that the selectivity estimation method is allowed to update itself in between queries.

**Data Set.** We performed our experiments on the XMark synthetic data set [72,85] and on several real data sets: DBLP [53], Swiss protein[3], and Shakespeare[4]. For the real data sets, we present only the representative results from the DBLP data set. The characteristics of each data set are summarized in Table 4.1.

[3]http://www.expasy.ch/sprot

[4]http://metalab.unc.edu/bosak/xml/eg/shaks200.zip

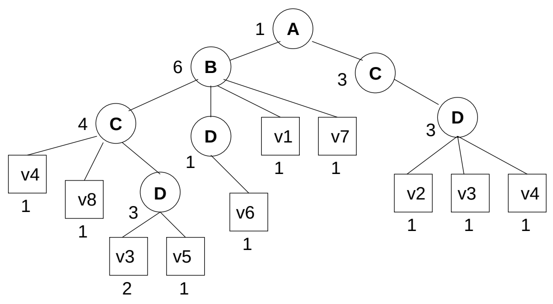


**Figure 4.7**: The path tree corresponding to the XML data tree in Figure 4.4. Circle nodes denote tag names and square nodes denote values.

**Query Workload.** In the experiments we present in this section we used workloads consisting of simple and single-value query path expressions with positive selectivity. We did generate negative workloads (consisting of query paths that do not appear in the data, i.e., query paths with zero selectivity) by generating random sequences of legal tags ending with a random legal value; however, for all the negative workloads that we generate, our XPathLearner consistently returns a selectivity of 1 for each negative path[5]. (The default return value for paths that are not captured in our Markov histogram is 1.) Hence, the average absolute error is 1. This result contrasts sharply with the summarized Markov tables of [1], where the average absolute error for negative workloads can be as high as 250.

Positive query workloads are generated from the path tree [1] of the given XML data set. Recall that a path tree summarizes an XML data tree by aggregating every sibling having the same tag into a single node annotated by a count of the number of occurrences in the original XML data tree. Figure 4.7 shows an example of a path tree corresponding the XML data tree in Figure 4.4. We generate positive path queries as follows. All the root-to-leaf paths in the path tree are first enumerated. A query path is generated by randomly choosing a root-to-leaf path and then randomly

[5]A positive workload of 1000 query paths was used as the training workload for that experiment.

choosing a starting level and a path length that are within limits of the length of the chosen root-to-leaf path. The random query path of the chosen length is then output starting from the chosen level in the chosen root-to-leaf path.

These root-to-leaf paths are not chosen uniformly, but from a distribution weighted according to their selectivities,

$$P[\text{choosing root-to-leaf path } p] = \frac{\sigma(p)}{\sum_{r \in R} \sigma(r)}, \tag{4.5.28}$$

where $R$ is the set of all root-to-leaf path of the given path tree. The reason for choosing the root-to-leaf path in this way is to prevent the query workload from having too many query paths with very small selectivities.

**Comparisons.** We compare the performance of the off-line method and several versions of XPathLearner. The labeling convention for the different versions are as follows:

**xpl-o1-dt-lb** denotes a first order ("o1") XPathLearner using the delta rule ("dt") and the compressed histogram approach for storing paths with leaf values ("lb" for leaf buckets);

**xpl-o1-ht-lb** denotes a first order XPathLearner using the heavy-tail rule ("ht") and the compressed histogram approach;

**xpl-o1-dt-ca** denotes a first order XPathLearner using the delta rule and the cache-based approach to storing paths with leaf values ("ca");

**xpl-o2-dt-ca** denotes a second order ("o2") XPathLearner using the delta rule and the cache-based approach.

The **off-line** method differs from the XPathLearner in that the Markov histogram is constructed by scanning the repository. It differs from the Markov table method [1] in that (1) no summarization is done of the tag-tag counts, (2) the tag-value counts are stored, and (3) the tag-value counts are summarized using the method described in Section 4.4.3.

**Initial Condition.** We assume that we do not know anything about the workload distribution at the start of each experiment; that is, we start with an empty Markov histogram. More sophisticated ways of obtaining an initial Markov histogram are possible, but an empty initial histogram represents a reasonable worst case.

**Counting Memory.** An order $(m-1)$ XPathLearner using the compressed histogram approach consists of $m$ tables and a compressed histogram data structure. Each table $i$ stores entries of the form $\langle path_i, count\rangle$, where $path_i$ is a simple path of length $i$. A compressed histogram stores $k$ entries of the form $\langle path_m, count\rangle$ where $path_m$ is a single-value path of length $m$, and some number of aggregated entries of the form $\langle path_{m-1}, feature, sum, num\rangle$, where $path_{m-1}$ is a simple path of length $m-1$. Each length $i$ path requires $i$ integers. All other fields take one integer each. Each integer takes four bytes. (For a first order example see Figure 4.6.)

An order $(m-1)$ XPathLearner using the cache-based approach consists of just $m$ tables, each table $i$ storing entries of the form $\langle path_i, count, counter\rangle$, where $path_i$ is a simple or single-value path of length $i$. Each length $i$ path again requires $i$ four-byte integers, each count requires one four-byte integer, and each counter requires one byte.

The memory requirements of each method is accounted for by counting the number of entries in each table and compressed histogram, and multiplying by the corresponding memory requirement of each type of entry.

**Parameters.** We set the learning rate $\gamma$ to 1 for the heavy-tail rule update strategy and 0.1 for the delta rule update strategy. For the cache-based approach, we set the threshold for evicting low count entries to 30, i.e., entries with count less than 30 are candidates for eviction. These values were found to be reasonably good by experimentation.

### 4.5.1 Accuracy vs Space

In this experiment, we measure the estimation error under varying memory constraints. Two different query workloads from the DBLP data set are used: one as the training set and the other as the testing set. Each workload consists of 4096 query paths, of which about 3100 paths are distinct. The average true selectivities of the training and testing workloads are 2034 and 2296[6], respectively.

The goal of this experiment is to see how our on-line Markov histogram performs on a workload that is different from its training workload. We define a workload difference measure with respect to a first-order Markov histogram in order to quantify the difference between two workloads.

**Workload Diff.** Given two workloads A and B, we construct for each workload the set of length-2 paths of all the query paths in the workload. Let the set of length-2 paths of A and B be $S_A$ and $S_B$, respectively. The workload difference

[6]Since the total number of nodes in the XML data tree is $N = 261,256$, these selectivities correspond to 0.77 % and 0.87 %.

measure of A and B is

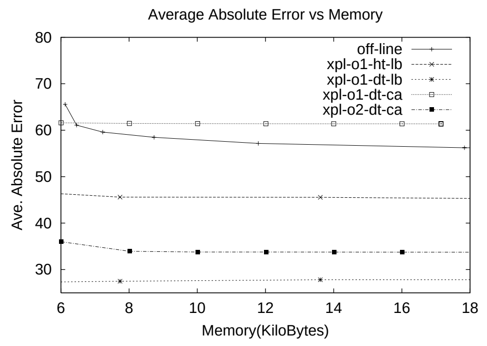
$$\text{workload\_diff(A,B)} = 1 - \frac{|S_A \cap S_B|}{|S_A \cup S_B|}. \tag{4.5.29}$$

Intuitively, the workload_diff measures how different the first-order Markov models of the two given workloads are.

We experimented on a large number of training-testing workload pairs and we present a typical result set in Figure 4.8. The workload_diff of the training and testing workload we present is 88.4%. For the XPathLearner using the compressed histogram approach, we measure the estimation error as $k$ varies from 32 to 4096. The $k$ values (for the top $k$ values) are then converted to memory usage in bytes and the estimation errors are plotted against memory usage. Our experiments show that in terms of absolute errors our on-line XPathLearner (**xpl-o1-ht-lb**, **xpl-o1-dt-lb**, and **xpl-o2-dt-ca**) is more accurate than the off-line version. Amongst the two on-line update strategies, the delta rule is usually more accurate than the heavy-tail strategy. In terms of relative errors, the second order **xpl-o2-dt-ca** is the most accurate under tight memory constraints. The performance of **xpl-o1-ht-lb** and **xpl-o1-dt-lb** are within 10% of the **off-line** method.

The relationship between $k$ and the memory usage in bytes for the off-line and on-line XPathLearner (using compressed histogram approach) is graphed in Figure 4.12. We note that, for fixed $k$, the memory requirement of the off-line method is more than that of the on-line method; for $k = 512$, the off-line method requires 2947 bytes and the on-line method only 1934 bytes. The off-line method has to store statistics for the entire XML repository while the on-line method only needs to store the statistics of the workload.

We also show our results for two workloads (training and testing) consisting of



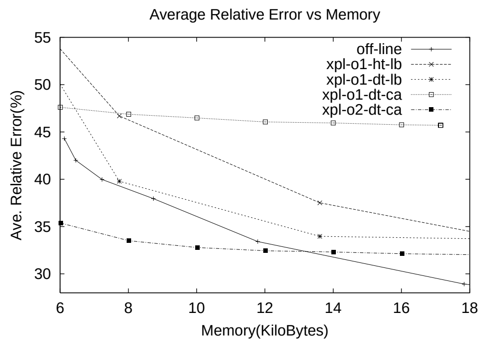


**Figure 4.8**: Accuracy vs Memory for DBLP data set. Accuracy of the on-line Markov histogram method on a testing workload that is 88.4% different from the training workload. Both workloads contain 4096 single-value query paths, about 3100 of which are distinct.

| Method | a.a.e. | a.r.e.(%) | Memory |
|---|---|---|---|
| **xpl-o1-dt-lb** | 0.086 | 0.197 | 764 Bytes |
| **off-line** | 0.110 | 0.331 | 796 Bytes |
| **xpl-o1-ht-lb** | 1.198 | 0.243 | 764 Bytes |
| **xpl-o1-dt-ca** | 87.9 | 19.535 | 760 Bytes |

**Table 4.2**: Accuracy of various methods for simple path expression queries on the DBLP data set. Estimation error of the **off-line** method, the on-line **xpl-o1-ht-lb**, **xpl-o1-dt-lb**, and **xpl-o1-dt-ca** methods for a workload consisting only of simple path expressions (tag-only path expressions). The on-line **xpl-o1-dt-lb** outperforms the others.

simple path expressions only (no value nodes involved). Both workloads consist of 1000 simple path expressions, and although the two workloads are different, their workload_diff is zero[7]. This property arises because the set of length-2 paths entailed by both workloads are the same. The estimation error rates are tabulated in Table 4.2.

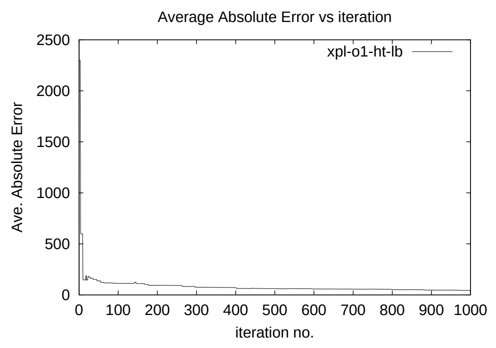
[7]Since the number of possible tags is small, a workload of 1000 paths captures most of the length-2 paths.

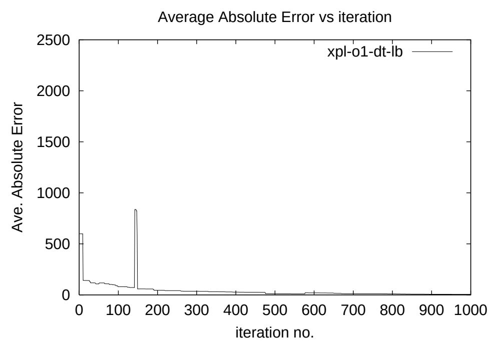
### 4.5.2 Convergence

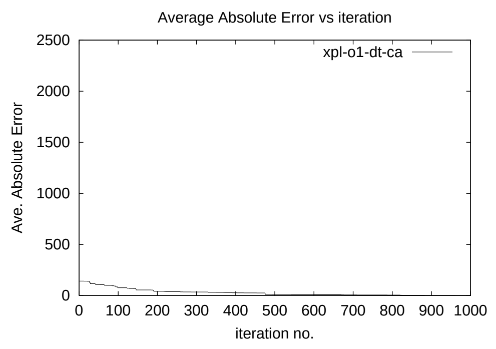
We want to investigate how well the on-line method converges to a given workload distribution. One query workload of 1000 query paths (840 distinct) from the DBLP data set is used in this experiment. We measure the average absolute and relative errors over the entire workload as the histogram learner processes each query path in the same workload. Since the Markov histogram is initially empty, the first few error measurements will be large, and as the Markov histogram converges to the workload distribution, the measured error will be small. The error measurements over each iteration or update of a Markov histogram are plotted in Figure 4.9 and Figure 4.10. All the methods are given the same amount of memory (7.7 KBytes or $k = 512$). The results in Figure 4.9 and Figure 4.10 show that the accuracy of our XPathLearner reaches very acceptable levels within the first 100 iterations. Figure 4.11 shows how the memory constraint (governed by $k$ when the compressed histogram approach is used) affects the convergence properties of XPathLearner. Our results show that XPathLearner can still be very accurate even when little memory is allocated. The spike in the **xpl-o1-dt-lb** plot of Figure 4.9 is due to the occurrence of a path that violates the Markov assumption.

### 4.5.3 Adapting to Data Distribution Change

This experiment investigates how the on-line Markov histogram will adapt to a workload that has its first 1000 query paths generated from the original DBLP path tree and the next 1000 query paths generated from a modified DBLP path tree with random perturbation to the counts at each node. The perturbation is intended to simulate the DBLP data changing over time. We introduce the perturbation by generating a random number $U \in [1 - \delta, 1 + \delta]$ for each node $r$ in the path tree.







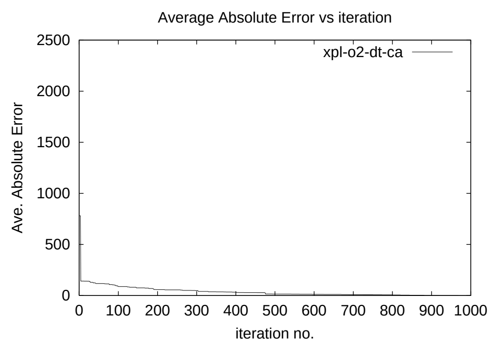


**Figure 4.9**: Convergence of *a.a.e.* (fixed memory) on the DBLP data set. The absolute error averaged over an entire workload at each iteration of the learning process.

The count associated with node $r$ is then scaled by the random number $U$,

$$count(r) \leftarrow \max\{1,\ U \times count(r)\} \tag{4.5.30}$$

The modified path tree that we generated using $\delta = 0.7$ has a Kullback-Liebler divergence of 0.129299 bits. The Kullback-Liebler (KL) divergence of a modified path tree $f_1$ with respect to the original path tree $f_0$ is defined as

$$KL(f_0|f_1) = \sum_{y \in \{\text{root-to-node paths}\}} f_0(y) \log \frac{f_0(y)}{f_1(y)}. \tag{4.5.31}$$

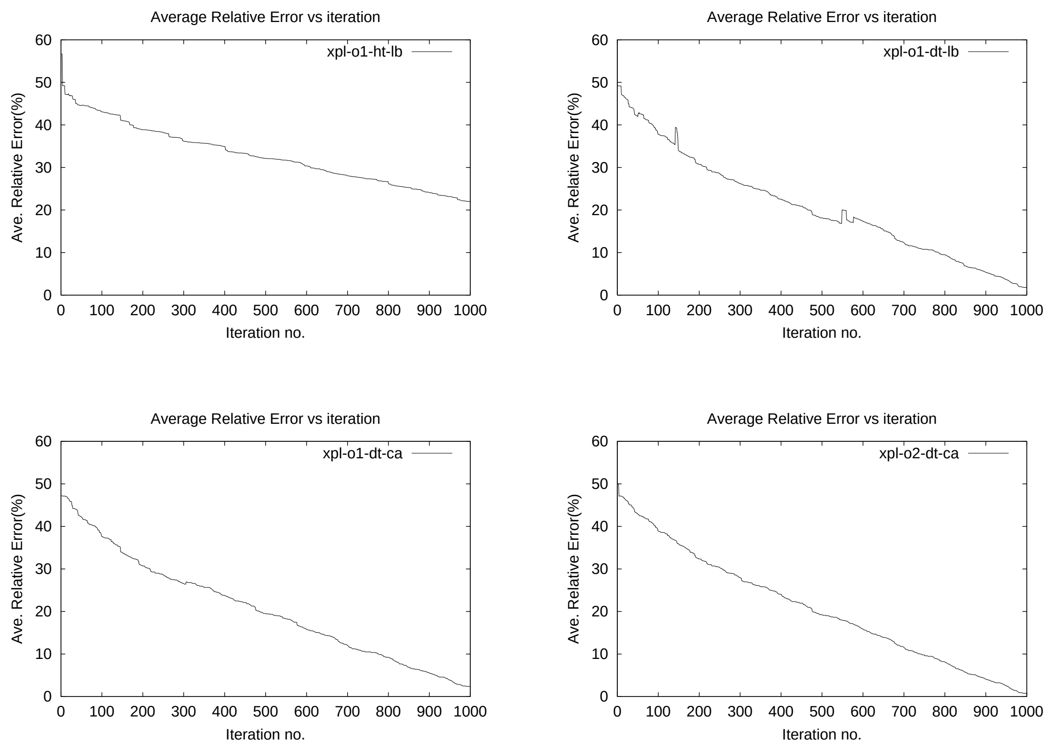


**Figure 4.10**: Convergence of *a.r.e.* (fixed memory) for the DBLP data set. The relative error averaged over an entire workload at each iteration of the learning process.

The KL divergence is a common difference measure of distributions [47].

This experiment is performed as follows. Let the workload of 1000 query paths generated from the original path tree be $Q_{old}$ and the workload of 1000 query paths generated from the modified path tree be $Q_{new}$. Let $Q_{mix}$ be the concatenation of $Q_{old}$ and $Q_{new}$. We let our XPathLearner ($k = 32$) learn the Markov histogram from this mixed workload. For the first 1000 iterations we measure the average absolute error over $Q_{old}$ after each update and for the next 1000 iterations we measure the average absolute error over $Q_{new}$ after each update. The average absolute error at the end of each iteration is plotted in Figure 4.13. The spikes near iteration 320 in the plot for **xpl-o1-dt-ca** are probably an artifact of the cache-based approach.

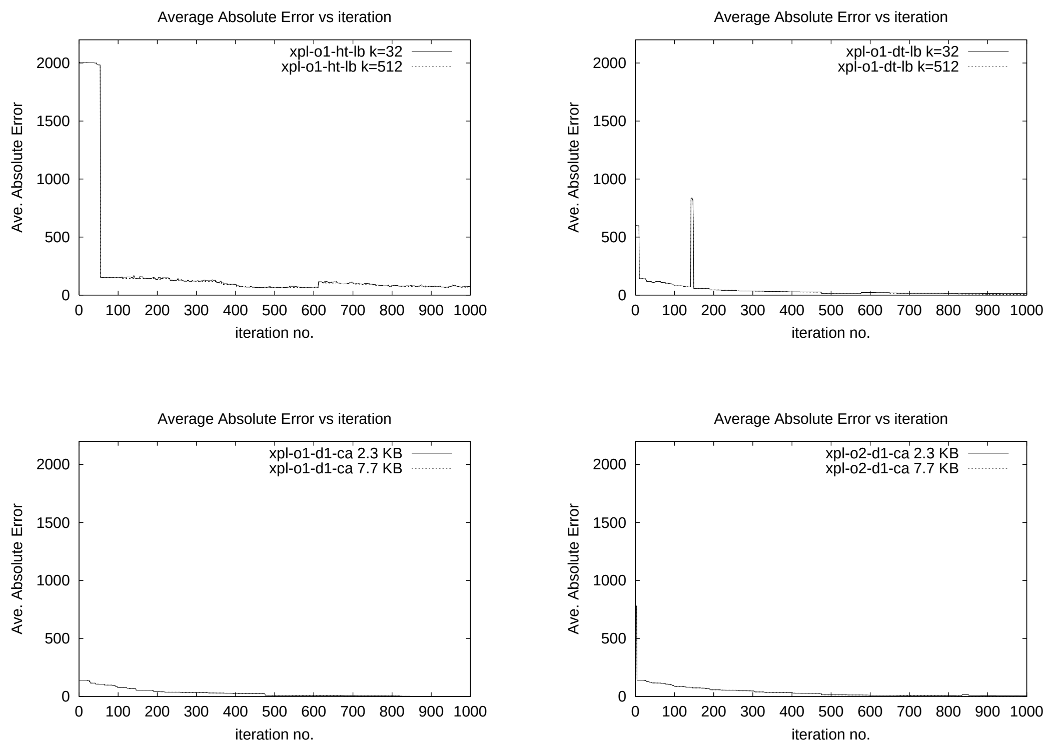


**Figure 4.11**: Convergence of *a.a.e.* (variable memory) for the DBLP data set. Each plot shows the absolute error convergence curves for two different memory constraint and for most of the time, the two curves are indistinguishable. These plots show that convergence is not sensitive to the memory constraint.

The spike at iteration 1001 is expected and shows the transition from workload $Q_{old}$ to workload $Q_{new}$. Since the distribution underlying $Q_{old}$ is different from that underlying $Q_{new}$, the average absolute estimation error with respect to $Q_{new}$ at iteration 1001 is very large. About 100 iterations after the transition, the on-line method has adapted to $Q_{new}$.

## 4.5.4 On-line Accuracy vs Space

How does XPathLearner perform in an on-line setting? We measure the on-line estimation errors of XPathLearner under different memory constraints and plot

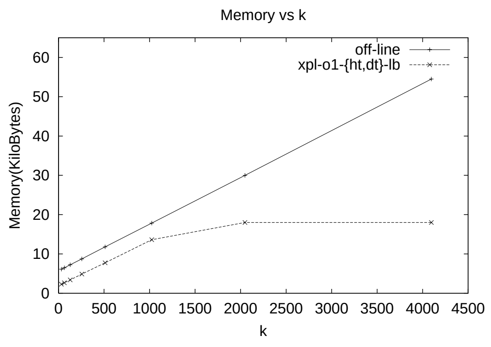


**Figure 4.12**: Memory vs $k$ for the DBLP data set.

the results in Figure 4.14. A query workload of 10,000 queries generated from the XMark data set is used. Recall that the on-line average error measures the error of each query in the workload while allowing the estimation method to update itself after each query. The on-line average error therefore measures the performance of XPathLearner while it is "in action". In terms of *on-line a.r.e.*, the XPathLearner using the delta rule and compressed histogram is the most accurate method among the four methods. In terms of *on-line a.a.e.*, the cache-based approach (with delta rule) seems to be more accurate.

### 4.5.5 On-line Errors vs Selectivity

We further analyze the on-line estimation errors (absolute errors) by partitioning the errors into 40 bins according to the true selectivity of the queries. The errors in each bin are averaged and the average error for each bin is plotted in Figure 4.15. The on-line errors we used are for the same workload of 10,000 queries from the XMark data set used in Section 4.5.4. Each XPathLearner method is given about 7.3 KBytes of memory. The plots show that the compressed histogram approach tend to produce larger errors for larger selectivities compared to the cache-based approach. This result is consistent with the on-line accuracy results in Section 4.5.4

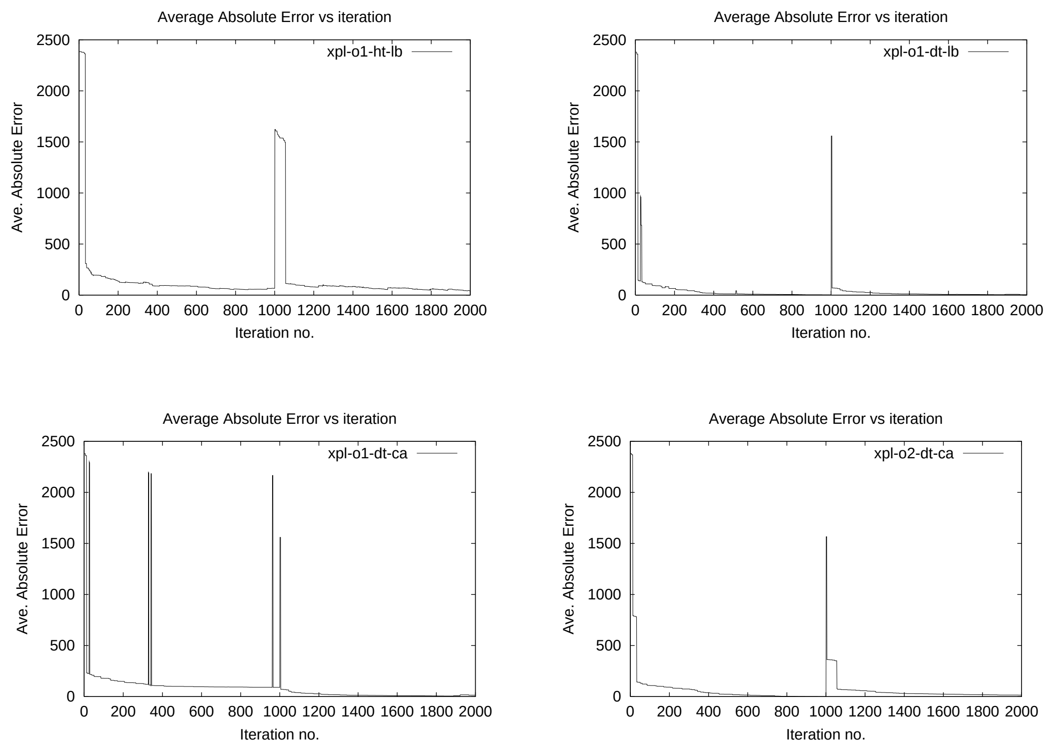


**Figure 4.13**: Adaptability (DBLP data set). Average absolute error averaged over $Q_{old}$ for iteration 1-1000 and averaged over $Q_{new}$ for iteration 1001-2000.

where the compressed histogram approach is more accurate than the cache-based approach in terms of relative error, even though the cache-based approach is better in terms of absolute error.

### 4.5.6 The Frequency of Online Errors

We investigate how often on-line errors of a certain (relative) magnitude occurs. The on-line relative errors we used are for the same workload of 10,000 queries from the XMark data set used in Section 4.5.4. Each XPathLearner method is given about 7.3 KBytes of memory. The on-line relative errors are partitioned into 40 bins by their magnitude and the normalized count of the number of errors in each

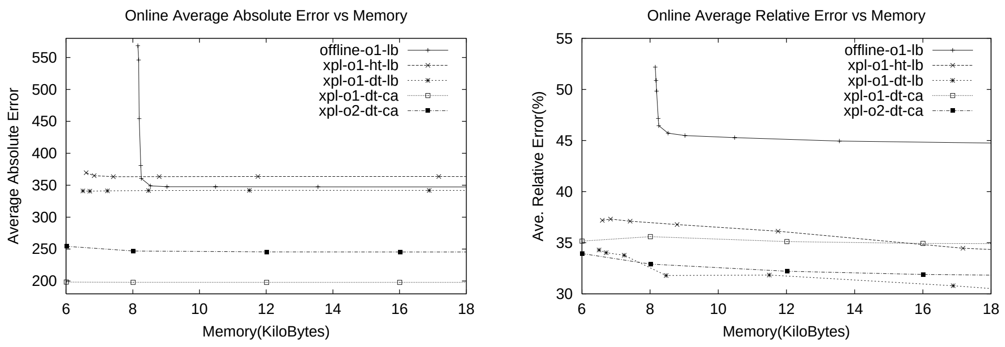


**Figure 4.14**: On-line error vs memory for the XMark data set. The on-line estimation performance of XPathLearner under varying memory constraints on a workload of 10,000 queries drawn from the XMark data set.

bin is plotted in Figure 4.16. Our results show that the relative estimation error is very small most of the time. Relative errors larger than 200% occur less than 2 % of the time.

### 4.5.7 Discussion

Our experiments have shown that XPathLearner is very accurate in terms of the traditional error measures as well as the on-line error measures we proposed. Moreover, we have shown that most of the the errors that XPathLearner makes are very small relative errors. XPathLearner adapts readily to changing data distributions and converges to very low error rates even under tight memory constraints.

What may be surprising is that XPathLearner can be even more accurate than the costly off-line method. We would expect the off-line method to be more accurate than the on-line method because the Markov histograms are constructed by scanning the data itself. The main reason for the on-line method being more accurate than the off-line is that the off-line method attempts to model all the data using the limited amount of given memory, whereas the on-line method uses the same amount of

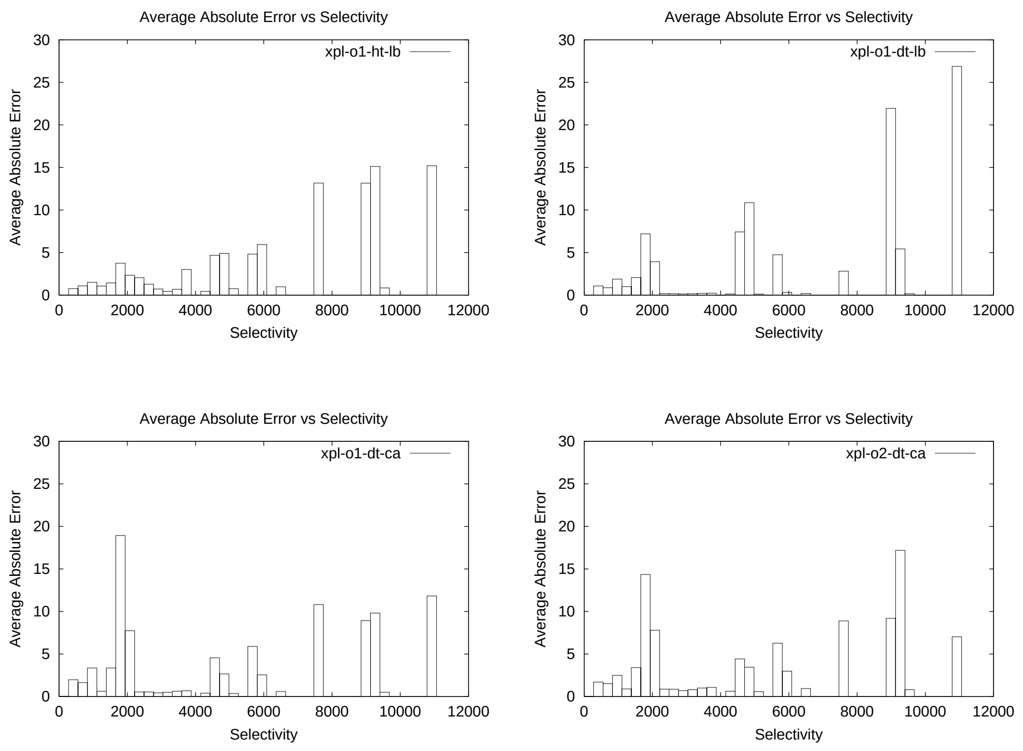


**Figure 4.15**: On-line error distribution across selectivities for the XMark data set.

memory to model the portion of the data that is frequently queried in the workload. A secondary reason is that in constructing a Markov histogram over all the data, the off-line method can be affected by paths that violate the Markov assumption. Recall that for an order-1 Markov chain, the frequency of the next tag depends on the frequency of the current tag only. We illustrate effect these Markov-violating paths with an example.

**Example 4.11** ***(Paths violating the order-1 Markov assumption.)*** *Consider the path tree in Figure 4.17(a) and the two paths* `//A/C/D` *and* `//B/C/D` *in the path tree. Assume that the tags* `A, B, C, D` *do not occur anywhere else in the path tree. The two paths* `//A/C/D` *and* `//B/C/D` *violate the order-1 Markov assumption because the count of* `D` *conditioned on being at* `C` *(via* `B`*) is* 1/200 *and the the*

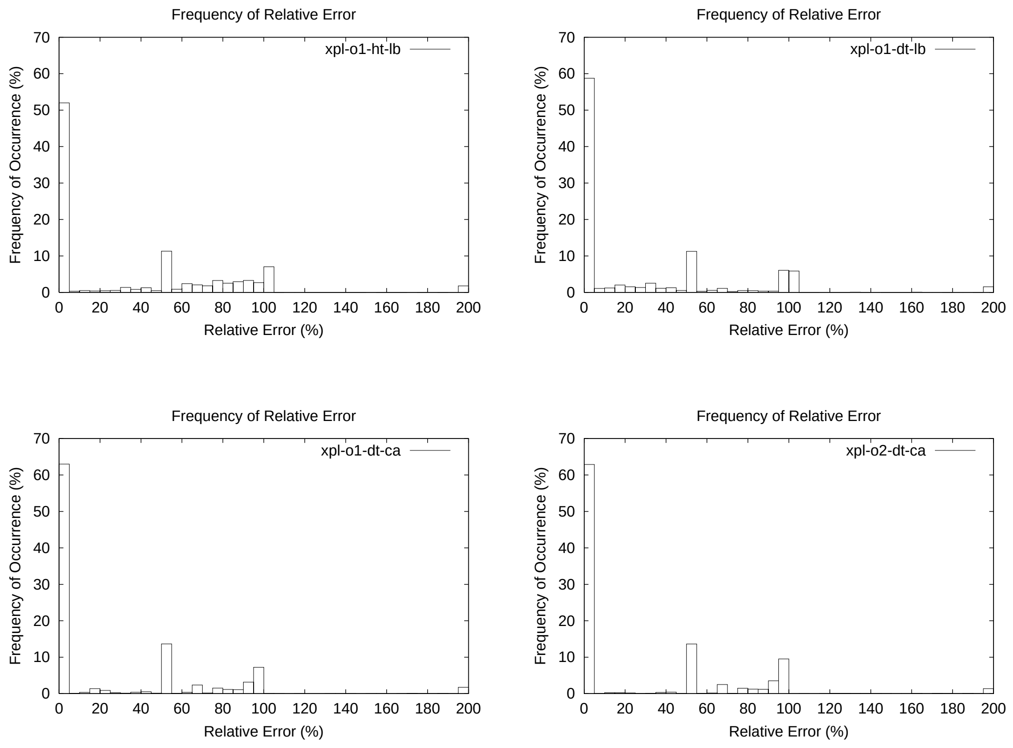
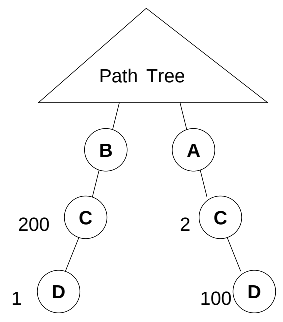
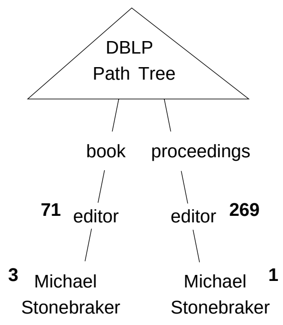


**Figure 4.16**: Frequency of on-line relative errors for the XMark data set. The occurrence frequency of different ranges of on-line relative errors for an XMark query workload of 10,000 queries are plotted. More than half the errors that XPathLearner makes are relative errors of less than 5%.

*count of `D` conditioned on being at `C` (via `A`) is* 50*. The violation is due to the large difference between the two conditional counts. In the off-line method, these two conditional counts will be aggregated and the entries* $f(AC) = 2$*,* $f(BC) = 200$*, and* $f(CD) = 101$ *will be stored. Hence, the selectivity of path `//B/C/D` will be computed as* $\hat{\sigma}(BCD) = 200 \times 101 \div 202 \approx 100$ *which has an absolute error of* 99*. For the on-line method, in the best case when the workload does not contain any path with `//A/C` or `//C/D`, the on-line method with delta rule will learn that* $f(BC) = 200$ *and* $f(CD) = 1$*. XPathLearner with delta rule will then give a more accurate estimate of `//B/C/D`.*



(a) An artificial example.



(b) An example from the DBLP data.

**Figure 4.17**: Two examples of a pair of paths that violate the order-1 Markov assumption.

The effect of these Markov-violating paths on the on-line method depends on whether the Markov-violating paths occur in the workload and how they are interleaved in the workload. The bottom line is that the on-line method has a chance of avoiding these Markov-violating paths, while the off-line method does not.

## 4.6 Conclusions

In this chapter, we presented XPathLearner, a new on-line method for estimating the selectivities of path expressions (simple, single-value, multi-value) without examining the XML data. All previous methods require an off-line scan of the XML repository to collect the statistics. Our method relies on the feedback from the query execution engine to construct and refine a Markov histogram of the underlying path selectivity statistics. Besides the on-line property, our method also has two other novel features: (a) XPathLearner is workload aware in collecting the statistics and thus can be more accurate than the more costly off-line method under tight memory constraints, and (b) XPathLearner automatically adjusts the statistics using query feedback when the underlying XML data change.

We have proposed two approaches to deal with storing the statistics of the large number of single-value paths (tag-value pairs). These single-value path statistics are used to estimate the selectivity of paths containing data values. The cache-based approach treats the Markov histogram like a cache and evicts entries when the given memory is exceeded. The compressed histogram method stores the largest-$k$ single-value paths (tag-value pairs) exactly and aggregates the rest into buckets.

We have also presented two on-line update strategies, the heavy-tail rule and the delta rule, as well as outlined two batch update strategies, subpath elimination and batch delta rule.

Our experiments show that our method is accurate under modest memory requirements on both synthetic data and real data and that XPathLearner adapts readily to changing data distributions and converges to very low error rates even under tight memory constraints.

# Chapter 5

# On-line Classification-Based Histograms

## 5.1 Introduction

One of the key insights gleaned from Chapter 4 is that given limited space, for the purpose of selectivity estimation, modeling queries can be more effective than modeling the data. In this chapter, we build on this insight and present CXHist, a new type of on-line histogram for storing the approximate mapping between queries and their selectivities.

Previous types of histograms are often restricted to a single data and query type (see [39] for a survey). For example, a histogram technique for numerical attributes cannot be used for string attributes, and a histogram technique for single attribute queries cannot accurately estimate multi-attribute queries. Hence a database system often has to maintain multiple histograms of different types according to attribute type and query type. CXHist addresses this issue by providing a single histogram technique that can be used for different types of data and queries. The key paradigm shift that allows CXHist this flexibility is that CXHist models queries instead of data. Therefore CXHist is different from all previous types of histograms that model the data. The decoupling of CXHist from the underlying data model also enables CXHist

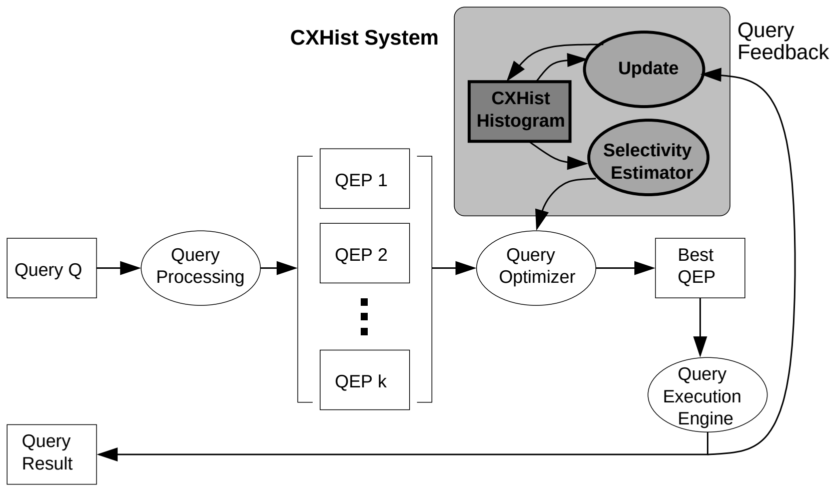


**Figure 5.1**: Schematic of CXHist.

to be applicable to both relational and XML databases.

CXHist also differs from previous histogram types in that it uses a Bayesian classifier to remember in an approximate manner the mapping between queries and their selectivities. CXHist builds a histogram in an on-line manner by grouping queries into buckets using their true selectivities obtained from query feedback (see Figure 5.1). The correspondence between a bucket and its associated set of queries is captured approximately using a Bayesian classifier. The Bayesian classifier is learned from query feedback in an on-line manner. An introduction to Bayesian classifiers is given in Section 5.2.2 for the benefit of readers not familiar with the subject.

In this chapter, we will describe CXHist (Section 5.2) and apply CXHist to the specific problem of estimating the selectivity two important types of XML queries (Section 5.3). The two types of XML queries are of the form $\langle path, string \rangle$ and differ in the string constraints specified by *string*. Exact match queries retrieve all XML nodes that match the query path and the query string value exactly. Substring match queries are similar, but relax the matching of the query string to a substring match. In Section 5.6 we will show experimentally that CXHist provides accurate selectivity estimation for both exact match queries and substring match queries. Section 5.7 concludes this chapter.

## 5.2 CXHist Histograms

### 5.2.1 Overview

A key difference between our on-line CXHist technique and traditional off-line histograms is that the off-line methods are modeling data whereas CXHist is modeling queries. A query $q$ is modeled as a set of features $\vec{x} = \langle x_1, \ldots, x_k \rangle$. The mapping of a query to its feature set is determined by the *query model*. Query models are problem-specific, and we describe the query model for the specific XML string selectivity estimation problem in Section 5.5.1.

A CXHist histogram consists of a set of $m$ buckets indexed by $\mathcal{B} = \{1, 2, \ldots, m\}$. Each bucket $b$ stores

1. $sum(b)$, the sum of the selectivities of all the query feedbacks seen so far that are associated with bucket $b$,
2. $count(b)$, one plus the number of query feedbacks seen so far that are associated with bucket $b$,
3. $\{P(X_i|B{=}b) : i = 1, \ldots, k\}$, a set of query feature probability distributions. One distribution is stored for each feature random variable $X_i$.

Each pair of $sum(b)$ and $count(b)$ fields stores an average selectivity for all the queries associated with the bucket $b$. Traditional bucket-based histograms for numerical attributes typically partition the data space into regions and associate a region of space with each bucket. CXHist differs from traditional bucket-based histograms in that feature probability distributions are stored in each bucket instead. These feature distributions (and associated algorithms to be described) form a Bayesian classifier.

Given a query $q$, CXHist estimates the selectivity $\sigma(q)$ by first computing the bucket $\hat{b}$ that is associated with $q$ using the query model and the feature distributions for each bucket. The selectivity of $q$ is then computed as

$$est(\hat{b}) = \frac{sum(\hat{b})}{count(\hat{b})}. \tag{5.2.1}$$

The feature distributions of a bucket $b$ approximately "remembers" which queries belong to bucket $b$. Each feature distribution $P(X_i{=}x_i|B{=}b)$ is not stored as probabilities, but using counts[1]. Let $N(X_i{=}x_i, B{=}b)$ be the number of times a particular feature has been observed to be associated with bucket $b$. The probabilities are recovered using,

$$P(X_i{=}x_i|B{=}b) = \frac{N(X_i{=}x_i, B{=}b)}{\sum_\xi N(X_i{=}\xi, B{=}b)}. \tag{5.2.2}$$

CXHist updates itself using query feedback. Suppose the query execution engine computes $\sigma(q)$ as the true selectivity for query $q$. CXHist finds the bucket $b^*$ whose selectivity is closest to $\sigma(q)$. If the computed bucket $\hat{b}$ is not the same as $b^*$, the feature distributions of bucket $b^*$ are updated so that CXHist will remember that query $q$ should be associated with bucket $b^*$.

The update process potentially adds new entries to the feature distributions and hence increases the memory used by CXHist. Whenever CXHist exceeds a given size threshold, the feature distributions are pruned so that the size of CXHist stays below

[1]Note that fractional counts are allowed for the purpose of updates.

the given threshold. Since the size threshold is set to a small constant (usually 1–2% of the data size), the space complexity of CXHist is effectively $O(1)$. The time complexity of the selectivity estimation, update and pruning procedures are therefore also effectively $O(1)$ because their time complexity are only dependent on the size of the CXHist histogram.

We give an introduction to Bayesian classifiers next, before describing each component of CXHist in greater detail.

### 5.2.2 Introduction to Bayesian Classifiers

A *classifier* is a mechanism for storing a mapping $f : \mathcal{X} \mapsto \mathcal{B}$ of a set of feature vectors $\mathcal{X}$ to a set of class labels $\mathcal{B}$ approximately. Given a classifier $f'$ and a feature vector $\vec{x} \in \mathcal{X}$, the class label of $\vec{x}$ can be approximated using the classifier as $f'(\vec{x})$.

A classifier can be built using a set of examples. An *example* is a pair consisting of an instance of a feature vector and its associated class label. The process of building a classifier takes as input a set of examples $\{(\vec{x}_1, f(\vec{x}_1)), (\vec{x}_2, f(\vec{x}_2)), \ldots, (\vec{x}_k, f(\vec{x}_k))\}$ and outputs the classifier $f'$ that approximates the true mapping $f$.

A *Bayesian classifier* $f'$ models the mapping $f : \mathcal{X} \mapsto \mathcal{B}$ as a joint probability distribution over the space of $\mathcal{X} \times \mathcal{B}$ [29]. Let $X_i$ denote the random variable for feature $i$ and $B$ denote the random variable for the class label. The joint probability density function over $\mathcal{X} \times \mathcal{B}$ is

$$P(X_1{=}x_1, X_2{=}x_2, \ldots, X_n{=}x_n, B{=}b)$$

where $b \in \mathcal{B}$ and $\langle x_1, x_2, \ldots, x_n \rangle \in \mathcal{X}$. For conciseness, we will use vector notation for a sequence of random variables and also omit the value for each random variable

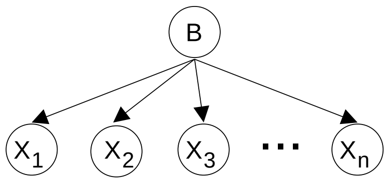


**Figure 5.2**: The graphical model corresponding the naive Bayes assumption.

when it is clear from context,

$$\begin{aligned} P(X_1{=}x_1, X_2{=}x_2, \ldots, X_n{=}x_n, B{=}b) &\equiv P(\vec{X}{=}\vec{x}, B{=}b) \\ &\equiv P(\vec{X}, B) \end{aligned}$$

Given a vector of feature values $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$, a Bayesian classifier $f'$ classifies $\vec{x}$ into the class label $b$ that maximizes the *a posteriori* probability,

$$f'(\vec{x}) = \arg\max_b P(B{=}b|\vec{X}{=}\vec{x}).$$

A *naive Bayesian classifier* is a Bayesian classifier that makes the *naive Bayes assumption* on the joint probability distribution. The naive Bayes assumption states that features $X_i, X_j, i \neq j$ are conditionally independent given the class label $B$, ie,

$$P(X_1, X_2, \ldots, X_n|B) = \prod_{i=1}^{n} P(X_i|B).$$

In the context of graphical models, the naive Bayes assumption imposes the conditional independence model of Figure 5.2 on the joint probability distribution. Using Bayes rule and the naive Bayes assumption, the *a posteriori* probability can be computed as,

$$\begin{aligned} P(B{=}b|X_1, X_2, \ldots, X_n) &= \frac{P(X_1, X_2, \ldots, X_n|B{=}b)P(B{=}b)}{\sum_{b'} P(X_1, X_2, \ldots, X_n|B{=}b')P(B{=}b')} \\ &= \frac{\left\{\prod_{i=1}^{n} P(X_i|B{=}b)\right\}P(B{=}b)}{\sum_{b'}\left\{\prod_{i=1}^{n} P(X_i|B{=}b')\right\}P(B{=}b')} \end{aligned} \tag{5.2.3}$$

Note that the denominator is a constant with respect to the random variable $B$ for class label and therefore can be omitted during the classification computation because it does not affect the maximal point in the classification formula,

$$f'(\vec{x}) = \arg\max_b \left\{ P(B{=}b) \prod_{i=1}^{n} P(X_i|B{=}b) \right\}$$

The classification formula in (5.2.3) requires only $P(X_i|B)$ for each $i = 1, 2, \ldots, n$ and $P(B)$. Hence, to construct a naive Bayesian classifier from a set of $k$ examples, we count for each distinct feature-label pair $(x_i, b)$ the number of times it occurs in the set of examples. Let $N(X_i{=}x_i, B{=}b)$ denote the number of examples in which feature $X_i$ takes the value $x_i$ and the class label $B$ takes the value $b$. The required probabilities can be computed from

$$P(X{=}x_i|B{=}b) = \frac{N(X_i{=}x_i, B{=}b)}{N(B{=}b)}$$

and

$$P(B{=}b) = \frac{N(B{=}b)}{k}.$$

where $N(B{=}b)$ is the number of examples with class label $b$. Note that $N(B{=}b)$

| $X_1$ | $X_2$ | $B$ |
|---|---|---|
| AddisonW | Knuth | 1 |
| Houghton | Tolkien | 0 |
| AddisonW | Lamport | 1 |
| Cambridge | Knuth | 1 |
| Ballantine | Tolkien | 0 |

(a) Training examples.

| $X_1$ | $B$ | $N(X_1, B)$ |
|---|---|---|
| AddisonW | 1 | 2 |
| Houghton | 0 | 1 |
| Cambridge | 1 | 1 |
| Ballantine | 0 | 1 |

(b) $N(X_1, B)$

| $X_2$ | $B$ | $N(X_2, B)$ |
|---|---|---|
| Knuth | 1 | 2 |
| Tolkien | 0 | 2 |
| Lamport | 1 | 1 |

(c) $N(X_2, B)$

**Figure 5.3**: The training examples and the corresponding naive Bayesian classifier for Example 5.12.

can be computed from $N(X_i{=}x_i, B{=}b)$ by

$$N(B{=}b) = \sum_x N(X_i{=}x, B{=}b).$$

We illustrate the use of a naive Bayesian classifier with an example.

**Example 5.12** *Suppose a librarian needs to shelve a pile of books onto two shelves. Shelf 0 is for fiction and shelf 1 is for non-fiction. Not particularly fond of manual labor, he decides to automate the classification task using a naive Bayesian classifier. Two features were chosen: $X_1$ represents the name of the publisher, and $X_2$ represents the name of the author. The shelf number is represented by random variable $B$. He manually classified five books to be used as training examples (see Figure 5.3(a)). The naive Bayesian classifier constructed from these training examples consists of the tables $N(X_1, B)$ and $N(X_2, B)$ as shown in Figure 5.3(b) and*

*Figure 5.3(c).*

*To classify a book authored by "Knuth" and published by "AddisonW", the classifier computes the a posteriori probability given those features,*

$$\begin{aligned}
&P(B{=}0|X_1{=}AddisonW, X_2{=}Knuth)\\
&\qquad\propto \quad P(X_1{=}AddisonW|B{=}0)P(X_2{=}Knuth|B{=}0)P(B{=}0)\\
&\qquad= \quad \frac{0}{2}\times\frac{0}{2}\times\frac{2}{5}=0\\
&P(B{=}1|X_1{=}AddisonW, X_2{=}Knuth)\\
&\qquad\propto \quad P(X_1{=}AddisonW|B{=}1)P(X_2{=}Knuth|B{=}1)P(B{=}1)\\
&\qquad= \quad \frac{2}{3}\times\frac{2}{3}\times\frac{3}{5}=\frac{4}{15}.
\end{aligned}$$

*Since shelf 1 maximises the a posteriori probability, the classifier returns shelf 1 (non-fiction) as the answer.*

Note that the classifier is ambiguous (the *a posteriori* probability is uniformly zero) when it encounters an author or publisher that did not appear in the training examples. A default shelf or a special 'unknown' flag can be returned for such cases.

We now show how a naive Bayesian classifier can be used in CXHist for selectivity estimation.

### 5.2.3 Estimating Selectivity with CXHist

Given a query $q$, selectivity estimation is done by determining which bucket the query belongs to. The mapping between a query and its bucket is modeled as a joint probability distribution with the random variable $B$ for the bucket and a series of random variables, $\vec{X}=\langle X_1,\ldots,X_k\rangle$, for the features of the query (under a given query model). Suppose that the query model maps the query $q$ into a set of features $\vec{x}=\langle x_1,\ldots,x_k\rangle$, then the bucket $\hat{b}$ that $q$ belongs to is computed as the bucket that maximizes the posterior probability of the query belonging to a bucket $b$ given its feature values,

$$\begin{aligned}\hat{b} &= \arg\max_{b\in\mathcal{B}}\left\{P(B=b|\vec{X}=\vec{x})\right\} &(5.2.4)\\ &= \arg\max_{b\in\mathcal{B}}\left\{P(B=b)P(\vec{X}=\vec{x}|B=b)\right\}, &(5.2.5)\end{aligned}$$

where the second equality is obtained using Bayes rule and by simplifying the denominator into a constant that is independent of $b$ and hence does not affect the maximal point. In the machine learning literature, this technique is known as a Bayesian classifier [29] where each bucket is the class label. A *naive Bayesian classifier* further simplifies the conditional distribution $P(\vec{X}=\vec{x}|B=b)$ by assuming conditional independence among the features given the bucket (the *naive Bayes assumption*),

$$P(X_1,X_2,\ldots,X_n|B=b) = \prod_{i=1}^{n} P(X_i|B=b). \qquad (5.2.6)$$

Therefore, a naive Bayesian classifier computes the bucket of a set of feature $\vec{x}$ using,

$$\hat{b}=\arg\max_{b\in\mathcal{B}}\left\{P(B=b)\prod_{i=1}^{n}P(X_i=x_i|B=b)\right\}. \qquad (5.2.7)$$

Naive Bayesian classifiers have been shown to be very accurate even for datasets with strong dependencies among the features [28, 51]. The probabilities $\{P(X_i|B=b):\forall i\}$ are precisely those stored in each bucket and the probability $P(B=b)$ can be computed as,

$$P(B{=}b) = \frac{count(b) - 1}{\sum_{b'}(count(b') - 1)}. \tag{5.2.8}$$

Once the bucket $\hat{b}$ for query $q$ is computed, the selectivity of $q$ is then computed as $est(\hat{b})$. There is a boundary case when the posterior distribution is flat and no maximum point exists. In this case, a default selectivity such as the minimum can be returned.

Observe that if a particular feature $X_j$ has uniform probability $P(X_j{=}x|B{=}b)$ over the space of $B$ for each value $x$, the bucket computed by a naive Bayesian classifier using Equation 5.2.7 will be unaffected by the exclusion of feature $X_j$. Such features are uninformative and could be excluded from the query model without any loss in classification accuracy.

### 5.2.4 Initializing CXHist

Given a fixed query model, a fixed number of buckets $m$, the minimum selectivity $l$, and the maximum selectivity $h$, a CXHist histogram is initialized with $m$ buckets, such that, for each bucket $b$, the bucket count $count(b)$ is set to one, the feature distributions are set to empty, and the $sum(b)$ field is initialized with a *representative* value from the interval $[l, h]$. Recall that each bucket $b$ in CXHist is associated with an average selectivity encoded using $sum(b)$ and $count(b)$. Each $count(b)$ field is initialized to one, because initializing it with zero would result in an undefined average selectivity. The initial value of the $sum(b)$ field will determine the initial representative average selectivity of the bucket.

The set of representative values used to initialize each $sum(b)$ field can be chosen in several ways.

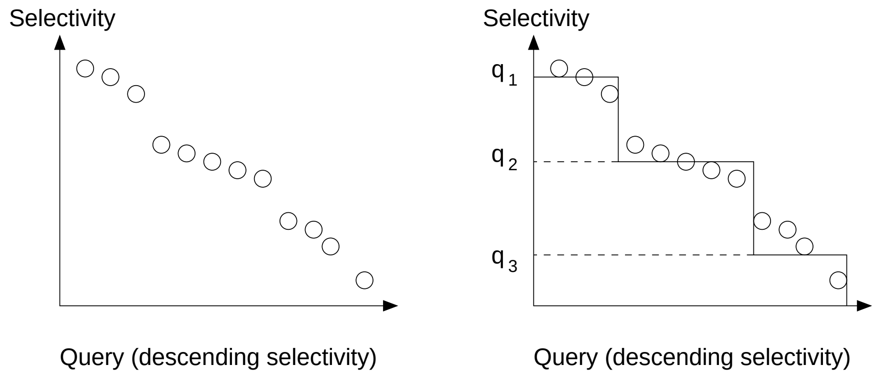


**Figure 5.4**: Initializing CXHist with a sample workload of 12 queries using clustering. The left diagram plots the selectivity of the queries (denoted by the hollow circles) in the sample workload sorted by descending selectivity. The diagram on the right shows the representative selectivity values $\{q_1, q_2, q_3\}$ if three buckets are to be used.

1. *Clustering.* If a sample query workload is available during initialization, the MaxDiff [68] algorithm or the Lloyd-Max algorithm [57, 60] can be used. See Figure 5.4 for an example.

2. *Uniform intervals.* If a query workload is not available, the representative values can be picked to be uniformly spaced in the given interval.

3. *Exponential intervals.* An alternative to uniform intervals is exponentially increasing intervals (e.g. $\{1, 2, 4, 8, 16, \ldots\}$).

4. *Uniform-exponential hybrid.* If the number of buckets $m$ is larger than $\log_2(h-l)$, the smallest $j$ representative values, where $j$ is a tunable parameter, are chosen to be exponentially increasing and the remaining values uniformly spaced in the rest of the interval,

$$sum(b) = \begin{cases} l \times 2^{b-1} & b \leq j \\ l \times 2^{j-1} + (b-j)\frac{h-l\times 2^{j-1}}{m-j} & j < b < m \end{cases} \tag{5.2.9}$$

For example, if $m$=10 and $j$=5, the representative values for the interval $[1, 66]$

are $\{1, 2, 4, 8, 16, 26, 36, 46, 56, 66\}$.

In our experiments we have used the uniform-exponential hybrid method for choosing the representative values.

Picking exponentially increasing representative values for the smaller selectivities might give better estimates in terms of relative error. For the intuition, consider the hypothetical case in which CXHist remembers the mapping between queries and their associated bucket perfectly and assume that the representative values in the buckets do not change over time (i.e., no updates). If the representative values are chosen to be exponentially increasing, the relative error of each estimate is no more than 50%. On the other hand, if uniformly spaced representative values are chosen, the absolute error of each estimate will be bounded by the distance between consecutive representative values.

### 5.2.5 Updating the CXHist Histogram

CXHist is an on-line histogram: once it has been initialized, it updates itself after each query using query feedback. We discuss how this update is performed in this section.

Given a query feedback, i.e. a $\langle q, \sigma(q)\rangle$ pair, we want to update the histogram so that it will give a better estimate when it encounters query $q$ again. For our discussion assume that query $q$ maps to the feature vector $\vec{x} = \langle x_1, \ldots, x_k\rangle$. The update algorithm is outlined in Algorithm 5.2 and it requires a subroutine AddInstance$(b, \langle x_1, \ldots, x_k\rangle)$ outlined in Algorithm 5.1 that increments all the feature distribution entries associated with each given feature value $x_i$ in the given bucket $b$.

First, find the bucket $b^*$ whose estimate $est(b^*)$ is closest to $\sigma(q)$ (see Algo-

**Algorithm 5.1** AddInstance ( $b, \langle x_1, \ldots, x_k \rangle$)

```
INPUT: bucket index b and query feature vector $\langle x_1, \ldots, x_k \rangle$
for all $i \in \{1, \ldots, k\}$ do
    if $\nexists$ entry $N(X_i{=}x_i|B{=}b)$ then
        add new entry $N(X_i{=}x_i|B{=}b) = 0$
    end if
    increment $N(X_i{=}x_i|B{=}b)$
end for
```

**Algorithm 5.2** Update ( $\langle q, \sigma(q) \rangle, \langle x_1, \ldots, x_k \rangle, \gamma$)

```
INPUT: query feedback $\langle q, \sigma(q) \rangle$, feature vector $\langle x_1, \ldots, x_k \rangle$ for query q, learning
    rate $\gamma$
$b^* \leftarrow \arg\min_b\{|\sigma(q) - est(b)|\}$
$sum(b^*) \leftarrow sum(b^*) + \sigma(q)$
increment $count(b^*)$
compute $\hat{b}$ using Equation (5.5.19).
if $\nexists \hat{b}$ or $b^*{=}\hat{b}$ then
    AddInstance($b^*, \langle x_1, \ldots, x_k \rangle$)
else
    $\hat{p} \leftarrow P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b})/P(B{=}b^*)$
    $p^* \leftarrow P(\vec{X}{=}\vec{x}|B{=}b^*)$
    if $p^*{=}0$ then
        AddInstance($b^*, \{x_1, \ldots, x_k\}$)
        $p^* \leftarrow P(\vec{X}{=}\vec{x}|B{=}b^*)$
    end if
    while $p^* < \hat{p}$ do
        for all $i \in \{1, \ldots, k\}$ do
            update $N(X_i{=}x_i|B{=}b^*)$ using Equation (5.2.17)
        end for
        recompute $p^*$
    end while
    if $p^* = \hat{p}$ then
        AddInstance($b^*, \langle x_1, \ldots, x_k \rangle$)
    end if
end if
```

rithm 5.2 line 1). Second, update the bucket $b^*$ by incrementing $count(b^*)$ and adding $\sigma(q)$ to $sum(b^*)$ (lines 2-3). Third, update the feature distributions of bucket $b^*$. If either no maximum point is found when estimating the selectivity of $q$ or the closest bucket $b^*$ is the same as the bucket that maximizes the posterior probability $\hat{b}$, i.e., no classification error, then increment the count $N(X_i{=}x_i|B{=}b^*)$ for each $i = 1, \ldots, k$ (lines 5-6). Otherwise, there is a classification error, i.e., $b^* \neq \hat{b}$. We want to update the feature distributions such that $q$ will be classified into the bucket $b^*$ instead of $\hat{b}$. Specifically, the desired condition is

$$P(B{=}b^*)P(\vec{X}{=}\vec{x}|B{=}b^*) \quad > \quad P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b}) \tag{5.2.10}$$

$$\equiv P(\vec{X}{=}\vec{x}|B{=}b^*) \quad > \quad \frac{P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b})}{P(B{=}b^*)}. \tag{5.2.11}$$

A naive method for updating the feature distributions is to keep incrementing the count $N(X_i{=}x_i|B{=}b^*)$ for each $i = 1, \ldots, k$ until the above condition is met.

A more principled method for getting the same effect is to perform *gradient descent* [70] until the required condition is met (line 14). The number of iterations can be bounded by a specified constant to prevent the `while`-loop (line 14) from taking too long. For gradient descent, we define the error of a selectivity estimate as,

$$\epsilon(\vec{x}) = P(\vec{X}{=}\vec{x}|B{=}b^*) - \hat{p}, \tag{5.2.12}$$

and the error function to be minimized as,

$$E(\vec{x}) \quad = \quad \Big[\epsilon(\vec{X}{=}\vec{x})\Big]^2 \tag{5.2.13}$$

$$= \quad \Big[P(\vec{X}{=}\vec{x}|B{=}b^*) - \hat{p}\Big]^2, \tag{5.2.14}$$

where $\hat{p}{=}P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b})/P(B{=}b^*)$ is a constant target value. Let $w_i$ be a shorthand for the count $N(X_i{=}x_i, B{=}b^*)$. For each feature value $x_i$ of the query, the gradient descent method updates the count $w_i$ using the negative gradient,

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \gamma \frac{\partial E(\vec{x})}{\partial w_i}, \tag{5.2.15}$$

where $w_i^{(t)}$ and $w_i^{(t+1)}$ are the counts before and after the update respectively, and $\gamma$ is the learning rate parameter. We will adopt the shorthand $\Delta w_i$ for the derivative $\partial E(\vec{x})/\partial w_i$. Simplifying the derivative,

$$\Delta w_i \;=\; 2\epsilon(\vec{x}) \frac{\partial P(\vec{X}{=}\vec{x}|B{=}b^*)}{\partial w_i}, \tag{5.2.16}$$

the update equation becomes,

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - 2\gamma\epsilon(\vec{x}) \frac{\partial P(\vec{X}{=}\vec{x}|B{=}b^*)}{\partial w_i}, \tag{5.2.17}$$

where the derivative $\partial P(\vec{X}{=}\vec{x}|B{=}b^*)/\partial w_i$ depends on the query model used in the actual problem.

Two boundary cases need to be addressed. The first boundary case occurs before the gradient descent iteration when $w_i{=}0$ for some $i$ and hence $P(\vec{X}{=}\vec{x}|B{=}b^*){=}0$ and the gradient may be undefined. This situation is avoided by incrementing $N(X_i{=}x_i, B{=}b^*)$ for $i = 1, \ldots, k$ (lines 10-12). The other boundary case occurs after the gradient descent loop when $P(\vec{X}{=}\vec{x}|B{=}b^*) = \hat{p}$. The desired condition is $P(\vec{X}{=}\vec{x}|B{=}b^*) > \hat{p}$, so the feature distributions in $b^*$ are incremented once more to break the tie (line 20).

One problem with using gradient descent in this form is that the updates $\Delta w_i$

can be very small (less than $10^{-10}$) when the number of feature values are large. We address this problem by computing the updates $\Delta w_i$ for all $i$ and normalizing them so that the smallest $\Delta w_i$ is one. The learning rate is then applied onto the normalized updates.

### 5.2.6 Pruning CXHist

The size of a CXHist histogram is the sum over each bucket $b$ of the size of the *sum*($b$) field, the *count*($b$) field and the feature distributions $\{N(X_i|B{=}b)\}$. The initialization and selectivity estimation procedures do not increase the size of the histogram. The update procedure potentially adds new entries to the feature distributions. After some number of updates, the CXHist histogram might need to be pruned so that it does not exceed the memory allocated to it.

Pruning is governed by two parameters both in units of bytes: the *triggersize* and the *targetsize*. Whenever the size of the histogram exceeds *triggersize* bytes, the pruning procedure is activated and the CXHist histogram is pruned down to *targetsize* bytes.

Pruning is achieved by discarding entries with small counts in the feature distributions. All feature distribution entries $\{N(X_i{=}x_i|B{=}b) : \forall i, x_i, b\}$ are sorted by magnitude and we keep discarding the smallest entry until the desired target size is achieved. Mathematically, discarding an entry is the same as setting the magnitude of that entry to zero. The entries with the smaller counts are good candidates for pruning because (1) they are likely to be less frequently used, otherwise the update procedure would have incremented their counts, and (2) they are less likely to affect the maximal point in the selectivity estimation computation, since their probabilities are already small.

## 5.3 XML String Selectivity Estimation

For the rest of the chapter, we apply CXHist to a specific selectivity estimation problem in XML databases: estimating the number of leaf nodes reachable via a given path and whose associated data value matches a given string constraint. We motivate and describe the problem in this section.

Recall that in XML native DBMSs, XML documents are stored as trees (see Figure 5.5) and queries are processed using either indexes or tree traversals (a more detailed exposition about query optimization in XML databases has been given in Chapter 4 Section 4.1.1). A path is a sequence of tag names that specify a navigational trajectory to a set of nodes in the XML tree. A rooted path is one that begins at the root node of the XML tree. For efficient processing, complex path expressions in XML queries are often preprocessed into a set of candidate $\langle path, pred \rangle$ query pairs, where *path* is a linear rooted path and *pred* is a string predicate on the leaf value reachable via *path*. Consequently, an XML query (such as XQuery) can be mapped to several retrieval operations using $\langle path, pred \rangle$ query pairs [44, 76, 87, 89]. For the example data in Figure 5.5, the path expres-

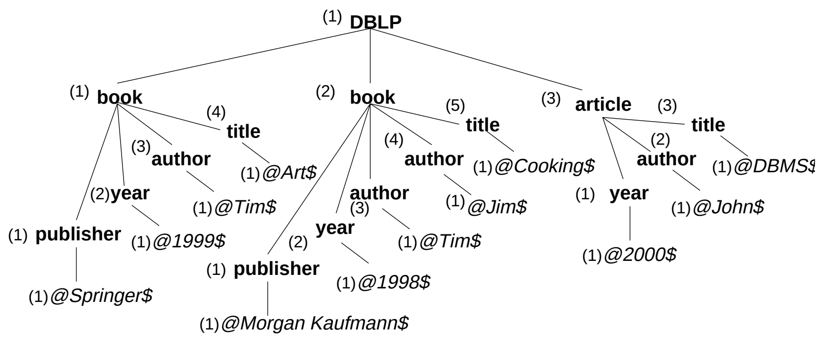


**Figure 5.5**: An example XML data tree. Tag names are in bold and data values are in italics. The special symbols '@' and '$' denote the start and end of data values respectively.

sion `//author=@Tim$`[2] can be mapped to query pairs ⟨`/DBLP/book/author`, `@Tim$`⟩ and ⟨`/DBLP/article/author`, `@Tim$`⟩, where the string predicate is the exact match predicate. These retrieval operations using ⟨*path*, *pred*⟩ query pairs form the set of basic query processing operators. Accurate estimation of the selectivity of such ⟨*path*, *pred*⟩ query pairs is therefore crucial for choosing an optimal execution plan in cost-based query optimization.

The XML string selectivity estimation problem is defined as follows: Given a ⟨*path*, *pred*⟩ query, where *pred* is a string predicate, estimate the number of nodes in the XML data that are reachable by *path* and whose associated string values satisfy the string predicate *pred*. Examples of queries with different string predicates include exact match queries and substring queries. An *exact match query* is specified by a ⟨*path*, *string*⟩ pair and retrieves all nodes reachable via *path* whose string value matches the query string exactly. A *substring query* is specified by a ⟨*path*, *substring*⟩ pair and retrieves all nodes reachable via *path* whose string value contains *substring*. Using the example XML tree in Figure 5.5, the selectivity of exact match query ⟨`/DBLP/book/author`, `@Tim$`⟩ is 2 and the selectivity of substring query ⟨`/DBLP/book/author`, `im$`⟩ is 3.

The string aspect of the XML string selectivity estimation problem is especially important to XML DBMSs for two reasons. First, XML data are weakly-typed and therefore XML databases must support the retrieval of XML data as string data, even though some data could be treated as numeric data as well. For numeric leaf values, selectivity estimation can be easily done using traditional histograms and this approach has been adopted by [35, 67]. Second, the set of distinct path-string pairs in an XML tree is typically orders of magnitude larger than the set of distinct

[2]The symbols '@' and '$' denote the start and end of a string respectively.

rooted paths. For example, the DBLP XML data set has 2,026,814 distinct path-string pairs, but only 275 distinct rooted paths. The small number of paths means that the selectivity of all distinct rooted paths can be easily stored in an index using little space[3]. The selectivity of distinct path-string pairs, however, would require more sophisticated approximation techniques.

Three unique challenges make XML string selectivity estimation a nontrivial problem:

1. How to capture substring statistics accurately in limited storage?
2. How to capture the correlation between paths and the substring statistics accurately?
3. How to support a broad class of query types?

Capturing substring statistics for selectivity estimation in relational databases has been studied in [15, 41–43, 49, 82]; however, the accuracy of these proposed solutions is far from satisfactory [15]. The problem is hard because the set of distinct strings is large and the corresponding set of possible substrings is even larger (by orders of magnitude). Storing the selectivity of each distinct string or substring is clearly infeasible and more sophisticated approximation techniques are required to capture the substring statistics.

XML string selectivity estimation differs from relational substring selectivity estimation in that a correlated path (whether implicitly encoded as a path ID or explicitly as a XPath) is associated with the query substring. Hence, the XML string selectivity estimation problem is harder than the relational substring selectivity estimation problem, because the correlation between path and substring statistics

[3]In the case that the number of distinct rooted paths is significantly large, techniques based on the Markov assumption and/or bi-similarity can be used [1, 17, 55, 66, 67].

needs to be captured as well. Previous work on XML selectivity estimation has emphasized the selectivity of navigational paths and tree structures [1,17,35,55,66, 67] over the selectivity of string predicates on leaf values. While correlated subpath trees (CSTs) [17] and XPathLearner [55] do address the correlation between paths and their associated string values in a limited way, XPathLearner does not capture substring statistics and CSTs suffer from the underestimation problem [15]. The problem of accurately capturing substring statistics that are correlated with paths has not been adequately addressed.

As a consequence of the flexibility of XML, XML query languages, and XML applications, XML DBMSs need to support a broad class of query types (exact match, substring, prefix, suffix, etc.). Two particularly important query types are exact match queries and substring queries. Exact match queries occur frequently in transaction processing and in application generated queries, whereas substring queries occur frequently in user generated queries, because users do not usually remember a text string exactly. While separate statistics and techniques can be used for each query type, a single estimation framework that is accurate for a broad class of query types is preferable. Previous work that supports string predicates is very restrictive in this aspect. The CST method [17] supports substring queries, but is prone to underestimation for exact match queries [15]. The XPathLearner [55] method supports exact match queries, but cannot handle substring queries at all. The challenge then is a single framework for the accurate estimation of the selectivity of a broad class of string predicates.

The XML string selectivity problem is even more challenging when we impose on-line requirements:

1. no off-line scans of underlying data,

2. adaptivity to query workload characteristics, and
3. adaptivity to changes in underlying data.

With the exception of [55], all existing techniques suffer from several limitations of off-line techniques: they require expensive scans over the original data to gather statistics, they do not adapt to workload characteristics, and they do not adapt to changes in the underlying data.

We propose using CXHist for estimating the selectivity of exact match and substring queries in XML databases. CXHist is a novel on-line selectivity estimation method that is capable of supporting selectivity estimation on a broad class of query types. CXHist captures accurate path-correlated statistics not from costly off-line scans of the underlying data, but from *query feedback*, i.e. past query answers (see Figure 5.1). Consequently CXHist is able to adapt to changes in the query workload characteristics and in the underlying data.

The rest of the chapter is organized as follows. Related work is discussed in the next section. Section 5.2 presents the CXHist customized for the XML string selectivity estimation problem. Experimental results are given in Section 5.6. For ease of exposition, we make several unnecessary assumptions that will simplify our description of CXHist. CXHist will be presented in the context of exact match and substring queries, even though CXHist is applicable to a broad class of query types. We will also assume that rooted paths have been mapped to path identifiers (pathIDs), because our main focus is on string predicates. This assumption is motivated by the fact that many XML DBMSs map rooted paths to pathIDs for more efficient data storage and query processing [44,76,87,89]. If the set of distinct paths is too large to be mapped to a set of pathIDs, a Markov model can be used to model the set of paths explicitly [1,55] in the query model.

## 5.4 Related Work

Sub-string selectivity estimation for relational databases has been studied in [41–43, 49, 82]. All these techniques rely on a pruned suffix tree (PST) as the summary data structure. Note that to adapt these techniques for XML selectivity estimation, one PST will be required for each distinct path. PST-based methods typically parse a query string into possibly (overlapping) component substrings that have an entry in the PST. The counts of the component substrings are combined using either complete independence or Markov assumptions. These assumptions often result in large underestimation errors [15]. CXHist is a new type of histogram that is different from a PST and therefore do not suffer from the same limitations.

Recently, Chaudhuri et al. [15] proposed using a regression tree on top of a basic string estimation technique such as a PST to correct the underestimation problem often associated with independence and Markov assumptions. While modeling of estimation errors can always be used to reduce or correct the errors in a basic string estimation technique, a simpler, more elegant solution is preferable. CXHist is based on a new paradigm using Bayesian classifiers. CXHist does not combine constituent substring selectivities using the independence or Markov assumptions; therefore CXHist does not share the same limitations as PST-based methods.

XML selectivity estimation has also been studied in [1,17,35,55,66,67,84]. These methods can be categorized by whether they are on-line or off-line, whether they handle subtree queries or path queries only, whether they handle leaf values, and whether the leaf values are assumed to be strings or numbers. One major difference between our use of CXHist and previous work is that previous work has focused on queries on the navigational paths or subtrees, whereas our use of CXHist focuses on string-based queries on the leaf values reachable via a given path.

The correlated subpath tree (CST) method [17, 18] is an augmented PST constructed from the XML data by treating element tags as atomic alphabets and the leaf values as strings. The CST method is off-line, handles subtree queries, and supports substring queries on the leaf values. Being a PST-based technique, the CST method suffers from the underestimation problem. CXHist is a new histogram technique that overcomes this limitation.

The Markov Table (MT) method of [1] uses a fixed order Markov model to capture sub-path statistics in XML data. Queries on leaf values are not supported. The XPathLearner method [55] extends the MT method [1] to an on-line method. In addition, XPathLearner supports exact match queries on leaf values using a form of compressed histogram to store the tag-value pair distribution. However, XPathLearner cannot handle substring queries at all. CXHist is a new type of histogram (not based on the Markov model) that can handle a broad class of query types including substring queries.

The XSketch summary structure [66] is a variable order Markov model constructed by performing a greedy search for best fitting model. XSketch as presented in [66] handles subtree queries on element tags only. The extension presented in [67] handles numeric leaf values using standard histograms – string predicates on leaf values are not supported. Our use of CXHist assumes that ambiguity in navigational path has been resolved to a set of rooted paths and addresses string-based queries on the leaf values instead.

A different approach is taken by StatiX [35]. StatiX exploits structural information in XML schemas to build histograms that support subtree queries on numerical leaf values. Our use of CXHist addresses string-based queries on leaf values.

Position histograms [84] encode the ancestor-descendent relationship of each

XML node using points on a 2D plane and builds a histogram for those points. Position histograms specifically address ancestor-descendent queries, whereas CXHist is used to address string-based queries on leaf values.

The concept of using query feedback to build histograms has been used in [2,12] for relational, numerical data and in [55] for XML path expressions. CXHist uses query feedback in a similar way, but builds a different kind of histogram for string data.

## 5.5 CXHist for XML String Selectivity Estimation

### 5.5.1 Modelling XML Queries

To use CXHist histograms for estimating exact match and substring queries, a query model for the two types of queries needs to be defined. A query model $\mathcal{M}(q)=\vec{x}$ maps a query to a feature set. A good query model should include features that distinguish queries associated with different buckets with respect to the classifier. Conversely, a good query model should exclude features that do not distinguish queries associated with different buckets: a feature $X_i$ whose probability $P(X_i=x|B=b)$ is uniform over the space of $B$ for each value $x$ would not affect the classification computation and hence can be excluded without affecting classification accuracy.

For the substring or exact match queries that we consider, our query model maps each query to a pathid and all the $n$-grams of the query (sub)string, where $n$ is a tunable parameter. In terms of probability modeling, each query is modeled as a random variable $T$ for the pathID and a series of random variables $\{G_1, G_2, \ldots, G_k\}$

for the sequence of $n$-grams of the query (sub)string. The distinction between substring and exact match queries are implicitly encoded using a start of string symbol '`@`' and an end of string symbol '`$`'. For example, an exact match query (5, `@LIM$`) is mapped to ⟨5, `@L`, `LI`, `IM`, `M$`⟩ for $n = 2$.

### 5.5.2 Estimating String Selectivity

Using the query model described in Section 5.5.1, each substring or exact match query is modeled as a random variable $T$ for the pathID and a series of random variable $\{G_1, G_2, \ldots, G_k\}$ for the sequence of $n$-grams of the query (sub)string. In addition to conditional independence (the naive Bayes assumption), we assume stationarity of the $n$-gram distribution,

$$P(G_i|B) = P(G|B) \quad \forall i. \tag{5.5.18}$$

The $n$-gram model and the stationarity assumption were first used by Shannon [74] for predicting English text and has since been widely used in text modeling. Stationarity allows us to store just one $n$-gram distribution per bucket instead of one distribution for each position in a string. Hence, given a query with feature values $\langle t, g_1, g_2, \ldots, g_k \rangle$, its associated bucket is computed as,

$$\hat{b} = \arg\max_{b \in \mathcal{B}} \left\{ P(B{=}b) P(T{=}t|B{=}b) \prod_{i=1}^{k} P(G{=}g_i|B{=}b) \right\}, \tag{5.5.19}$$

and the selectivity of $q$ is estimated as $est(\hat{b}) = sum(\hat{b})/count(\hat{b})$.

### 5.5.3 Updating the CXHist Histogram

In Section 5.2.5 we have discussed gradient descent update for CXHist without assuming a specific query model. We now show the update equations for the the query model we adopted for substring and exact match queries. To simplify the notation, we will use the following shorthand,

$$\begin{aligned} p(t, g_1, \ldots, g_k) &= P(T{=}t|B{=}b^*) \prod_{i=1}^{k} P(G{=}g_i|B{=}b^*), \\ w_x &= N(X{=}x, B{=}b^*), \\ W_X &= \sum_x N(X{=}x, B{=}b^*), \end{aligned}$$

where $X$ is a feature random variable (either $T$ or $G$). The squared error function for gradient descent is,

$$\begin{aligned} E(t, g_1, \ldots, g_k) &= [\epsilon(t, g_1, \ldots, g_k)]^2 \\ &= [p(t, g_1, \ldots, g_k) - \hat{p}]^2 . \end{aligned} \tag{5.5.20}$$

For the pathID distribution, the update for $w_t$ is,

$$\begin{aligned} \Delta w_t &= \frac{\partial E(t, g_1, \ldots, g_k)}{\partial w_t} \\ &= 2\epsilon(t, g_1, \ldots, g_k) p(t, g_1, \ldots, g_k) \left( \frac{1}{w_t} - \frac{1}{W_T} \right). \end{aligned} \tag{5.5.21}$$

For the $n$-gram distribution, recall that we have assumed stationarity and that $\{g_1, \ldots, g_k\}$ are not necessarily all distinct; hence, for each distinct $n$-gram $g$ that occurs $\alpha$ times in $\{g_1, \ldots, g_k\}$, the update in $w_g$ is computed as,

$$\begin{aligned}\Delta w_g &= \frac{\partial E(t, g_1, \ldots, g_k)}{\partial w_g} \\ &= 2\epsilon(t, g_1, \ldots, g_k) p(t, g_1, \ldots, g_k) \left( \frac{\alpha}{w_g} - \frac{k}{W_G} \right). \end{aligned} \tag{5.5.22}$$

We now illustrate the use of CXHist with an example.

### 5.5.4 Example

Consider the following query workload along with the true selectivities:

| No. | Path ID | String | Selectivity |
|---|---|---|---|
| 1 | 0 | @LIM$ | 2 |
| 2 | 1 | @MIN | 20 |
| 3 | 0 | @LIM | 10 |
| 4 | 0 | @LIM$ | 2 |
| 5 | 0 | IM | 18 |

Note that query 2 and 3 are prefix queries, query 1 and 4 are exact match queries, and query 5 is a proper substring query.

Given that the minimum and maximum selectivity are 1 and 20 respectively, and that 5 buckets are to be used, we initialize the representative values as $1, 2, 4, 8, 16$. The *count*($b$) field for each $b$ is set to 1. The query model maps each query into a pathID random variable $T$, and a series of 2-gram random variables $G_i$. We further assume stationarity for the 2-grams and store only a single 2-gram distribution per bucket. The feature distributions are initially empty. We adopt the following shorthand for computing the posterior probability for a given query,

$$\begin{aligned}
L(b) &= P(B{=}b)P(T{=}t|B{=}b)\prod_{i=1}^{k} P(G{=}g_i|B{=}b), \\
\hat{p} &= \frac{P(B{=}\hat{b})}{P(B{=}b^*)}P(T{=}t|B{=}\hat{b})\prod_{i=1}^{k} P(G{=}g_i|B{=}\hat{b}) \\
p^* &= P(T{=}t|B{=}b^*)\prod_{i=1}^{k} P(G{=}g_i|B{=}b^*)
\end{aligned}$$

**Query 1.** CXHist gives a default estimate of 1 (50 % relative error) for the first query because all the feature distributions are empty and no maximal point exists. CXHist receives the true selectivity 2 as feedback and updates itself. The bucket with the closest estimate is bucket $b^*{=}2$ and we update bucket 2 by adding 2 to $sum(2)$ and incrementing $count(2)$. The feature distributions are initially empty, so AddInstance is called and the resulting CXHist is:

| $B$ | $sum(b)$ | $count(b)$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 4 | 2 |
| 3 | 4 | 1 |
| 4 | 8 | 1 |
| 5 | 16 | 1 |

| $B$ | $G$ | $N(G,B)$ |
|---|---|---|
| 2 | @L | 1 |
| 2 | LI | 1 |
| 2 | IM | 1 |
| 2 | M$ | 1 |

| $B$ | $T$ | $N(T,B)$ |
|---|---|---|
| 2 | 0 | 1 |

**Query 2.** The second query is processed similarly and the resulting feature distributions are shown as follows:

| $B$ | $sum(b)$ | $count(b)$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 4 | 2 |
| 3 | 4 | 1 |
| 4 | 8 | 1 |
| 5 | 36 | 2 |

| $B$ | $G$ | $N(G,B)$ |
|---|---|---|
| 2 | @L | 1 |
| 2 | LI | 1 |
| 2 | IM | 1 |
| 2 | M$ | 1 |
| 5 | @M | 1 |
| 5 | MI | 1 |
| 5 | IN | 1 |

| $B$ | $T$ | $N(T,B)$ |
|---|---|---|
| 2 | 0 | 1 |
| 5 | 1 | 1 |

**Query 3.** For the third query, only bucket 2 gives a non-zero posterior probability,

$$\begin{aligned} L(2) = \quad & P(B{=}2) \times P(T{=}0|B{=}2) \times P(G{=}\text{@L}|B{=}2) \\ & \times P(G{=}\text{LI}|B{=}2) \times P(G{=}\text{IM}|B{=}2) \\ = \quad & \frac{1}{2} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \quad = \quad 0.007813, \end{aligned}$$

therefore the selectivity is estimated as $est(2) = 2$ with a relative error of 400 %.

Using the query feedback, the bucket with the closest estimate is $b^* = 4$ and $count(4)$ and $sum(4)$ are updated accordingly. Since the feature distributions in bucket 4 are empty, we apply the subroutine AddInstance once and check if gradient descent is required,

$$\begin{aligned} \hat{p} = \quad & \frac{1/3}{1/3} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} = 0.015625, \\ p^* = \quad & 1 \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{3} = 0.037037. \end{aligned}$$

Since $p^* > \hat{p}$, no further updates are required . The resulting state of CXHist is:

| $B$ | $sum(b)$ | $count(b)$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 4 | 2 |
| 3 | 4 | 1 |
| 4 | 18 | 2 |
| 5 | 36 | 2 |

| $B$ | $T$ | $N(T,B)$ |
|---|---|---|
| 2 | 0 | 1 |
| 4 | 0 | 1 |
| 5 | 1 | 1 |

| $B$ | $G$ | $N(G,B)$ |
|---|---|---|
| 2 | @L | 1 |
| 2 | LI | 1 |
| 2 | IM | 1 |
| 2 | M$ | 1 |
| 4 | @L | 1 |
| 4 | LI | 1 |
| 4 | IM | 1 |
| 5 | @M | 1 |
| 5 | MI | 1 |
| 5 | IN | 1 |

**Query 4.** Only bucket 2 has a non-zero posterior probability,

$$L(2) = \frac{1}{3} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} = 0.001302, \tag{5.5.23}$$

so CXHist computes the estimate as $est(2) = 2$ with zero error. The update process applies the subroutine AddInstance once on the feature distributions of bucket 2. The resulting histogram is as follows.

| $B$ | $sum(b)$ | $count(b)$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 3 |
| 3 | 4 | 1 |
| 4 | 18 | 2 |
| 5 | 36 | 2 |

| $B$ | $T$ | $N(T,B)$ |
|---|---|---|
| 2 | 0 | 2 |
| 4 | 0 | 1 |
| 5 | 1 | 1 |

| $B$ | $G$ | $N(G,B)$ |
|---|---|---|
| 2 | @L | 2 |
| 2 | LI | 2 |
| 2 | IM | 2 |
| 2 | M$ | 2 |
| 4 | @L | 1 |
| 4 | LI | 1 |
| 4 | IM | 1 |
| 5 | @M | 1 |
| 5 | MI | 1 |
| 5 | IN | 1 |

**Query 5.** Computing the posterior probabilities,

$$\begin{aligned} L(1) = &\ L(3) = L(5) = 0 \\ L(2) = &\ \frac{2}{4} \times 1 \times \frac{2}{8} = 0.125 \\ L(4) = &\ \frac{1}{4} \times 1 \times \frac{1}{3} = 0.083333, \end{aligned}$$

CXHist concludes that bucket 2 maximizes the posterior probability and returns $est(2) = 2$ as the estimate (with 89 % relative error).

Using the query feedback, the bucket with the closest estimate is computed to be $b^* = 5$, and $count(5)$ and $sum(5)$ are updated accordingly. Since $L(5) = 0$, the subroutine AddInstance is applied once. The resultant state of the CXHist histogram is shown below:

| $B$ | $sum(b)$ | $count(b)$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 3 |
| 3 | 4 | 1 |
| 4 | 18 | 2 |
| 5 | 54 | 3 |

| $B$ | $T$ | $N(T,B)$ |
|---|---|---|
| 2 | 0 | 2 |
| 4 | 0 | 1 |
| 5 | 1 | 1 |
| 5 | 0 | 1 |

| $B$ | $G$ | $N(G,B)$ |
|---|---|---|
| 2 | @L | 2 |
| 2 | LI | 2 |
| 2 | IM | 2 |
| 2 | M$ | 2 |
| 4 | @L | 1 |
| 4 | LI | 1 |
| 4 | IM | 1 |
| 5 | @M | 1 |
| 5 | MI | 1 |
| 5 | IN | 1 |
| 5 | IM | 1 |

In order to determine whether gradient descent is required, the update procedure computes $\hat{p}$ and $p^*$,

$$\hat{p} = \frac{2/5}{2/5} \times 1 \times \frac{2}{8} = 0.25, \quad p^* = \frac{1}{2} \times \frac{1}{4} = 0.125.$$

Since $\hat{p}>p^*$, gradient descent is required and we compute the deltas for $w_t{=}N(T{=}0, B{=}5)$

and $w_q$=$N(G$=IM$, B$=5$)$ using Equation (5.5.21) and (5.5.22),

$$\Delta w_t = 2 \times (0.125 - 0.4) \times 0.125 \times (1 - \frac{1}{2}) = -0.034$$
$$\Delta w_q = 2 \times (0.125 - 0.4) \times 0.125 \times (1 - \frac{1}{4}) = -0.052.$$

Normalizing $(\Delta w_t, \Delta w_q)$ by the minimum absolute delta value, we have $(\Delta w_t = -1, \Delta w_q = -1.5)$. Using a learning rate of 1, CXHist updates the following,

$$\begin{aligned} N(T{=}0, B{=}5) &\leftarrow N(T{=}0, B{=}5) + 1, \\ N(G{=}\mathrm{IM}, B{=}5) &\leftarrow N(G{=}\mathrm{IM}, B{=}5) + 1.5. \end{aligned}$$

Recomputing $p^*$, we have

$$p^* = \frac{2}{3} \times \frac{2.5}{5.5} = 0.303030 > \hat{p},$$

and the gradient descent loop terminates.

## 5.6 Experiments

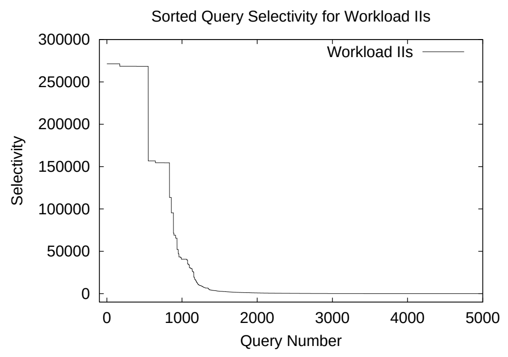
In this section, we present our experimental evaluation of CXHist for the XML string selectivity estimation problem. We study the effects of different parameters, evaluate the performance of CXHist under varying memory constraints and workload skew, and also investigate the stability of CXHist under changes in the workload characteristics.

| | | **s**ubstring | | | **e**xact match | | | **m**ixed | | |
|---|---|---|---|---|---|---|---|---|---|---|
| ID | $s$ | uniq | usize | size | uniq | usize | size | uniq | usize | size |
| I | 100K | 3,279 | 56,449 | 87,825 | 4,964 | 221,426 | 223,020 | 8,243 | 277,875 | 310,841 |
| II | 10K | 3,337 | 57,283 | 87,302 | 4,687 | 207,945 | 222,033 | 8,024 | 265,228 | 309,331 |
| III | 1K | 2,712 | 46,877 | 87,584 | 2,808 | 125,285 | 223,184 | 5,520 | 172,162 | 310,764 |
| IV | 100 | 1,175 | 20,674 | 88,000 | 522 | 24,583 | 239,024 | 1,697 | 45,257 | 327,020 |
| V | 10 | 253 | 4,484 | 87,793 | 67 | 3,436 | 265,349 | 320 | 7,920 | 353,138 |

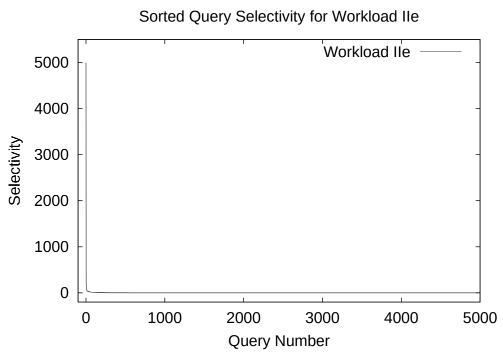
**Table 5.1**: Workload characteristics. The standard deviation of the Gaussian distribution used to generate the workload is given by column "$s$". The number and size of the distinct queries in the workload is given by the "uniq" and "usize" fields. The "size" field gives the size of the workload in bytes. Each exact match and substring workload has 5000 queries per workload and each mix workload has 10,000 queries.

**Data Set.** We have used the DBLP XML data (as of October 2003). The XML file has been preprocessed into a collection of $\langle pathid, string, count\rangle$ triples by hashing the rooted path associated with each leaf node into a numeric pathID and counting the occurrences of each distinct $\langle pathid, string\rangle$ pair. The total number of leaves is 5,045,240 of which there are 2,026,814 distinct $\langle pathid, string\rangle$ pairs. The most frequent $\langle pathid, string\rangle$ pair occurs 77,555 times.

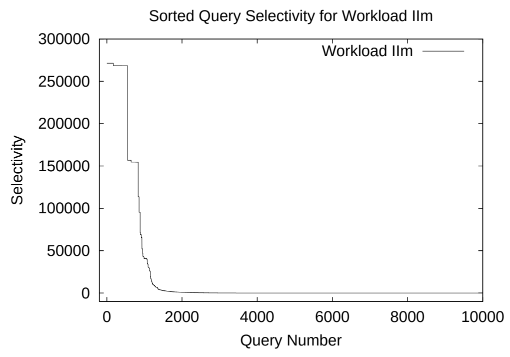
**Query workloads.** For each experiment we have used three types of workloads: substring match workload, exact match workload, and a mixed workload. Query workloads are generated using the collection of $\langle pathid, string, count\rangle$ triples. The $\langle pathid, string, count\rangle$ triples are first randomly permuted, so that the generated workload is not dependent of the initial order of the triples. Let the randomly permuted triples be indexed by $1, 2, \ldots, n$. We randomly pick an index $\mu$ as the mean or mode. Given a particular standard deviation $s$, we generate a discretized Gaussian distribution $G(n, \mu, s)$ over the indexed set, centered at $\mu$ with standard



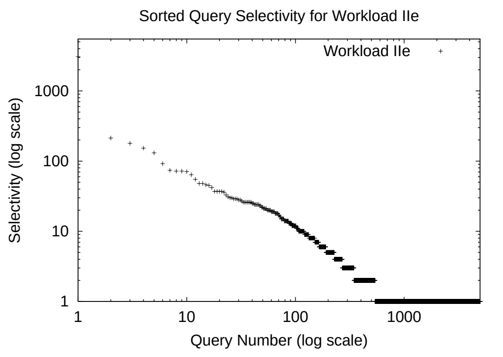
(a) IIs



(b) IIe



(c) IIm



(d) IIe (log scale)

**Figure 5.6**: Sorted query selectivities of Workload IIs, IIe, and IIm. The plot for IIe is shown in log scale to highlight the similarity to a Zipf distribution.

deviation $s$. To generate a query in a query workload, we generate a random index[4] $i$ from the discretized distribution $G(n, \mu, s)$. If we are generating an exact match query, the generated query consists of the pathid and text string from the $i$-th triple. If we are generating a substring query, the generated query consists of the pathid from the $i$-th triple and a substring token generated by uniformly randomly picking a token from the text string of the $i$-th triple. A substring query is generated only

[4]This can be accomplished by generating a uniform random variate $u \in [0, 1]$ and then finding the index $i$ such that $P(I{=}i) = u$, where $I \sim G(n, \mu, s)$.

if the chosen triple has a text string with more than one token and tokens with length shorter than three characters are discarded. A string is broken into tokens using white space and punctuation symbols. After all the queries of the workload have been generated, the workload is randomly permuted. Mixed workloads are generated by mixing and randomly permuting a substring workload and an exact match workload.

We present results from a subset (mean $\mu$=992, 942) of the workloads we experimented with. Table 5.1 summarizes the important workload characteristics. The cardinality and size (in bytes) of the set of distinct queries in each workload (see Table 5.1) are important in allocating memory to a selectivity estimation method: If the allocated memory is large enough to store the set of distinct queries (and the corresponding selectivities), it forms a cache that can estimate selectivity with zero error.

Figure 5.6 shows the typical distribution of query selectivities (sorted in descending order) of the three types of workloads. Note that these distributions are consistent with Zipf's Law for word frequencies [90]: the selectivity distribution over query rank in log scale (see Figure 5.6(d)) is almost linear.

Each workload will be referenced using its ID and a letter (s,e,m) denoting whether it is a substring, exact match, or mixed workload respectively. For example, workload IIm refers to the mixed workload generated with standard deviation 10,000.

**Comparisons.** CXHist will be compared with two other methods: the off-line pruned suffix tree (PST) method and the on-line compressed histogram (CH) method. The PST method constructs one pruned suffix tree per pathID for the leaf values associated with the pathID. Note that this is the same as constructing one large suffix tree where the first level nodes contain the pathIDs as alphabets. The suffix trees are pruned by increasing threshold counts to obtained the desired size. The CH method is adapted from XPathLearner [55]. CH caches the top-$k$ query feedback with the largest selectivity and aggregates the selectivity of the other query feedback into buckets according to the pathID and the first $q$ letters of the leaf value. Whenever CH uses more than *triggersize* bytes of memory, CH discards buckets in increasing order of selectivity until memory usage is less than *targetsize* bytes.

**Counting Memory.** For CXHist, 4-byte integers are used for the $sum(b)$ and $count(b)$ fields. Each pathID distribution stores a list of 8-byte $\langle pathid, count\rangle$ pairs and each $n$-gram distribution stores a list of $(n + 4)$-byte $\langle n\text{-gram}, count\rangle$ pairs. For the CH method with parameters $k$ and $q$, the memory usage consists of the memory used to store the top $k$ queries and their selectivity, and the memory used by the list of buckets. Each bucket stores a $q$-byte prefix string, and three 4-byte fields for the pathID, sum, and count. For the PST method, consult [49] for memory accounting.

**Parameters.** For both CXHist and CH the parameter *targetsize* is set to be 10% lower than *triggersize*. The parameter *triggersize* is set to be a fraction of *usize*, the size of the set of distinct queries in the workload. For CH, the parameter $k$ needs to be a small fraction of *uniq*, the number of distinct queries in the workload, otherwise all the distinct queries will be cached. Where there is no confusion which workload is being used, the values for *triggersize* and $k$ will be specified as a percentage. For our experiments, we have set $k = 0.125 \times uniq$ or equivalently $k = 12.5\%$. The notation CH($k, q, triggersize$) denotes the CH method with parameters $k$, $q$ and *triggersize*. For CXHist, the buckets are initialized with *nexp* exponential values and the gradient descent update method with a learning rate of 1 is used. The maximum selectivity is set to half the maximum number of $\langle pathid, string \rangle$ pairs, i.e., 2,522,620. The notation CXHist$(n, m, nexp, triggersize)$ denotes the CXHist method with the following parameters: $n$-gram features, $m$ buckets, *nexp* exponential representative bucket values and *triggersize* as the pruning threshold. Section 5.6.1 discusses the effect of the parameters $n$, $m$ and *nexp* on accuracy. For PST, we found the independence strategy $I_1$ of [49] to be most accurate in terms of average relative error among the different estimation strategies in [43, 49]; hence, the results we present for PST uses strategy $I_1$.

**Metrics.** We use the average relative error (*a.r.e.*) to measure the accuracy of the selectivity estimation methods. The *a.r.e.* with respect to a set $W$ of $n$ queries is defined as

$$a.r.e. = \frac{1}{n} \sum_{q \in W} \frac{|\sigma(q) - \hat{\sigma}(q)|}{\sigma(q)}, \tag{5.6.24}$$

where $\sigma(q)$ is the selectivity of query $q$ in the workload $W$ and $\hat{\sigma}(q)$ is the corresponding estimated selectivity. The state of the selectivity estimation method is assumed to remain unchanged for all the queries in workload $W$. For on-line methods, the *on-line a.r.e.* better reflects the performance of the on-line selectivity estimation method during deployment. The *on-line a.r.e.* is defined as in (5.6.24), except that the selectivity estimation method is allowed to update itself in between queries. For an off-line method, the *on-line a.r.e.* and the *a.r.e.* are the same.

## 5.6.1 Effect of Parameters

CXHist is governed by three main parameters: the number of buckets, the number of exponential values *nexp* used in the initialization of the bucket representative values (see Section 5.2.4), and the length $n$ of each $n$-gram. We investigate the effect of each of these three parameters on the performance of CXHist (all else being constant). We present the results for the mixed workload IIm.

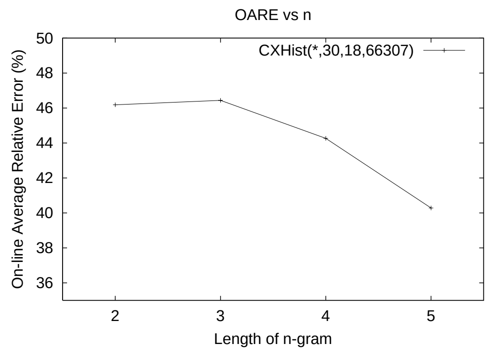


**Figure 5.7**: Performance of CXHist for different settings of $n$, the length of each $n$-gram feature.

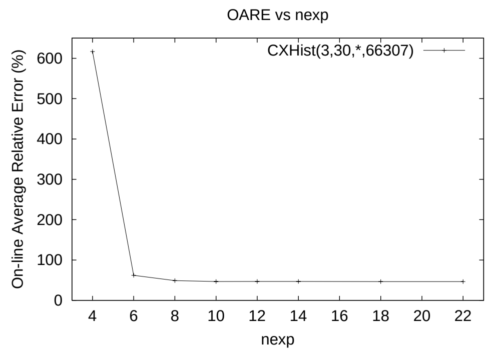


**Figure 5.8**: Performance of CXHist for different settings of *nexp*, the number of exponential representative values used for initializing the buckets.

Figure 5.7 shows the effect of varying the length of each $n$-gram. Using longer $n$-grams as features does improve the accuracy of CXHist, but note that longer $n$-

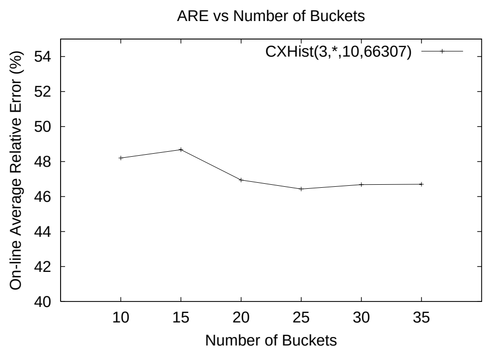


**Figure 5.9**: Performance of CXHist with different number of buckets.

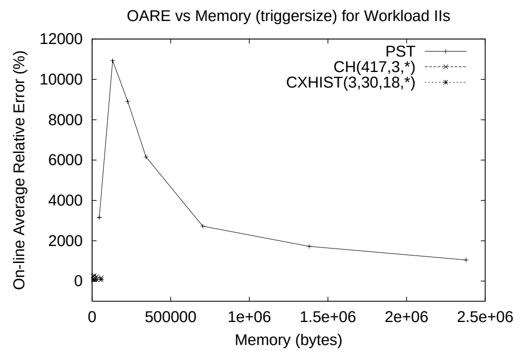
grams also require more storage space for fixed number of entries in the feature distributions.

Figure 5.8 shows the effect of varying *nexp*, the number of exponential values used to initialized the buckets. Our results show large improvement in performance when increasing *nexp* from 4 to 10, but less significant improvement beyond 10. This is not surprising, since smaller values affect the relative error more.

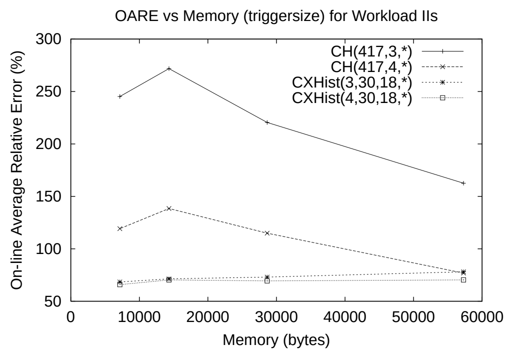
Figure 5.9 shows the effect of varying the number of buckets with the parameter *nexp* set to ten, the smallest number of buckets we experimented with. Our results show that increasing the number of buckets beyond ten does not significantly affect the relative error.

### 5.6.2 Accuracy vs Memory

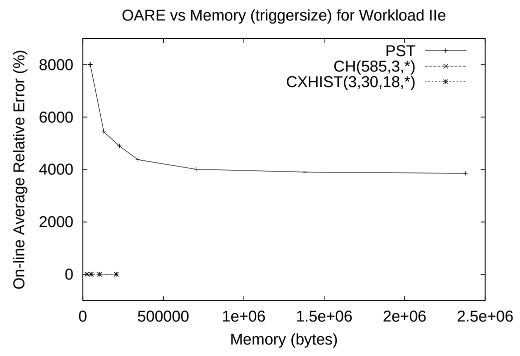
In this experiment, we vary the amount of memory allocated to each method via the *triggersize* parameter and measure the relative error performance for each workload. Note that *triggersize* is set to a fraction $(\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1)$ of the *usize* of the workload (see Table 5.1), which in turn is less than 1% the size of the XML data (ca. 125 MB). We present a set of representative results (Figure 5.10) using Workload II. The



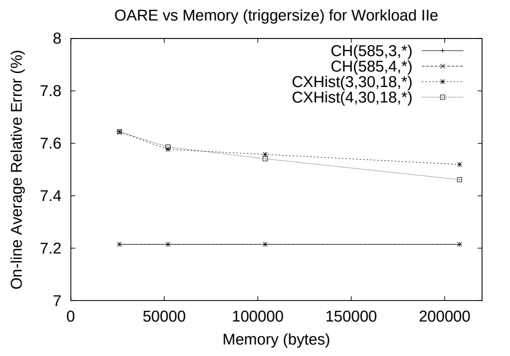
(a) IIs Overview



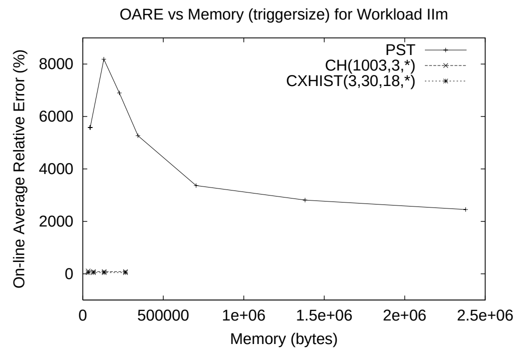
(b) IIs Detailed



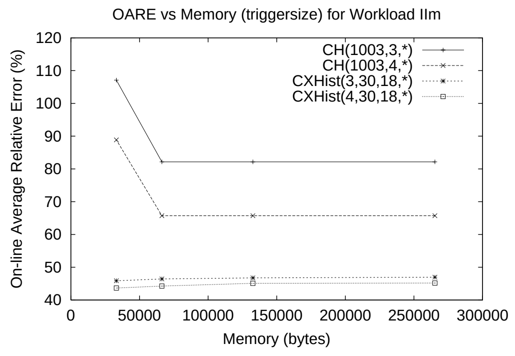
(c) IIe Overview



(d) IIe Detailed



(e) IIm Overview

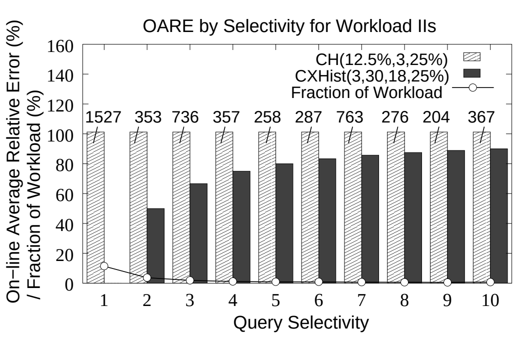


(f) IIm Detailed

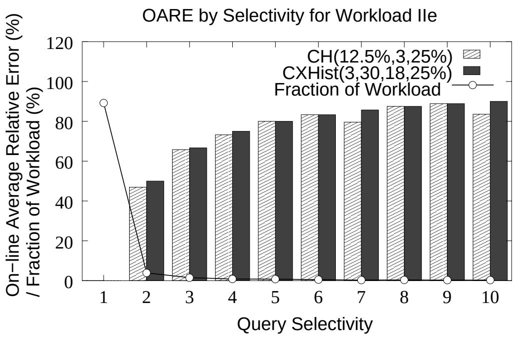
**Figure 5.10**: Performance of the PST, CH and CXHist methods over varying memory (triggersize) allocations.

overview plots (Figure 5.10(a), 5.10(c) and 5.10(e)) show that the PST method is less accurate and uses more memory compared to CH and CXHist. The detailed plots (Figure 5.10(b), 5.10(d) and 5.10(f)) compare the performance of CH and CXHist. We also show the results of the CH method for two prefix lengths (3 and 4) and the results of CXHist for two $n$-gram lengths (3 and 4), because accuracy is dependent on both memory and the prefix/$n$-gram length. For substring and mixed workloads, CXHist is more accurate than CH (Figure 5.10(b) and 5.10(f)). For exact match queries, CH is slightly (less than 1%) more accurate than CXHist (Figure 5.10(d)). The reason is that the selectivity distribution of the exact match queries is more skewed than that of the substring queries (see Figure 5.6) and hence the top-$k$ cache in CH captures almost all of the queries with large selectivities and the buckets need only capture the remaining selectivities which are uniformly small and close to one.

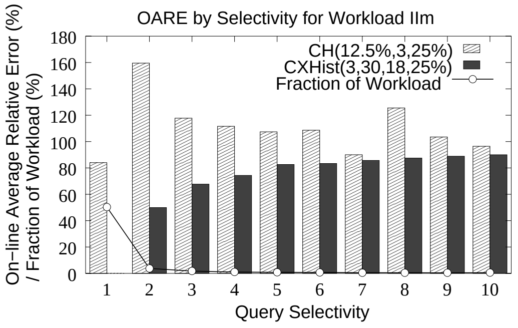
We further analyze the results for *triggersize*=25% by partitioning the selectivity estimates according to the true selectivity. Because small selectivities occur much more frequently in the data and consequently in the workload, we show the average relative error for the small selectivity partitions in Figure 5.11 and augment the figures with the fraction (normalized to a percentage) of the queries in the workload with the specified true selectivity. The CH method is not accurate for queries with small selectivities when the selectivity distribution in the workload is less skewed (for example, in the substring workload, see Figure 5.11(a)). The reason is that the selectivities of the queries not captured by the top-$k$ cache is less uniform and cannot be captured by the buckets accurately. The CXHist, on the other hand, seems to be consistently accurate for queries with small selectivities. For queries with very large selectivity, however, CH is more accurate than CXHist, because it



(a) IIs



(b) IIe



(c) IIm

**Figure 5.11**: Performance of the CH and CXHist methods over Workload IIs, IIe, and IIm partitioned by selectivity for the smallest 10 selectivities.

caches the selectivity of the $k$ queries with the largest selectivity. This is not a limitation in CXHist, because (1) queries with large selectivity are infrequent, and (2) the CXHist buckets with large representative values can be modified to store their associated queries exactly (without query modeling) to mimic the behavior of a cache for queries with large selectivity.

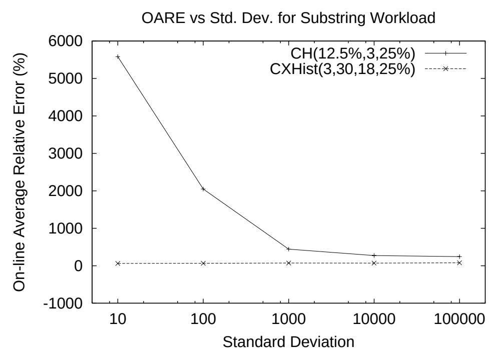
### 5.6.3 Accuracy vs Workload Skew

In this experiment, the parameter *triggersize* is fixed at $25\% \times \textit{usize}$ and the performance on workloads generated with different skew (standard deviation) is measured. Increasing standard deviation denote increasing uniformity, i.e., queries in the workload cover the XML data more uniformly. Highly skewed workloads are generally easy for on-line selectivity estimators, because the set of distinct queries are small and repetitions are frequent. Highly uniform workloads are generally hard, because repetitions seldom occur and almost every query is unique. Figure 5.12 shows the *on-line a.r.e.* vs skew plots. For substring workloads, CH is significantly less accurate than CXHist especially when the workload is highly skewed. For exact match workloads, both CH and CXHist are very accurate with CH being slightly more accurate for medium skew. For mixed workload, the superior accuracy of CXHist on substring queries dominates.

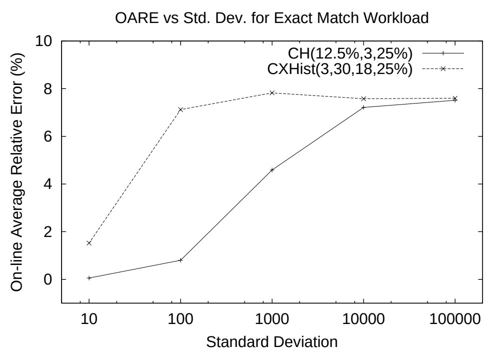
### 5.6.4 Changing Workload Characteristics

In this experiment, we investigate how changes in the workload characteristics affect accuracy of the estimates. The performance of CH and CXHist are tested on special workloads that are constructed by concatenating two workloads generated with different mean and standard deviation. We present the results for the concatenated workload IIm-IIIm. Other combinations of workloads exhibit similar behavior.

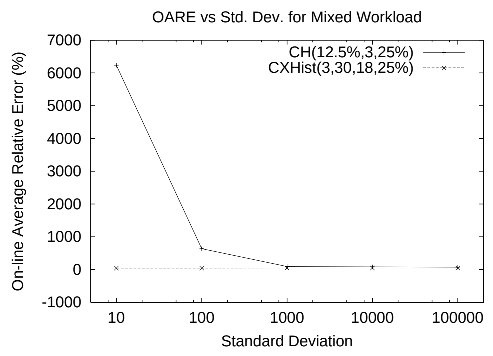
In Figure 5.13, we measure the *a.r.e.* with respect to IIm after each of the first 10,000 queries and we measure the *a.r.e.* with respect to IIIm, after each of the last 10,000 queries. CXHist remains accurate throughout the 20,000 queries, whereas the accuracy of CH degrades after the first 5000 queries.



(a) IIs



(b) IIe



(c) IIm

**Figure 5.12**: Performance of the CH and CXHist methods over workloads with different skew.

In Figure 5.14, we show the accuracy of CH and CXHist for different memory allocations on the same concatenated workload IIm-IIIm. Compared to the performance on Workload IIm alone (see Figure 5.10(f)), the performance of CH on the concatenated workload has degraded significantly (the parameter settings are the same for both figures). The performance of CXHist, on the other hand, remains stable.

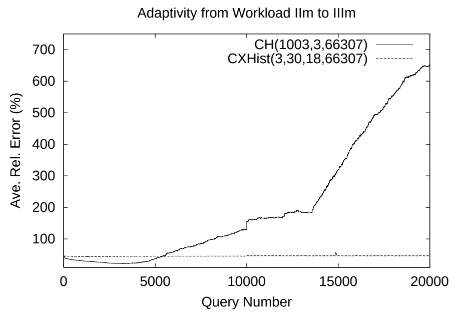


**Figure 5.13**: Average relative error for the CH and CXHist methods measured with respect to Workload IIm after each update for queries 1 to 10000 and with respect to Workload IIIm after each update for queries 10001 to 20000.

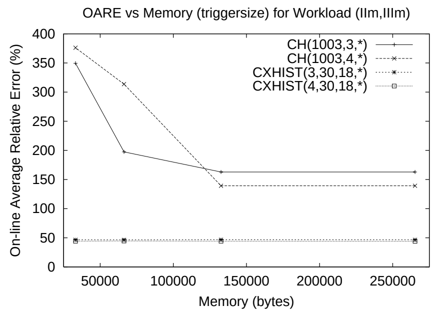


**Figure 5.14**: On-line performance of the CH and CXHist methods with different *triggersize* settings over a concatenated workload IIm-IIIm.

## 5.7 Conclusion

In this chapter, we have presented a new type of histogram called CXHist. CXHist uses a Bayesian classifier to capture the mapping between queries and their selectivity. The key idea is to partition the query selectivities into buckets and use a Bayesian classifier to approximately capture the queries that are associated with each bucket. To the best of our knowledge, CXHist is the first type of histogram based on classification. CXHist is on-line: it gathers statistics from query feedback rather than from costly data scans, and hence adapts to changes in workload characteristics and in the underlying data. We have designed an update algorithm based on the gradient descent method (delta rule) for learning a CXHist histogram from query feedback. CXHist is general and can be applied to a broad class of queries on possibly different data models by defining an appropriate query model. CXHist can be easily implemented and deployed in practice.

We have applied CXHist to an important selectivity estimation problem in XML databases — the XML string selectivity estimation problem. Previous work on XML selectivity estimation has focused mainly on the tag-labeled paths (tree structure) of the XML data; however, for most real XML data, the number of distinct string values at the leaf nodes is orders of magnitude larger than the set of distinct rooted tag paths. The real challenge lies in accurately estimating the selectivity of string predicates on the leaf values reachable via a given path. We have defined a query model for the XML string selectivity estimation problem using an $n$-gram model of the (sub)string predicate together with the stationarity assumption on the $n$-gram distribution. The algorithms for selectivity estimation and for updating the CXHist histogram have also been customized accordingly. Our experiments have shown that CXHist provides very accurate estimates for both exact match and substring queries — no previous technique have this property. Moreover, its performance is stable over changes in workload characteristics.

# Chapter 6

# Dissertation Conclusion

In this dissertation, we have proposed on-line algorithms for the inverted index update problem and three particular versions of the selectivity estimation problem.

The landmark-diff method exploits the non-uniform locality of edits in an updated document to minimize the amount updates required on an inverted index.

SASH builds and maintains a set of histograms using only query feedback information in order to support selectivity estimation of range queries on a relational database. In gathering statistics using query feedback, SASH exploits the non-uniform query characteristics of the workload to optimize the use of limited amount of given memory to store statistics that are most relevant to the query workload.

XPathLearner addresses the selectivity estimation of path expressions in XML databases. XPathLearner models the selectivity of path expressions using a Markov chain, but differs from previous work in that the model is inferred from query feedback information as opposed to constructing the model from the XML data itself. The statistics gathered by XPathLearner is therefore tuned to the query workload characteristics.

Building on the insight that modeling the query workload is more effective and space efficient than modeling the data, we develop a classification based histogram called CXHist. CXHist builds a histogram by grouping queries into buckets according to their selectivities and maintains the query-to-bucket mapping using a Bayesian classifier. We apply CXHist to the estimation of substring and exact match queries in XML data with excellent empirical results.

All the methods proposed in this dissertation avoids off-line scans of the underlying data and exploits non-uniform update or query characteristics to optimized performance or accuracy. Our experiments have shown that on-line methods can be very effective in practice and sometimes surpass the performance of off-line methods. While current information processing methods are predominantly off-line in nature, as more and more information is generated, collected and accumulated in digital media, off-line scans will become increasingly costly and infeasible. On-line methods as exemplified by those presented in this dissertation will then become increasingly important.

# Bibliography

[1] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB 2001*, pages 591–600, 2001.

[2] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD 1999*, pages 181–192, 1999.

[3] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. *Proceedings of the 7th Intl. Conf. on Extending Database Technology (EDBT '00)*, 1777:413–429, 2000.

[4] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[5] Ricardo A. Baeza-Yates and Gonzalo Navarro. Block addressing indices for approximate text retrieval. *Journal of the American Society on Information Systems*, 51(1):69–82, 2000.

[6] Andrew Barron, Jorma Rissanen, and B Yu. The minimum description length principle in coding and modeling. *IEEE Trans. Information Theory*, 44(6):2743–2760, 1998.

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1976.

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (2nd edition). *W3C Recommendation*, October 6, 2000.

[9] Brian Brewington and George Cybenko. Keeping up with the changing web. *IEEE Computer*, 33(5):52–58, May 2000.

[10] Andrei Z. Broder. On the resemblence and containment of documents. *Proceedings of Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[11] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *20th Intl. Conf. on Very Large Data Bases*, pages 192–202, 1994.

[12] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: a multidimensional workload-aware histogram. In Walid G. Aref, editor, *SIGMOD 2001*, pages 211–222. ACM Press, 2001.

[13] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery 1.0: An XML query language. *W3C Working Draft*, June 7, 2001.

[14] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM PODS*, pages 34–43, 1998.

[15] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE 2004*, 2004.

[16] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 161–172, 1994.

[17] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *ICDE 2001*, pages 595–604, 2001.

[18] Zhiyuan Chen, Flip Korn, Nick Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *PODS 2000*, pages 216–225, 2000.

[19] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. *26th Intl. Conf. on Very Large Data Bases*, 2000.

[20] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change. *ACM Transactions on Internet Technology*, 3(3):256–290, 2003.

[21] James Clark. expat—XML parser toolkit, 2000.

[22] James Clark and Steve DeRose. XPath 1.0: XML path language. *W3C Recommendation*, November 16, 1999.

[23] C. Clarke and G. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system. *Tech. Report CS-95-01, Univ. of Waterloo CS Dept.*, 1995.

[24] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with on-line update. *Tech. Report CS-94-40, Univ. of Waterloo CS Dept.*, 1994.

[25] Doug Cutting and Jan Perdersen. Optimizations for dynamic inverted index maintenance. *Proceedings of SIGIR*, pages 405–411, 1990.

[26] Amol Deshpande, Minos Garofalakis, and Michael I. Jordan. Efficient stepwise selection in decomposable models. In *Uncertainty in Artificial Intelligence: Proceedings of the 17th Conference (UAI-2001)*, pages 128–135. Morgan Kaufmann Publishers, 2001.

[27] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 199–210, 2001.

[28] Pedro Domingos and Michael J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *International Conference on Machine Learning*, pages 105–112, 1996.

[29] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1972.

[30] Malcom C. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, pages 230–241, May 1986.

[31] Christos Faloutsos and Ibrahim Kamel. Relaxing the uniformity and independence assumptions using the concept of fractal dimension. *Journal of Computer and System Sciences JCSS*, 55(2):229–240, 1997.

[32] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *Proc. of the 12th Intl. conf. on World Wide Web*, pages 669–678. ACM Press, 2003.

[33] Michael J. Fischer and Richard E. Ladner. Data structures for efficient implementation of sticky pointers in text editors. *Dept. of Computer Science, Univ. of Washington, Tech. Report 79-06-08*, June 1979.

[34] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[35] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jerome Simeon. StatiX: Making XML count. In *SIGMOD 2002*, pages 181–191, 2002.

[36] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 461–472, 2001.

[37] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating themlore data model and query language. *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[38] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[39] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB 2003*, pages 19–30, 2003.

[40] H. V. Jagadish, Hui Jin, Beng Chin Ooi, and Kian-Lee Tan. Global optimization of histograms. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 223–234, 2001.

[41] H. V. Jagadish, Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. Multi-dimensional substring selectivity estimation. In *VLDB 1999*, pages 387–398, 1999.

[42] H. V. Jagadish, Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *VLDB Journal 2000*, 9(3):214–230, 2000.

[43] H. V. Jagadish, Raymond T. Ng, and Divesh Srivastava. Substring selectivity estimation. In *PODS 1999*, pages 249–260, 1999.

[44] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. XParent: An efficient RDBMS-based XML database system. In *ICDE 2002*, pages 335–336, 2002.

[45] Michael I. Jordan, editor. *Learning in Graphical Models*. MIT Press, 1999.

[46] Marcin Kaszkiel and Justin Zobel. Passage retrieval revisited. In *Proceedings of the 20th Annual Intl. ACM SIGIR Conf.*, pages 178–185. ACM, 1997.

[47] Michael J. Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Shapire, and Linda Sellie. On the learnability of discrete distributions. *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 273–282, 1994.

[48] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[49] P. Krishnan, Jeffrey S. Vitter, and Balakrishna R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD 1996*, pages 282–293, 1996.

[50] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

[51] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.

[52] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[53] Michael Ley. DBLP XML records, 2001.

[54] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *27th Intl. Conf. on Very Large Data Bases*, pages 361–370, 2001.

[55] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey S. Vitter, and Ronald Parr. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB 2002*, 2002.

[56] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh C. Agarwal. Characterizing web document change. In *Advances in Web-Age Information Management, 2nd Intl. Conf., WAIM 2001*, pages 133–144, 2001.

[57] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. on Info. Theory*, 28:129–137, March 1982.

[58] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the Winter 1994 USENIX Conf.*, pages 23–32. USENIX, 1994.

[59] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 448–459, 1998.

[60] J. Max. Quantizing for minimum distortion. *IRE Trans. on Info. Theory*, 1960.

[61] Jason McHugh and Jennifer Widom. Query optimization for XML. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of 25th Intl. Conf. on Very Large Data Bases*, pages 315–326. Morgan Kaufmann, 1999.

[62] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *Proceedings of the 10th Intl. WWW Conf.*, 2001.

[63] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.

[64] Lawrence Page and Sergey Brin. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the 7th Intl. WWW Conf.*, pages 107–117, 1998.

[65] R. Petke and I. King. Registration procedures for URL scheme names. *IETF RFC 2717*, 1999.

[66] Neoklis Polyzotis and Minos N. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD 2002*, pages 358–369, 2002.

[67] Neoklis Polyzotis and Minos N. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB 2002*, pages 466–477, 2002.

[68] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB 1997*, pages 486–495, 1997.

[69] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD 1996*, pages 294–305, 1996.

[70] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, chapter 8, pages 318–362. MIT Press, 1986.

[71] Gerard Salton, James Allan, and Chris Buckley. Approaches to passage retrieval in full text information systems. In Robert Korfhage, Edie M. Rasmussen, and Peter Willett, editors, *Proceedings of the 16th Annual Intl. ACM-SIGIR Conf*, pages 49–58. ACM, 1993.

[72] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of 28th Intl. Conf. on Very Large Data Bases*, pages 974–985, Hong Kong, China, August 2002.

[73] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[74] Claude Shannon. Prediction and entropy of printed english. *Bell Systems Technical Journal*, 30:50–64, 1951.

[75] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. LEO – DB2 LEarning Optimizer. In *Proceedings of 27th Intl. Conf. on Very Large Data Bases*, pages 19–28. Morgan Kaufmann, 2001.

[76] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD 2002*, pages 204–215, 2002.

[77] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. *Proceedings of 1994 ACM SIGMOD Intl. Conf. of Management of Data*, pages 289–300, May 1994.

[78] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[79] Jeffrey S. Vitter. Faster methods for random sampling. *Communications of the ACM*, 27, July 1984.

[80] Jeffrey S. Vitter. An efficient I/O interface for optical disks. *ACM Trans. on Database Systems*, pages 129–162, June 1985.

[81] Jeffrey S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 193–204, 1999.

[82] Min Wang, Jeffrey S. Vitter, and Balakrishna R. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *ICDE 1997*, pages 169–180, 1997.

[83] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[84] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *EDBT 2002*, pages 590–608, 2002.

[85] XMark 100 MB standard dataset., 2002. http://www.xml-benchmark.org.

[86] Xyleme home page, 2001. http://www.xyleme.com.

[87] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Inter. Tech.*, 1(1):110–141, 2001.

[88] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *Proceedings of 2001 ACM SIGMOD Intl. Conf. of Management of Data*, pages 361–370, 2001.

[89] Aoying Zhou, Hongjun Lu, Shihui Zheng, Yuqi Liang, Long Zhang, Wenyun Ji, and Zengping Tian. VXMLR: A visual XML-relational database system. In *VLDB 2001*, pages 719–720, 2001.

[90] George Kingsley Zipf. Human behaviour and the principle of least effort: an introduction to human ecology, 1949.

[91] Justin Zobel, Alistair Moffat, Ross Wilkinson, and Ron Sacks-Davis. Efficient retrieval of partial documents. *Information Processing and Management*, 31(3):361–377, 1995.

# Biography

Lipyeow Lim was born on 15th March 1974 in Singapore. He received the B.Sc. in information systems and computer science from the National University of Singapore in 1998 under the supervision of Dr. Y. C. Tay, and the M.Sc. in information systems and computer science from the National University of Singapore in 1999 under the supervision of Dr. Philip M. Long.

Lipyeow Lim was the first to implement the Internet Engineering Task Force (IETF) RFC 2290 *MIPv4 Option for PPP IPCP* in Linux's point-to-point protocol (PPP) source code in 1998. In 1999, Lipyeow worked on image compression research and wrote his Master's thesis *A Theoretical Look at Pixel Ordering*. During his doctoral studies, Lipyeow worked on text indexing and database optimization research under the supervision of Dr. Jeffrey S. Vitter, and has published the following articles: *Characterizing Web Document Change (2001), Wavelet-Based Cost Estimation for Spatial Queries (2001), XPathLearner: An On-Line, Self-Tuning Markov Histogram for XML Path Selectivity Estimation (2002), Dynamic Maintenance of Web Indexes Using Landmarks (2003)*, and *SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads (2003).*

In 2003 Lipyeow Lim was awarded the IBM Ph.D. fellowship.