COST ESTIMATION TECHNIQUES FOR DATABASE SYSTEMS

By

Ashraf Aboulnaga

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

2002

© Copyright by Ashraf Aboulnaga 2002

All Rights Reserved

Abstract

This dissertation is about developing advanced selectivity and cost estimation techniques for query optimization in database systems. It addresses the following three issues related to current trends in database research: estimating the cost of spatial selections, building histograms without looking at data, and estimating the selectivity of XML path expressions.

The first part of this dissertation deals with estimating the cost of spatial selections, or window queries, where the query windows and the data objects are general polygons. Previously proposed cost estimation techniques only handle rectangular query windows over rectangular data objects, thus ignoring the significant cost of exact geometry comparison (the refinement step in a "filter and refine" query processing strategy). The cost of the exact geometry comparison depends on the selectivity of the filtering step and the average number of vertices in the candidate objects identified by this step. We develop a cost model for spatial selections that takes these parameters into account. We also introduce a new type of histogram for spatial data that captures the size, location, and number of vertices of the spatial objects. Capturing these attributes makes this type of histogram useful for accurate cost estimation using our cost model, as we experimentally demonstrate.

The second part of the dissertation introduces self-tuning histograms. While similar in structure to traditional histograms, self-tuning histograms are built not by examining the data or a sample thereof, but by using feedback from the query execution engine about the selectivities of range selections on the histogram attributes to progressively refine the histogram. Since self-tuning histograms have a low up-front cost and the cost of building them is independent of the data size, they are an attractive alternative to traditional histograms, especially multi-dimensional histograms. The low cost of self-tuning histograms can help a self-tuning self-administering database system experiment with building many different histograms on many different combinations of data columns. This is useful since the system cannot rely on a database administrator to decide which histograms to build.

The third part of this dissertation is about estimating the selectivity of XML path expressions. XML is increasingly becoming the data representation format of choice for data on the Internet, which makes querying XML data very useful. Queries over XML data use path expressions to navigate through the structure of the data, and optimizing these queries requires estimating the selectivity of these path expressions. We propose two techniques for estimating the selectivity of simple XML path expressions over complex large-scale XML data: path trees and Markov tables. Both techniques work by summarizing the structure of the XML data in a small amount of memory and using this summary for selectivity estimation. We experimentally demonstrate the accuracy of our proposed techniques over real and synthetic XML data, and we explore the different situations that would favor one technique over the other.

Acknowledgements

I was very fortunate to have Jeff Naughton as my advisor. Jeff has always given me a lot of freedom to do what I want in my graduate career, but he knew when to step in to provide the required focus and direction. He was always very patient with me, and I knew that I could always count on him for any support I needed. I looked forward to my meetings with Jeff because I knew they would be full of technical insights, sound advice, and a healthy dose of wit!

I would also like to thank David DeWitt. I have found David to be a very caring and considerate person, in addition to being one of the technical leaders of the database field. It was a pleasure working with David. I learned a lot from him, and he has given me a lot of support.

I would like to thank Raghu Ramakrishnan for all his contributions to the database group at the University of Wisconsin, and for his helpful comments and suggestions. Thanks are also due to Professors Remzi Arpaci-Dusseau and Mikko Lipasti for being on my dissertation committee.

Part of the research for this dissertation was done during two summers that I spent as an intern in the Database Group at Microsoft Research. I would like to thank Surajit Chaudhuri for being my mentor these two summers. I learned a lot from Surajit on how to do high-quality, relevant research and how to have a successful career in industrial research. Thanks also go to David Lomet for putting together and managing such a great database research group. Vivek Narasayya was a good colleague and friend, and he helped me in continuing to conduct experiments at Microsoft after returning to Wisconsin. Part of my financial support during graduate school was generously provided by Microsoft through a Microsoft Graduate Fellowship.

This dissertation could not have been completed without a lot of help from many people. Jignesh Patel and Kristin Tufte helped me with Paradise. Nancy Hall patiently answered my questions about SHORE. Raghu Ramakrishnan provided me with real point of sale data for my experiments on self-tuning histograms, and Kevin Beyer and Venkatesh Ganti helped me use this data. Zhiyuan Chen and Divesh Srivastava generously provided me with their code

for estimating the selectivity of path expressions and with real XML data. Chun Zhang and I together wrote the XML data generator which I used for some of my experiments. My work on estimating the selectivity of XML path expressions could not have been completed without the assistance of my good friend Alaa Alameldeen. Alaa went well above and beyond the call of duty in helping me co-author the paper presenting that work.

I would also like to thank Anastassia Ailamaki, Johannes Gehrke, Johan Larson, Navin Kabra, Biswadeep Nag, Leonidas Galanis, Stratis Viglas, Josef Burger, and the entire Niagara team.

If I were to name one person who deserves the most thanks, it would be my wife, Naglaa Wahby. I thank her for all her patience and support. I would also like to thank my parents, Ismail Aboulnaga and Hedaya Karam, for their support and encouragement. Finally, I would like to thank my daughter, Aaya, and my son, Ibraheem, for helping me keep everything in perspective.

Contents

Abstract i

Acknowledgements iii

Contents v

List of Tables ix

List of Figures x

1 Introduction 1

2 Accurate Estimation of the Cost of Spatial Selections 5

2.1 Introduction 5

2.2 Related Work 7

2.3 A Cost Model for Window Queries 8

2.3.1 Filtering 9

2.3.2 Refinement 11

2.3.3 Notes 11

2.4 SQ-histograms 12

2.4.1 Partitioning the Data into Buckets 13

2.4.2 Handling Objects With Varying Complexities 17

2.4.3 Assuming Uniformity Within a Bucket 18

2.4.4 Estimation Using SQ-histograms 19

2.4.5 Comparison with MinSkew Partitioning 20

2.5 Experimental Evaluation 22

2.5.1 Generating Synthetic Polygons 22

2.5.2 Experimental Setup 23

2.5.3 Importance of the Cost of Refinement 25
2.5.4 Estimation Accuracy Using SQ-histograms 27
2.5.5 Accuracy of the Window Query Cost Model 28
2.5.6 Effect of Histogram Construction Parameters 31
2.5.7 Using Sampling for Selectivity Estimation 35
2.6 Conclusions 37

3 Self-tuning Histograms: Building Histograms Without Looking at Data 38
3.1 Introduction 38
3.1.1 Self-tuning Histograms 39
3.1.2 Applications of Self-tuning Histograms 41
3.2 Related Work 43
3.3 One-dimensional ST-histograms 44
3.3.1 Initial Histogram 45
3.3.2 Refining Bucket Frequencies 45
3.3.3 Restructuring 48
3.4 Multi-dimensional ST-histograms 52
3.4.1 Initial Histogram 54
3.4.2 Refining Bucket Frequencies 55
3.4.3 Restructuring 55
3.5 Experimental Evaluation 57
3.5.1 Experimental Setup 58
3.5.2 Accuracy of One-dimensional ST-histograms 62
3.5.3 Accuracy of Multi-Dimensional ST-histograms 63
3.5.4 Effect of Locality of Reference in the Query Workload 66
3.5.5 Adapting to Database Updates 68
3.5.6 Refinement Parameters 69
3.5.7 Effect of Available Memory 70
3.5.8 On-line Refinement and Convergence 73

3.5.9 Accuracy of ST-histograms on Real Data 75
3.6 Conclusions 79

4 Estimating the Selectivity of XML Path Expressions for Internet-scale Applications 80
4.1 Introduction 80
4.2 Related Work 83
4.3 Path Trees 85
4.3.1 Sibling-$*$ 86
4.3.2 Selectivity Estimation 89
4.3.3 Level-$*$ 90
4.3.4 Global-$*$ 91
4.3.5 No-$*$ 92
4.4 Markov Tables 92
4.4.1 Suffix-$*$ 94
4.4.2 Global-$*$ 95
4.4.3 No-$*$ 96
4.5 Experimental Evaluation 96
4.5.1 Estimation Using Pruned Suffix Trees 96
4.5.2 Data Sets 97
4.5.3 Query Workloads 98
4.5.4 Performance Evaluation Method 99
4.5.5 Summarizing Path Trees 100
4.5.6 Summarizing Markov Tables 101
4.5.7 Estimation Accuracy 103
4.6 Conclusions 106

5 Conclusions 108

Bibliography 110

A Generating Synthetic Complex-structured XML Data **117**

A.1 Generating the Tree Structure of the Data 117

A.2 Tag Names – Recursion and Repetition 118

A.3 Element Frequency Distribution 120

List of Tables

1 Parameters of the window query cost model and how they are obtained 9

2 Cost of a window query with MBR area equal to 1% of space 26

3 Calibration constants for the window query cost model 28

4 ST-histograms initialized using different one-dimensional histograms 66

5 Accuracy of ST-histograms with and without periodic restructuring 69

6 Average result sizes of the XML query workloads 98

List of Figures

1 A quadtree partitioning and the assignment of polygons to quadtree nodes . . . 15
2 Algorithm for building SQ-histograms 17
3 The x dimension of a query overlapping a bucket 18
4 The difficulty of estimating actual selectivities 20
5 A synthetically generated polygon 22
6 The synthetic polygon data set used in our experiments 24
7 Query selectivity (Sequoia data set) 25
8 Error in estimating s_{MBR} for the Sequoia data set 28
9 Error in estimating v_{cand} for the Sequoia data set 29
10 Error in estimating s_{MBR} for the synthetic data set 29
11 Error in estimating v_{cand} for the synthetic data set 30
12 Execution times with a cold buffer pool for the Sequoia data set 31
13 Execution times with a warm buffer pool for the Sequoia data set 32
14 Execution times with a cold buffer pool for the synthetic data set 32
15 Execution times with a warm buffer pool for the synthetic data set 33
16 Effect of available memory on estimating s_{MBR} for the Sequoia data set 34
17 Effect of the number of levels in the initial quadtree on estimating s_{MBR} 34
18 Effect of the measure of variation on estimating s_{MBR} 35
19 Selectivity estimation using sampling for the Sequoia data set 36
20 On-line and off-line refinement of ST-histograms 41
21 Algorithm for updating bucket frequencies in one-dimensional ST-histograms . 46
22 Algorithm for restructuring one-dimensional ST-histograms 50
23 Example of ST-histogram restructuring 52
24 A two-dimensional ST-histogram and a range selection using it 54
25 Restructuring the vertical dimension of a two-dimensional ST-histogram 57
26 Accuracy of one-dimensional ST-histograms 62

27 Accuracy of two-dimensional ST-histograms starting with MaxDiff(V,A) 64
28 Accuracy of three-dimensional ST-histograms starting with MaxDiff(V,A) . . . 64
29 Accuracy of ST-histograms for workloads with locality of reference 67
30 Accuracy of ST-histograms in adapting to database updates 68
31 Effect of α on the accuracy of ST-histograms 70
32 Effect of available memory on the accuracy of one-dimensional ST-histograms 71
33 Effect of available memory on the accuracy of two-dimensional ST-histograms 71
34 Effect of available memory on the accuracy of three-dimensional ST-histograms 72
35 On-line refinement of one-dimensional ST-histograms 73
36 On-line refinement of two-dimensional ST-histograms 74
37 On-line refinement of three-dimensional ST-histograms 74
38 Two-dimensional ST-histograms starting with MaxDiff(V,A) – Point of sale data 76
39 Three-dimensional ST-histograms starting with MaxDiff(V,A) – Point of sale data 76
40 Two-dimensional ST-histograms starting with MaxDiff(V,A) – Census data . . 78
41 Three-dimensional ST-histograms starting with MaxDiff(V,A) – Census data . . 78
42 An XML document and its path tree 85
43 An example path tree 88
44 The sibling-$*$ summarization of the path tree in Figure 43 88
45 The level-$*$ summarization of the path tree in Figure 43 90
46 The global-$*$ summarization of the path tree in Figure 43 91
47 A path tree and the corresponding Markov table for $m = 2$ 93
48 Path tree summarization for the synthetic data set and random paths workload . 100
49 Path tree summarization for the synthetic data set and random tags workload . . 101
50 Markov table summarization for the synthetic data set and random paths workload 102
51 Markov table summarization for the synthetic data set and random tags workload 103
52 Estimation accuracy for the synthetic data set and random paths workload . . . 104
53 Estimation accuracy for the synthetic data set and random tags workload 104
54 Estimation accuracy for the DBLP data set and random paths workload 105
55 Estimation accuracy for the DBLP data set and random tags workload 105

56 Markovian memory in the generated path tree due to repeated tag names 119

57 Different ways of assigning frequencies to the nodes of the generated path tree . 121

Chapter 1

Introduction

A key feature of database systems is that they allow users to query the data that they store using a declarative query language, such as SQL [DD93] for relational data or XQuery [CFR+01] for XML data. These declarative languages specify *what* the result of the query should be, not *how* to compute this result. A database system can compute the result of a query in many different ways. Each one of these ways is called a *query execution plan*. One of the components of any database system is the *query optimizer* [JK84], which chooses the query execution plan with the *least estimated cost* for any given query.

The cost of executing the different query execution plans for a particular query can vary across several orders of magnitude. Thus, it is very important to have an effective query optimizer that can identify optimal – or close-to-optimal – plans and avoid really bad plans. This is not a trivial task because the cost of a plan can vary widely depending on factors such as the data distribution of the underlying data, the physical organization of the data on disk, and the sorting order of the data. The best plan for one case can be a very bad plan for another case.

To choose the best plan for executing a query, a query optimizer enumerates a number of candidate plans according to some strategy for exploring the plan space, such as dynamic programming [SAC+79]. The query optimizer uses a cost model to estimate the cost of each of the enumerated plans, and it chooses the plan with the least estimated cost. The quality of the plan chosen by the optimizer clearly depends on the accuracy of the optimizer cost model. An inaccurate cost model can make the optimizer choose a very bad plan if it incorrectly estimates a low cost for this plan.

Estimating the cost of query operators requires estimating the *selectivity* of these operators. The selectivity of an operator is the number of results it produces. Selectivity estimation is important because two of the factors affecting the cost of an operator are the size of its output

and the sizes of its inputs. The size of the output of an operator is determined by its own selectivity. The sizes of the inputs of an operator are determined by the selectivities of operators that are below it in the query execution plan.

This dissertation focuses on selectivity and cost estimation for query optimization. Current advances in database research present new challenges to query optimizers and require the use of advanced selectivity and cost estimation techniques. This dissertation is about developing such techniques. It addresses three topics: estimating the cost of spatial selections, building histograms without looking at the data, and estimating the selectivity of XML path expressions.

The first part of this dissertation deals with estimating the cost of spatial selections. There is a trend for database systems to support new data types through the use of *object-relational* techniques [CDN+97]. An important class of data to be supported by object-relational database systems is spatial data [P+97]. A common type of spatial data is polygon data, and a common type of query to ask over such data is "find all data polygons that overlap this query polygon." This is a *spatial selection*, or *window query*, which may be part of a more complex user query. The query polygon is known as the *query window*.

Window queries are processed using a two step strategy. The first step, known as the *filtering* step, identifies data objects whose minimum bounding rectangles overlap the minimum bounding rectangle of the query window. The second step, known as the *refinement* step, tests the exact geometry representation of these objects to determine which objects actually overlap the query window [Ore86].

Several cost models have been proposed for window queries, but all of them focus on I/O cost and only handle rectangular query windows over rectangular data objects. In effect, these cost models estimate the I/O cost of the filtering step. In the first part of this dissertation, we demonstrate that the CPU and I/O cost of the refinement step is often the major component in the cost of processing a window query, so it should not be ignored. We provide a cost model for window queries that takes into account the CPU and I/O costs of both the filtering and the refinement steps. Estimating the cost of the refinement step requires estimating the selectivity of the filtering step and the number of vertices in the candidate data objects identified by this step. We develop a new type of histogram for spatial data that helps in estimating these parameters.

These histograms, which we call *structural quadtree histograms*, capture the size, location, and number of vertices of the spatial objects. We experimentally verify the accuracy of our proposed techniques on real and synthetic spatial data in the context of the Paradise geospatial object-relational database system [P+97]. We also investigate sampling-based estimation approaches. Sampling can yield better selectivity estimates than histograms for polygon data, but at the high cost of performing exact geometry comparisons for all the sampled objects.

The second part of this dissertation introduces *self-tuning histograms*. Database systems use histograms that represent the distribution of the stored data for selectivity estimation. Histograms provide accurate selectivity estimates, but there is a cost associated with scanning and processing the data to build them. Self-tuning histograms are aimed at reducing this cost, especially for multi-dimensional histograms. Self-tuning histograms do not examine the data to determine how it is distributed, but rather they learn the data distribution using feedback from the query execution engine.

There is a current trend in database research toward building self-tuning self-administering database systems [CN97], and self-tuning histograms can be a useful tool for such systems. The low cost of self-tuning histograms can help a self-tuning self-administering database system experiment with building many different histograms on many different combinations of data columns. This is useful since the system cannot rely on an "expert" database administrator to tell it which histograms to build.

A self-tuning histogram on a set of data columns is initialized with whatever information is available about the data distribution of these columns. If there are traditional one-dimensional histograms on the columns, they are used to construct the initial self-tuning histogram under the assumption that the columns are independently distributed. If there are no such one-dimensional histograms, the initial self-tuning histogram is constructed assuming that the columns are uniformly distributed and independent. The self-tuning histogram is then refined based on information obtained from observing the execution of queries that use the histogram for selectivity estimation. In the second part of this dissertation, we present the details of self-tuning histograms, as well as an experimental evaluation of their accuracy and convergence using real and synthetic multi-dimensional data sets.

The third part of this dissertation is about estimating the selectivity of XML [BPS98] path expressions for Internet-scale applications. XML is increasingly becoming the data representation format of choice for data on the Internet. This enables novel applications that pose queries over "all the XML data on the Internet." For example, the Niagara query system [NDM+01] crawls the Internet searching for XML documents and allows users to pose queries against all the XML documents that it knows about.

Queries over XML data use path expressions to navigate through the structure of the data [CD99], and optimizing these queries requires estimating the selectivity of these path expressions. This requires using database statistics that contain information about the structure of the XML data. We propose two techniques for estimating the selectivity of simple XML path expressions over complex large-scale XML data as would be handled by Internet-scale applications: path trees and Markov tables. Both techniques work by summarizing the structure of the XML data in a small amount of memory and using this summary for selectivity estimation. We experimentally demonstrate the accuracy of our proposed techniques over real and synthetic XML data and explore the different situations that would favor one technique over the other.

The rest of this dissertation is organized as follows. Chapter 2 presents our techniques for estimating the cost of spatial selections. Chapter 3 describes self-tuning histograms. Chapter 4 describes our work on estimating the selectivity of XML path expressions. Chapter 5 contains concluding remarks. Since the three topics discussed in Chapters 2–4 are fairly independent, each chapter is fully self-contained, presenting the related work, experimental evaluation, and conclusions for the topic it discusses. Chapter 5 presents the overall conclusions of the dissertation.

Chapter 2

Accurate Estimation of the Cost of Spatial Selections

2.1 Introduction

For a database system to fully support spatial data, it must be able to optimize queries involving this data. This requires the query optimizer to estimate the selectivity and cost of spatial operations. We focus on estimating the selectivity and cost of spatial selections, also known as *window queries*. In a window query, a region called the *query window* is specified, and the query retrieves all objects in the data set that overlap this region. Our focus is estimating the selectivity and cost of window queries where the query windows and the underlying data objects are general polygons.

Database systems process window queries and other spatial operations using a two step *filter and refine* strategy [Ore86]. The filtering step identifies a set of *candidate objects* whose *minimum bounding rectangles* (MBRs) overlap the MBR of the query window. This set of candidates is a conservative approximation (i.e., a superset) of the result. The filtering step may use an R-tree index if one exists [Gut84]. The refinement step tests the exact geometry of the candidate objects identified by the filtering step to determine the set of objects that actually overlap the polygonal query window.

Several cost models for window queries have been proposed in the literature [FK94, BF95, TS96, APR99]. All these cost models assume that the query windows and the data objects are rectangles. In effect, they estimate the selectivity and cost of the filtering step and ignore the refinement step.

Ignoring the refinement step makes these cost models inaccurate for two reasons. First, the estimated selectivity of the filtering step, no matter how accurate, is only an upper bound that may significantly over-estimate the actual selectivity of the query. Second, the refinement step incurs significant costs that cannot be ignored. The refinement step involves fetching the exact geometry representation of all the candidate objects, thus incurring an I/O cost. It also involves testing these candidate objects to determine the ones that actually overlap the query window using computational geometry algorithms that have a high CPU cost, typically $O(n \log n)$, where n is the total number of vertices in the polygons being tested. An important property of the costs incurred by the refinement step is that they depend not only on the selectivity of the query, but also on the number of vertices, or *complexity*, of the query window and data objects. It has been shown that, for spatial joins, the CPU cost of the refinement step dominates the query execution cost [BKSS94, PD96].

The cost of the refinement step should, therefore, not be ignored when estimating the cost of window queries, especially since typical applications of spatial databases (e.g., GIS) involve objects with high complexities (i.e., a large number of vertices). Our experiments show that the refinement step can take over an order of magnitude longer than the filtering step (Section 2.5.3). Ignoring the cost of refinement is clearly a mistake.

We introduce a new type of histogram for polygon data that captures all properties of a data distribution required for estimating the cost of both the filtering and the refinement steps of spatial operations. We present a simple cost model that uses our histograms to estimate the cost of window queries where the query windows and data objects are general polygons. We also investigate the use of sampling for estimating the selectivity and cost of window queries.

The rest of this chapter is organized as follows. In Section 2.2, we present an overview of related work. In Section 2.3, we present our cost model for window queries. Section 2.4 introduces our novel approach to building histograms for spatial data. These histograms are used to estimate the parameters required by the cost model. Section 2.5 presents an experimental evaluation of the proposed techniques. Section 2.6 contains concluding remarks.

2.2 Related Work

Several techniques have been proposed for estimating the selectivity and cost of operations on traditional data types such as integers or strings. Techniques based on using histograms to approximate data distributions are widely used by current database systems [PIHS96]. Histograms for multi-dimensional data have also been proposed in the literature [MD88, PI97].

Another approach to selectivity estimation is sampling, which provides guaranteed error bounds at the cost of taking a sample of the data at query optimization time. A sequential sampling algorithm that guarantees an upper bound for both the estimation error and the number of samples taken is presented in [LNS90].

Traditional multi-dimensional histograms can be used for point data, but not for polygons or other spatial data types. Polygons have an *extent* in space, whereas these histograms only capture the *location* of the data. On the other hand, the same sampling approaches used for traditional data can be used for spatial data. However, dealing with spatial data increases the cost of sampling.

A cost model for window queries in R-trees is developed in [KF93], and independently in [PSTW93]. This cost model assumes that the data consists of uniformly distributed rectangles and estimates the number of disk I/Os needed to answer a given rectangular window query. The latter paper also suggests a framework for studying the cost of window queries based on a knowledge of the query and data distributions.

In [FK94] and [BF95], the authors suggest using the concept that all data sets are self-similar to a certain degree to represent the distribution of spatial data. The degree of self-similarity of a data set is represented by its *fractal dimension*. These papers present models developed based on this concept for estimating the selectivity of window queries over point data and the cost of these queries in R-trees. Using the fractal dimension is a significant departure from the uniformity assumption typically made by prior works.

Another cost model for window queries in R-trees is proposed in [TS96]. This cost model is based on the *density* of the dataset, which is the average number of data objects per point of the space. The authors propose using the density at several representative points of the space to

capture non-uniformity in the data distribution.

Acharya, Poosala, and Ramaswamy [APR99] study different partitionings that could be used to build spatial histograms, and introduce a new partitioning scheme based on the novel notion of *spatial skew*. This work is closely related to ours, and a detailed comparison is given in Section 2.4.5.

As mentioned earlier, all these works assume that the query windows are rectangles and that the data objects are points or rectangles, thus ignoring the refinement step. Furthermore, with the exception of [APR99], these works do not present general solutions for accurately approximating spatial data distributions.

A different approach to estimating the selectivity of spatial selections is given in [Aoki99]. This work assumes that an R-tree index for the spatial attribute exists, and proposes that each non-leaf entry store the number of tuples that it represents. Selectivity is estimated using a tree traversal augmented by sampling when necessary. Like other approaches, this approach ignores the refinement step. Furthermore, it requires I/O for selectivity estimation, making it a high-cost approach. The main appeal of this approach is that it works not only for spatial data, but also for any data type indexed by a generalized search tree (GiST) index [HNP95].

Subsequent to publishing the work presented in this chapter in [AN00], Wang, Vitter, Lim, and Padmanabhan extended it by proposing a technique that uses wavelet-based spatial histograms to estimate the parameters of our cost model [WVLP01]. These wavelet-based histograms are shown to be more accurate than our proposed histograms on some synthetic data sets, particularly data sets consisting of objects that have a high aspect ratio (long and skinny objects).

2.3 A Cost Model for Window Queries

In this section, we present a cost model for estimating the I/O and CPU costs of both the filtering and the refinement steps of a window query. The model assumes that the query window and the data objects are general polygons. The cost of the filtering step depends on whether a sequential scan or an R-tree index [Gut84] is used as the access method, and the cost of the refinement step

Parameter	Description	Source
N	Number of pages in the relation	Catalog
T	Number of tuples in the relation	
m	Average number of entries per R-tree node	
h	Height of the R-tree	
c_{seqio}	Per page cost of sequential read	Calibration
c_{randio}	Per page cost of random read	
c_{polyio}	Per object cost of reading a polygon	
c_{vertio}	Per vertex cost of reading a polygon	
$c_{MBRtest}$	CPU cost of testing rectangle overlap	
$c_{polytest}$	Cost coefficient for testing polygon overlap	
v_q	Number of vertices in the query polygon	Given
s_{MBR}	MBR selectivity	Estimated
v_{cand}	Average number of vertices per candidate polygon	

Table 1: Parameters of the window query cost model and how they are obtained

is assumed to be independent of the access method used for filtering. The parameters used by the cost model are given in Table 1.

2.3.1 Filtering

Sequential Scan

If the input relation is accessed by a sequential scan, the I/O cost of the filtering step, CF_{IOseq}, is given by:

$$CF_{IOseq} = N * c_{seqio}$$

where N is the number of pages in the relation, and c_{seqio} is the per page cost of a sequential read.

During the sequential scan, the MBRs of all tuples of the relation are tested to determine whether they overlap the query MBR. The CPU cost of this test, CF_{CPUseq}, is given by:

$$CF_{CPUseq} = T * c_{MBRtest}$$

where T is the number of tuples in the relation, and $c_{MBRtest}$ is the CPU cost of testing whether two rectangles overlap.

R-tree Index

To estimate the cost of the filtering step if an R-tree index is used as the access method, we assume that the R-tree is "good", in the sense that retrieving the data objects that overlap the query window MBR requires the minimum number of disk I/Os and rectangle overlap tests. We also assume that the buffer pool is managed in such a way that each required R-tree node is read from disk exactly once.

The filtering step retrieves $s_{MBR} * T$ tuples, where s_{MBR} is the *MBR selectivity* of the query, defined as the fraction of tuples in the relation identified as candidates by the filtering step. This is the fraction of tuples in the relation whose MBRs overlap the query window MBR. The assumption that the R-tree is "good" implies that the tuples retrieved by the filtering step will be in the minimum number of R-tree leaf nodes. This number can be estimated as $s_{MBR} * T/m$, where m is the average number of entries per R-tree node. Extending this argument, we can estimate the number of nodes that have to be read from the level above the leaves by $s_{MBR} * T/m^2$, from the next level up by $s_{MBR} * T/m^3$, and so on until we reach the root level, at which only 1 node has to be read. Thus, the I/O cost of this step, $CF_{IOrtree}$, is given by:

$$\begin{aligned} CF_{IOrtree} &= \left(\frac{s_{MBR} * T}{m} + \frac{s_{MBR} * T}{m^2} + \cdots + \frac{s_{MBR} * T}{m^{h-1}} + 1\right) * c_{randio} \\ &= \left[\left(\frac{1}{m-1}\right)\left(1 - \frac{1}{m^{h-1}}\right) * s_{MBR} * T + 1\right] * c_{randio} \end{aligned}$$

where h is the height of the R-tree (number of levels including the root node), and c_{randio} is the cost per page of a random read. We assume that we will not encounter any "false hits" while searching the R-tree. This means that we do not have to read any nodes beyond those accounted for in the above formula. Notice that, for typical values of m, the number of internal R-tree nodes read will be very small.

The filtering step has to test all the entries in each R-tree node read from the disk for overlap with the query window MBR. Since each node contains, on average, m entries, the CPU cost of this step can be estimated by:

$$CF_{CPUrtree} = \left[\left(\frac{1}{m-1}\right)\left(1 - \frac{1}{m^{h-1}}\right) * s_{MBR} * T + 1\right] * m * c_{MBRtest}$$

2.3.2 Refinement

The refinement step has to retrieve the exact representation of all the candidate polygons identified by the filtering step. We estimate the I/O cost of reading a polygon by two components. The first component is a fixed cost independent of the size of the polygon, which we call c_{polyio}. The second component is a variable cost that depends on the number of vertices of the polygon. The number of vertices of a polygon is referred to as its *complexity*. We estimate this component of the cost by $v_{cand} * c_{vertio}$, where c_{vertio} is the per vertex I/O cost of reading a polygon and v_{cand} is the average number of vertices in the candidate polygons. Thus, the I/O cost of the refinement step, CR_{IO}, can be estimated by:

$$CR_{IO} = s_{MBR} * T * (c_{polyio} + v_{cand} * c_{vertio})$$

The CPU cost of the refinement step depends on the algorithm used for testing overlap. Detecting if two general polygons overlap can be done in $O(n \log n)$ using a plane sweep algorithm, where n is the total number of vertices in both polygons [dBvKOS97]. We therefore use the following formula to estimate the CPU cost of the refinement step:

$$CR_{CPU} = s_{MBR} * T * (v_q + v_{cand}) \log (v_q + v_{cand}) * c_{polytest}$$

where v_q is the number of vertices in the query polygon, and $c_{polytest}$ is a proportionality constant. Database systems may use algorithms other than plane sweep to test for overlap between polygons. However, since the complexity of almost all overlap testing algorithms is a function of the number of vertices of the polygons, variations of the above formula can typically be used. Each system should replace the $n \log n$ term in the formula with the complexity of the overlap testing algorithm it uses.

2.3.3 Notes

- Estimating the cost of a window query does not require knowing its actual selectivity. It only requires knowing the selectivity of the filtering step, the MBR selectivity. All candidate polygons identified by the filtering step have to be tested in the refinement step, whether or not they appear in the final result of the query.

- The parameters required by the cost model are obtained from several sources (Table 1). N, T, m, and h should be available in the system catalogs. c_{seqio}, c_{randio}, c_{polyio}, c_{vertio}, $c_{MBRtest}$, and $c_{polytest}$ are calibration constants that must be provided by the system implementer at system development or installation time. These constants depend on the specific database system and its run-time environment. v_q is known at query optimization time. Finally, s_{MBR} and v_{cand} must be estimated. The next section introduces histograms that can be used to accurately estimate these two parameters.

- The cost model we have presented here is, like all estimation models used in query optimization, a simplification of reality. For example, it does not capture such things as buffer pool management, or the degree to which the system is able to overlap the CPU time of the refinement step on some polygons with the I/O time to fetch others. Certainly, variants of the equations we have given are possible, and different variants may be more accurate for different systems. Nevertheless, the key point remains that any reasonable model must involve the parameters s_{MBR} and v_{cand}.

2.4 SQ-histograms

In this section, we introduce a novel approach to building histograms that represent the distribution of polygon data. These histograms capture information not only about the location of the data polygons, but also about their size and complexity. We call these histograms *SQ-histograms*, for *structural quadtree histograms*, because they capture the structure of the data polygons and are based on a quadtree partitioning of the space.

We use SQ-histograms to estimate the MBR selectivity of window queries, s_{MBR}, and the average number of vertices in candidate polygons identified by the filtering step, v_{cand}. SQ-histograms can also be used in any application that requires an approximate representation of the spatial data distribution.

SQ-histograms partition the data objects into possibly overlapping rectangular buckets. The partitioning is based on the object MBRs, and tries to group similar objects together in the same

buckets, with each object being assigned to one histogram bucket. Each bucket stores the number of objects it represents, their average width and height, and their average complexity, as well as the boundaries of the rectangular region containing these objects. The objects within a bucket are assumed to be uniformly distributed. SQ-histograms are built off-line as part of updating the database statistics. If the database is update intensive, the SQ-histograms could potentially become inaccurate and should therefore be periodically rebuilt to ensure their accuracy.

2.4.1 Partitioning the Data into Buckets

The goal of partitioning the data into buckets is to have each bucket represent a "homogeneous" set of objects. This makes assuming uniformity within a bucket accurate and results in an accurate overall representation of the data distribution. Minimizing variation within a bucket is a common goal for all histogram techniques. The properties of the data that should be taken into account by the partitioning algorithm are:

- The location of the objects. A bucket should represent objects that are close to each other in the space. This minimizes the "dead space" within a bucket. Similar rules are used for histograms for traditional data.

- The size (area) of the objects. The size of the objects in a bucket determines the expected number of objects in this bucket that overlap a query window. The larger the objects in a bucket, the more likely they are to overlap a query window. Accurate estimation, therefore, requires that the average size of the objects in a bucket be as close as possible to the true size of these objects. This means that grouping objects with widely varying sizes in the same bucket should be avoided.

- The complexity of the objects. Estimating the cost of the refinement step requires accurately estimating v_{cand}, the average complexity of the candidate polygons. For accurate estimation, the variation in object complexity within a bucket should be as small as possible. This is to make the average number of vertices per object in a bucket a close approximation of the true number of vertices. Grouping objects with widely varying complexities

in the same bucket should, therefore, be avoided.

SQ-histograms are built using a quadtree data structure [Sam84]. A quadtree is a recursive data structure in which each node represents a rectangular region of the space. A node can have up to four children, each representing a quadrant of the region that the parent node represents. Thus, a quadtree partitions the input space into four quadrants, and allows each quadrant to be further partitioned recursively into more quadrants (Figure 1). SQ-histograms partition the space into buckets using this quadtree partitioning.

The algorithm for building an SQ-histogram starts by building a *complete quadtree* with l levels for the space containing the data, where l is a parameter of the histogram construction algorithm. The different levels of this complete quadtree represent different sized partitionings of the space. Thus, the quadtree partitions the space at several different resolutions. We use this property to separate the data polygons according to size and location, with each polygon being assigned to a particular quadtree node at a particular level of the quadtree.

The quadtree *level* to which a polygon is assigned is the maximum level (i.e., the furthest from the root) such that the width and height of the MBR of the polygon are less than or equal to the width and height of the quadtree nodes at this level. Informally stated, this means that the polygon “fits” in a quadtree node at this level but not at higher levels. Note that choosing a quadtree level for a polygon depends only on the dimensions of the polygon MBR and not on its location.

After choosing a quadtree level for a polygon, we choose the quadtree node at this level that contains the center of the polygon MBR. The polygon is assigned to this node. Figure 1 demonstrates assigning polygons to quadtree nodes. For the purpose of illustration, the quadtree in this figure is not a complete quadtree.

After assigning all the polygons to quadtree nodes, the complete quadtree can be used as an accurate histogram. Each quadtree node represents a number of polygons with similar location and size. Polygons that are far from each other will have MBRs whose centers lie within different quadtree nodes, and polygons with widely varying sizes will be assigned to nodes at different levels of the quadtree.

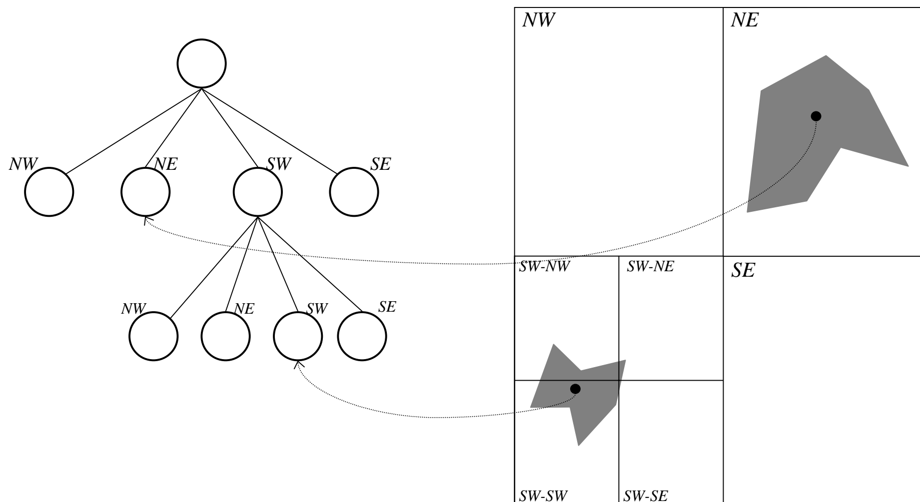


Figure 1: A quadtree partitioning and the assignment of polygons to quadtree nodes

The algorithm for assigning polygons to quadtree nodes does not take into account the complexity of the polygons. The algorithm assumes that polygons in the same vicinity and with similar sizes (and therefore assigned to the same quadtree node) have similar complexities. In the next section, we present a solution for cases in which this assumption does not hold.

The problem with the complete quadtree built by the initial phase of the SQ-histogram construction algorithm is that it may take too much memory. Database systems typically limit the amount of memory available to histograms, and an algorithm for building histograms must guarantee that they fit in the assigned memory. This translates to an upper bound on the number of buckets that a histogram can have. In our case, we can reduce the number of buckets by reducing the number of levels of the complete quadtree. However, this limits the granularity at which the space is partitioned. Instead, we want to start with a complete quadtree with as many levels as we wish, but still guarantee that the final histogram will fit in the assigned memory.

To satisfy this requirement, we start with a histogram in which the buckets correspond to the non-empty nodes of the complete quadtree. We repeatedly *merge* buckets corresponding to *sibling quadtree nodes* among which the data distribution has little variation, until the number of buckets drops to the required bound. We must choose a method of measuring the *variation in data distribution* among four histogram buckets (corresponding to sibling nodes in the quadtree). For example, we could use the variance of the number of polygons represented by

the buckets, or the maximum difference in the number of polygons represented by any two buckets. We use this measure to compute the variation in data distribution among every set of four buckets corresponding to four sibling nodes of the quadtree. Sets of siblings nodes at all levels of the quadtree are considered in this computation. After this computation, we merge the histogram buckets corresponding to the four sibling quadtree nodes with the *least variation in data distribution*.

To merge these buckets, we replace them with one new bucket that represents all the objects that they currently represent. The new bucket represents the same region of the space that is represented by the *parent* node of the quadtree nodes corresponding to the buckets being merged. Hence, the new bucket will correspond to this parent quadtree node. If, before merging, the parent quadtree node corresponded to one histogram bucket, after merging it will correspond to *two* buckets. It is important to note that these two buckets are kept separate, even though they represent the same region of the space, because they represent objects of different sizes that were assigned to different levels of the quadtree. After an object is assigned to a quadtree level, it is only combined with other objects from the same quadtree level. This guarantees that objects in a histogram bucket always have similar sizes.

The merging operation is repeated as many times as needed to satisfy the memory constraint. At each merging step, we compute the variation in data distribution among buckets that correspond to sibling quadtree nodes *and that contain objects assigned to the same quadtree level*, and we merge the buckets with the minimum variation. Repeatedly choosing the buckets with the least variation in data distribution can be done efficiently using a priority queue. Since we only merge buckets corresponding to sibling quadtree nodes, the partitioning of the space always remains a quadtree partitioning, and the same merging procedure can be repeated as many times as needed.

After choosing the histogram buckets, the boundaries of each bucket are set to the MBR of all the objects that it represents. This step is required because polygons can extend beyond the boundaries of the quadtree nodes to which they are assigned. It results in histogram buckets that potentially represent overlapping regions of the space. After this step, the regions represented by the histogram buckets no longer correspond exactly to the regions represented by the quadtree

```
algorithm BuildSQ-histogram
    Build a complete quadtree of height l representing the data space;
    Scan the data and assign polygons to quadtree nodes according to size and location;
    Set the histogram buckets to correspond to the quadtree nodes that contain data;
    while Current number of buckets > Required number of buckets do
        Merge the buckets corresponding to sibling quadtree nodes that have
            the minimum variation in data distribution;
    end while;
end BuildSQ-histogram;
```

Figure 2: Algorithm for building SQ-histograms

nodes. Thus, the quadtree cannot be used as an index to search for buckets that overlap a given query window. We use the quadtree only to build the SQ-histogram, not to search it at cost estimation time. At cost estimation time, a sequential search is used to determine the buckets that overlap the query window. Figure 2 presents an outline of the algorithm for building SQ-histograms. Note that this algorithm requires only one scan of the data.

2.4.2 Handling Objects With Varying Complexities

The above algorithm does not take into account the complexity of the polygons when creating the histogram buckets. To handle data sets in which polygons with similar sizes and locations may have widely varying complexities, we should build not one but *several* quadtrees, one for "low complexity" objects, one for "medium complexity" objects, and so on.

To build an SQ-histogram using this approach, we determine the minimum and maximum number of vertices of all the polygons in the data set. This requires an extra scan of the data. We also specify the number of quadtrees to build as a parameter to the algorithm. The range of vertices in the data set is divided into sub-ranges of equal width, where the number of sub-ranges is equal to the required number of quadtrees. Each quadtree represents all the objects with a number of vertices in one of these sub-ranges. We build the required number of complete quadtrees, and assign the data objects to quadtree nodes according to location, size, and complexity. We decide the quadtree to which an object is assigned based on the number of vertices of this object. To satisfy the memory constraint, we start with buckets corresponding to

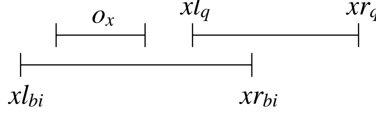


Figure 3: The x dimension of a query overlapping a bucket

all non-empty nodes in all quadtrees, and we repeatedly merge buckets until the histogram has the required number of buckets. When merging buckets, we only merge buckets corresponding to sibling nodes of the same quadtree. We merge the buckets with the least variation in data distribution among all sibling nodes of all quadtrees.

2.4.3 Assuming Uniformity Within a Bucket

To estimate the cost of a given query, we need to estimate s_{MBR} and v_{cand} for this query. This requires estimating the fraction of the objects in a bucket whose MBRs overlap the query MBR. This fraction of the objects in the bucket will appear in the result of the filtering step. We term this fraction f_i, where i is the index of the bucket in the histogram. In estimating f_i for a given histogram bucket, we assume a uniform distribution within this bucket.

Let b_i be a histogram bucket and q be a query MBR. If q does not overlap b_i, $f_i = 0$. If q totally encloses b_i, $f_i = 1$. If q is totally enclosed in b_i or partly overlaps it, f_i is the probability of q overlapping an object represented in b_i. Next, we consider computing this probability assuming the objects in b_i are uniformly distributed.

For q to overlap an object in b_i, it must overlap it in both the x and y (horizontal and vertical) dimensions. Consider the x dimension, and let the left and right boundaries of b_i be xl_{bi} and xr_{bi}, respectively. Let the left and right boundaries of q be xl_q and xr_q, respectively. Let o_x be the average width of the MBRs of the objects represented by b_i. Figure 3 illustrates these quantities.

Let fx_i be the probability of q overlapping an object represented in b_i, o, in the x dimension. fx_i is given by:

$$fx_i = 1 - \Pr\{o \text{ left of } q\} - \Pr\{o \text{ right of } q\}$$

Since we are assuming a uniform distribution within b_i, the leftmost point of o is uniformly

distributed between xl_{bi} and $xr_{bi} - o_x$. Thus, fx_i is given by:

$$fx_i = 1 - \frac{\max(xl_q - o_x - xl_{bi}, 0)}{xr_{bi} - o_x - xl_{bi}} - \frac{\max(xr_{bi} - o_x - xr_q, 0)}{xr_{bi} - o_x - xl_{bi}}$$

Similarly, the probability of q overlapping an object in the y dimension is given by:

$$fy_i = 1 - \frac{\max(yb_q - o_y - yb_{bi}, 0)}{yt_{bi} - o_y - yb_{bi}} - \frac{\max(yt_{bi} - o_y - yt_q, 0)}{yt_{bi} - o_y - yb_{bi}}$$

where yb_{bi} and yt_{bi} are the bottom and top boundaries of b_i, respectively, yb_q and yt_q are the bottom and top boundaries of q, respectively, and o_y is the average height of the MBRs of the objects represented by b_i. To estimate f_i, we use $f_i = fx_i * fy_i$.

From these formulas, we see that estimating f_i requires each histogram bucket to store the boundaries of the region that it represents, and the average width and height of the MBRs of the objects that it represents. To estimate s_{MBR} and v_{cand}, each bucket must also store the number of objects it represents and the average number of vertices in these objects.

2.4.4 Estimation Using SQ-histograms

To estimate s_{MBR} and v_{cand}, we use a sequential search of the histogram buckets to identify the buckets that overlap the MBR of the query window. We estimate the required quantities using the following formulas:

$$\begin{aligned} s_{MBR} &= \sum_{i \in B} f_i N_i \\ v_{cand} &= \frac{\sum_{i \in B} f_i N_i V_i}{s_{MBR}} \end{aligned}$$

where B is the set of indices in the histogram of buckets overlapping the query window MBR, N_i is the number of objects in bucket i, and V_i is the average number of vertices per object in bucket i.

SQ-histograms provide an estimate for the MBR selectivity of window queries, but not for their actual selectivity. We do not attempt to estimate the actual selectivity of window queries, as this would require information about the exact layout of the vertices of the query and data polygons. One cannot estimate whether or not two general polygons overlap based only on their

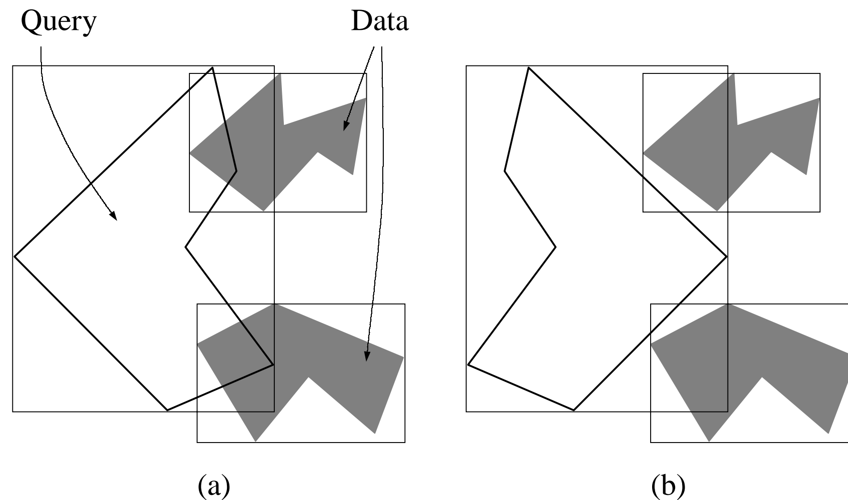


Figure 4: The difficulty of estimating actual selectivities. The query polygon overlaps the data polygons in (a) but not in (b)

MBRs, areas, or number of vertices. To demonstrate this, consider the two cases presented in Figure 4. The first case is a query polygon that overlaps two data polygons. The polygons in the second case are identical to the ones in the first case, except that the query polygon is flipped vertically. In the second case, the query polygon does not overlap either of the data polygons, despite the MBRs, areas, shapes and number of vertices being the same as in the first case.

The query optimizer can use the MBR selectivity estimated using an SQ-histogram as an upper bound on the actual selectivity of the query. This estimate may prove to be useful, especially since it is based on an accurate representation of the data distribution. Alternately, the actual selectivity of the query can be estimated using sampling (Section 2.5.7).

2.4.5 Comparison with MinSkew Partitioning

Acharya, Poosala, and Ramaswamy propose a partitioning scheme for building histograms for spatial data called *MinSkew partitioning* [APR99]. Like SQ-histograms, MinSkew partitioning is based on the MBRs of the data objects. Partitioning starts by building a uniform grid that covers the input space and determining the number of objects that overlap each grid cell. This grid is an approximation of the data distribution, and is used by MinSkew partitioning to construct the histogram.

The partitioning algorithm maintains a set of buckets currently in the histogram. This set initially contains one bucket representing the whole space. The algorithm repeatedly chooses a bucket from this set and splits it into two buckets until the histogram has the required number of buckets. The bucket to split and the split point are chosen to give the maximum reduction in *spatial skew*. The spatial skew of a bucket is defined as the variance of the number of objects in the grid cells constituting that bucket. The algorithm considers the space at multiple resolutions by building several grids at different resolutions and generating an equal number of histogram buckets from each grid. To reduce computation time, the splitting decision is based on the *marginal* frequency distributions of the grid cells in the buckets.

Both MinSkew partitioning and SQ-histograms have to choose a partitioning of the space from an intractably large number of possibilities. SQ-histograms deal with this problem by considering only quadtree partitionings of the space. MinSkew partitioning restricts itself to *binary space partitionings* along the grid lines, which is a more general set than quadtree partitionings. However, MinSkew partitioning based on the marginal frequency distribution uses a one-dimensional measure of variation to construct the multi-dimensional partitioning, while SQ-histograms use a multi-dimensional measure of variation.

Another advantage of SQ-histograms is taking the variation in object sizes into account. MinSkew partitioning only considers the number of objects that overlap a grid cell, and not the sizes of these objects. SQ-histograms, on the other hand, assign small and large objects to different quadtree levels and thus place them in different buckets.

The most important issue in comparing SQ-histograms and MinSkew partitioning is that SQ-histograms contain information about the complexity of the objects. This information is essential for accurate cost estimation. Our experiments in the next section demonstrate that SQ-histograms are more accurate than MinSkew partitioning, even if we add the number of vertices to the information stored in the MinSkew buckets.

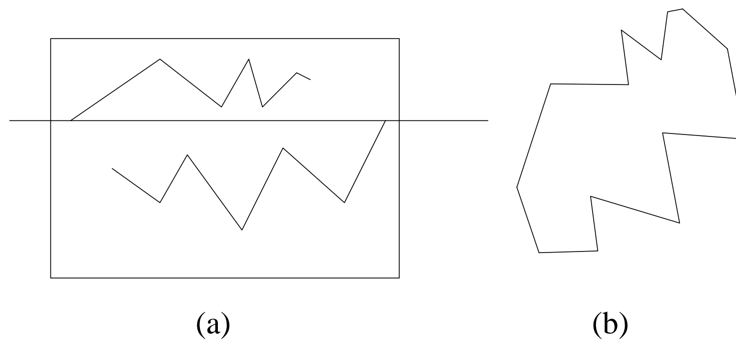


Figure 5: A synthetically generated polygon

2.5 Experimental Evaluation

2.5.1 Generating Synthetic Polygons

In our experiments, we need to generate random polygons for the test queries and the synthetic data sets. To generate a polygon, we start by choosing a rectangle in the space within which the polygon is generated. This rectangle specifies the size and location of the polygon. We then choose a number of points at random inside this rectangle. These points are the vertices of the polygon.

Next, we choose a random horizontal line that cuts through the rectangle, and divide the points into two groups: points that lie above this line and points that lie below it. The points in each of the groups are sorted by their x (horizontal) coordinate, and connected in the sorted order to create two "chains" of points. To avoid generating self-intersecting polygons, the leftmost and rightmost points of the two chains are moved vertically so that they lie on the splitting line.

Next, the two chains of points are connected at their end-points, forming a polygon. Finally, we rotate the polygon by a random angle to avoid generating polygons that are all horizontally aligned. This algorithm generates *monotone* polygons [O'R98], a very general class of polygons. Figure 5 gives an example of a polygon generated by this algorithm. Figure 5(a) shows the initial rectangle, the split line, and the two chains of points. Figure 5(b) shows the final polygon generated by connecting the two chains and rotating the polygon by a random angle.

2.5.2 Experimental Setup

Data Sets

We present results for one real and one synthetic data set (except in Section 2.5.3, where we use three other synthetic data sets). Results on other synthetic data sets corroborate the conclusions drawn here.

The real data set we use is the set of polygons representing land use in the state of California from the Sequoia 2000 benchmark [SFGM93]. This data set consists of 58,586 polygons having between 4 and 5,583 vertices, with an average of 56 vertices per polygon.

The synthetic data set we present here consists of 10,000 polygons generated using the procedure described above. The polygons have between 3 and 100 vertices, with an average of 20 vertices per polygon. 30% of the polygons are distributed uniformly throughout the space, and 70% of the polygons are distributed in three clusters at different parts of the space. The rectangles in which the points of the polygons were generated have areas between 0.0025% and 0.75% of the area of the space, and aspect ratios uniformly distributed in the range 1–3. Figure 6 presents a 20% sample of this data set.

Query Workloads

The number of vertices for the polygonal query windows is randomly chosen from the range 3–15. The polygons are generated inside rectangles of 9 different sizes, with areas ranging from 0.01% to 10% of the area of the space. Each workload contains 50 queries at each size, for a total of 450 queries.

When issuing a workload on some data set, we choose the centers of the rectangles in which the query polygons are generated at random from the centers of the MBRs of the data objects (i.e., the rectangles follow a distribution similar to the data [PSTW93]). Experiments with workloads in which the queries are uniformly distributed in the space gave the same conclusions. We use the same workload for each data set in all our experiments.

For the Sequoia data set, the average selectivities of the queries of different sizes are shown in Figure 7. The figure shows both the MBR selectivity (the selectivity of the filtering step) and

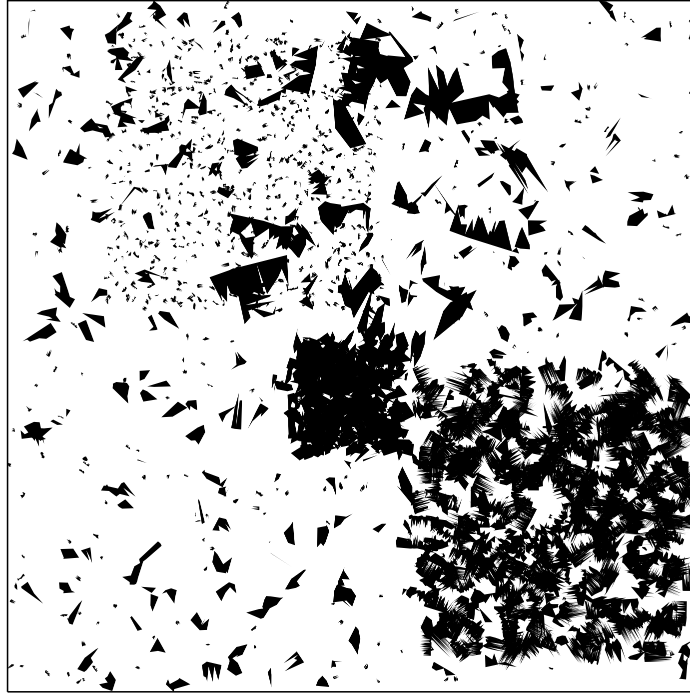


Figure 6: The synthetic polygon data set used in our experiments

the actual selectivity (the selectivity of the whole query after both filtering and refinement). In this figure, the selectivity of an operation is defined as the fraction of the objects in the data set that appear in the result of this operation. Figure 7 shows that the MBR selectivity of a window query is only a very loose upper bound for its actual selectivity.

Run-time Environment

Our experiments were conducted on a Pentium Pro 200 MHz machine with 128 MBytes of memory running Solaris 2.6. We used one disk for the database, another disk for the log, and a third disk for the software (the database system and our test programs, as well as any working files and result files). Our experiments were conducted on the university version of the Paradise object-relational database system [P$^+$97]. Both the server and the client programs were run on the same machine, and the server buffer pool size was set to 32 MBytes.

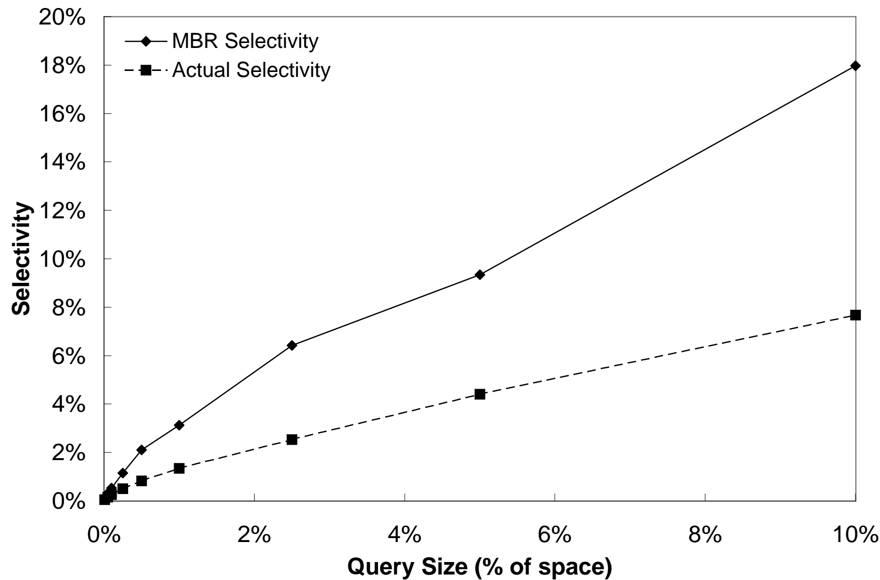


Figure 7: Query selectivity (Sequoia data set)

Error Metric

In measuring the estimation accuracy of the various techniques, we use the *average relative estimation error* as our error metric. The relative error in estimating a quantity, x, for one query, q, is defined as:

$$e_q = \frac{|\text{estimated value of } x - \text{measured value of } x|}{\text{measured value of } x}$$

For a set of M queries, the average relative estimation error is defined as:

$$E = \frac{\sum_{i=1}^{M} e_i}{M}$$

Queries with a result size of zero are removed from the test run. Since the query distribution in our experiments is similar to the data distribution, we encounter very few queries with a result size of zero.

2.5.3 Importance of the Cost of Refinement

In this section, we illustrate the significance of the cost of refinement, and the importance of including it in any window query cost model. The data sets and query workloads used in this section are different from those used in our remaining experiments.

Vertices	Execution time (sec)
10	0.2
100	0.5
1000	3.4

Table 2: Cost of a window query with MBR area equal to 1% of space

To generate the data sets for this experiment, we generate a set of 10,000 uniformly distributed squares, each with an area of 0.01% of the area of the space. We generate three data sets, each consisting of randomly generated polygons having these squares as their MBRs. The polygons of the first, second, and third data sets have 10, 100, and 1000 vertices, respectively. Since the polygons of the three data sets have the same MBRs, the filtering step for any window query will be the same for all three data sets.

Table 2 presents the average execution time for 50 window queries on each data set, starting with a cold buffer pool. The same queries are executed for all data sets. The query windows are random polygons with 100 vertices each, and their MBRs are squares with an area of 1% of the area of the space. The centers of these MBRs are randomly chosen from the centers of the data polygon MBRs. R-tree indexes are available for all data sets, and they are used by all the queries.

There is a significant difference in execution time between the three data sets. The MBRs of the polygons in the three data sets are the same. Therefore, the filtering step of a window query will be the same for all three data sets. The difference in execution time is due to differences in the refinement step.

This demonstrates the need to incorporate the cost of refinement in estimating the cost of window queries. Any window query cost model that does not take into account the complexity of the data and query polygons, and does not estimate the cost of the refinement step, will not be able to distinguish between these three data sets. Even if such a cost model accurately estimates the execution time for one data set, it will be inaccurate for the other two.

Table 2 also illustrates the significance of the cost of refinement, thereby validating one of our main premises. If we take the execution time for the 10-vertex data set to be an approximation of the filtering time for all three data sets, we see that refinement for the 1000-vertex data

set is over an order of magnitude more expensive than filtering.

In the next sections, we demonstrate the effectiveness of our proposed techniques in estimating the cost of window queries, including the cost of the refinement step. The remainder of our experiments use the real and synthetic data sets and workloads described in Section 2.5.2.

2.5.4 Estimation Accuracy Using SQ-histograms

In this section, we demonstrate the accuracy of SQ-histograms in estimating s_{MBR} and v_{cand} compared to MinSkew partitioning and assuming uniformity. We compare to MinSkew partitioning because it is identified as a winner among several techniques in [APR99]. We compare to assuming uniformity because it is the simplest approach in the absence of information about the data distribution. To allow MinSkew partitioning to be used for estimating v_{cand}, we have each bucket store the average complexity of the objects it represents, in addition to the information required in [APR99].

The SQ-histograms are given 5 KBytes of memory. They are built starting with 10 complete quadtrees of 8 levels each. We use 10 quadtrees to accommodate the varying complexities of the data objects. The histograms are built using the "maximum difference in the number of objects" to measure the variation in distribution among the quadtree nodes (Section 2.5.6 provides a detailed study of the effect of the different parameters of histogram construction). MinSkew partitioning is also given 5 KBytes of memory. We start the MinSkew partitioning with a 25×25 grid. This grid is progressively refined two times, so that the final buckets are generated from a 100×100 grid.

Figures 8–11 present the error in estimating s_{MBR} and v_{cand} for the Sequoia and synthetic data sets. Each point in the figures represents the average relative estimation error for 50 queries of a particular size. The figures show that using a histogram is always more accurate than assuming uniformity, and that SQ-histograms are generally more accurate than MinSkew partitioning. The figures also show that SQ-histograms are accurate enough in the absolute sense to be useful to a query optimizer. The irregularities in Figures 8 and 10 are due to one or two queries per data point that have a filtering step with a small measured result size appearing in the denominator

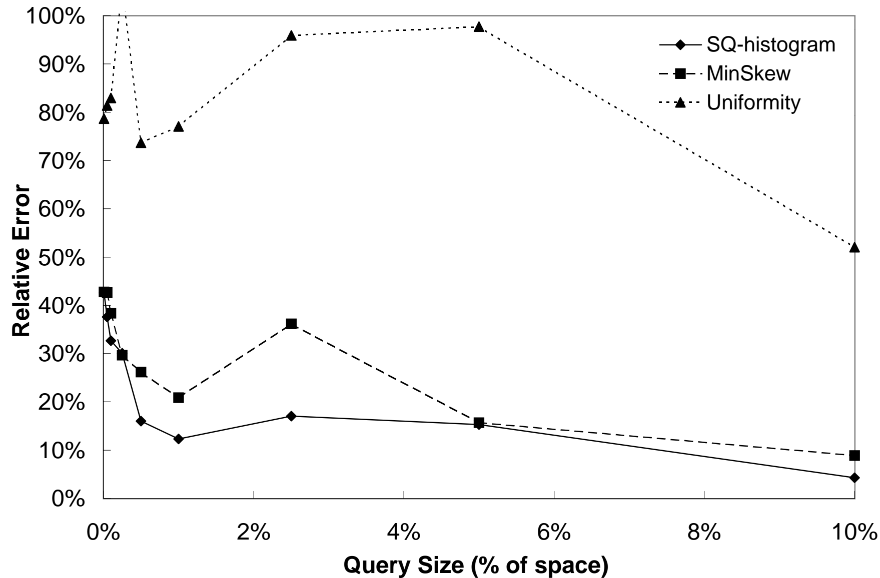


Figure 8: Error in estimating s_{MBR} for the Sequoia data set

Parameter	Cold Buffer Pool	Warm Buffer Pool
c_{seqio}	5×10^{-3}	8×10^{-4}
c_{randio}	1.5×10^{-2}	8×10^{-4}
c_{polyio}	5×10^{-7}	5×10^{-7}
c_{vertio}	2.5×10^{-5}	8×10^{-6}
$c_{MBRtest}$	0	0
$c_{polytest}$ (log base 10)	1.5×10^{-5}	1.5×10^{-5}

Table 3: Calibration constants for the window query cost model

of the error formula, thus leading to large relative estimation errors.

2.5.5 Accuracy of the Window Query Cost Model

We determine two sets of calibration constants for the window query cost model presented in Section 2.3. One set of constants is for a cold buffer pool and the other is for a warm buffer pool. These constants calibrate the cost model for use with Paradise in our run-time environment to estimate the execution time of window queries in seconds. The values of these constants are shown in Table 3.

Figures 12–15 show the actual execution times of the workloads on the Sequoia and synthetic data sets. The figures show the execution times when we start with a cold buffer pool for

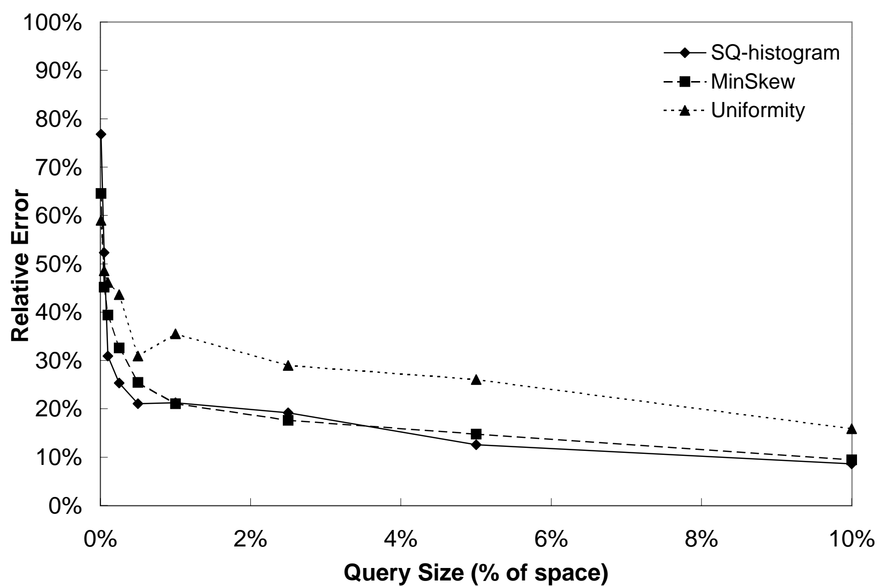


Figure 9: Error in estimating v_{cand} for the Sequoia data set

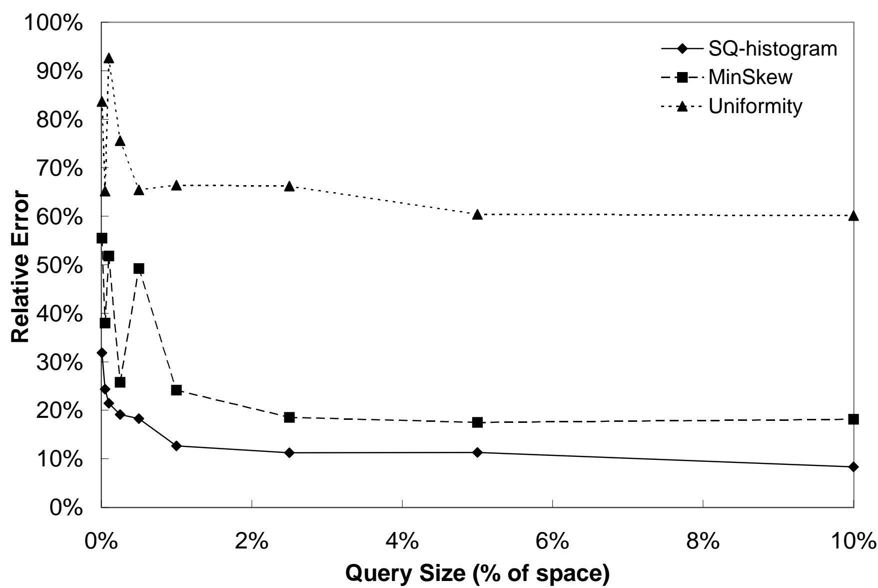


Figure 10: Error in estimating s_{MBR} for the synthetic data set

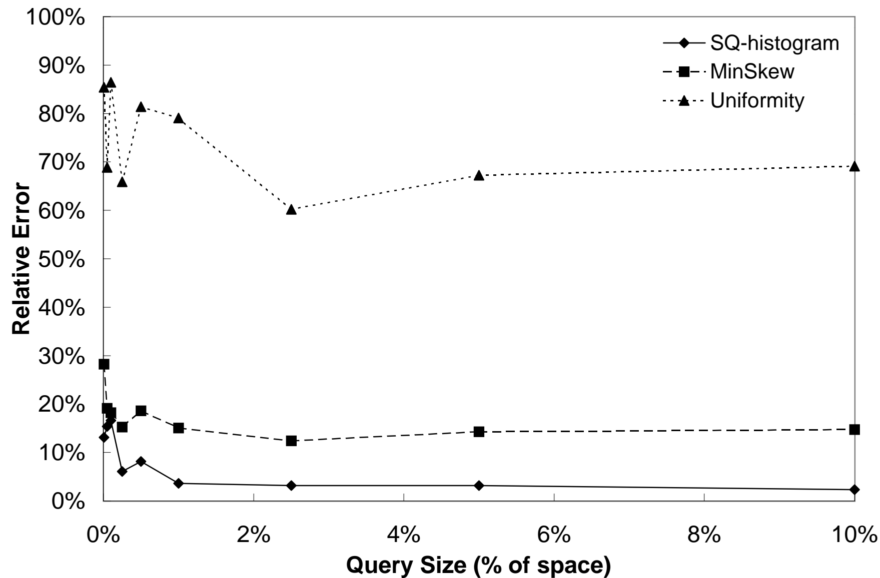


Figure 11: Error in estimating v_{cand} for the synthetic data set

every query (i.e., when the buffer pool is flushed between queries), and when the buffer pool is kept warm (i.e., not flushed between queries). An R-tree index is available, but the query optimizer may choose not to use it for queries with large areas and, hence, a large expected selectivity. The figures also show the estimated execution times using the calibration constants in Table 3 and with s_{MBR} and v_{cand} estimated using SQ-histograms built using the parameters described in the previous section. Each point in the figures is the average execution time for 50 queries of a particular size.

The figures show that, even with the variability in execution time, with the simplifying assumptions made by the cost model, and with the estimation errors introduced by histograms, the cost model still estimates the overall execution times of the window queries relatively accurately. While the estimated time does not, in general, match the actual time exactly, it is likely to be good enough for query optimization. The goal of query optimization is to distinguish between plans with widely varying costs and avoid very bad plans. Distinguishing between plans with similar costs is not necessary for query optimization. As such, the accuracy of our cost model should be adequate.

The cost model is more accurate for a warm buffer pool than it is for a cold buffer pool. A warm buffer pool reduces the variability in query execution time, making cost estimation easier.

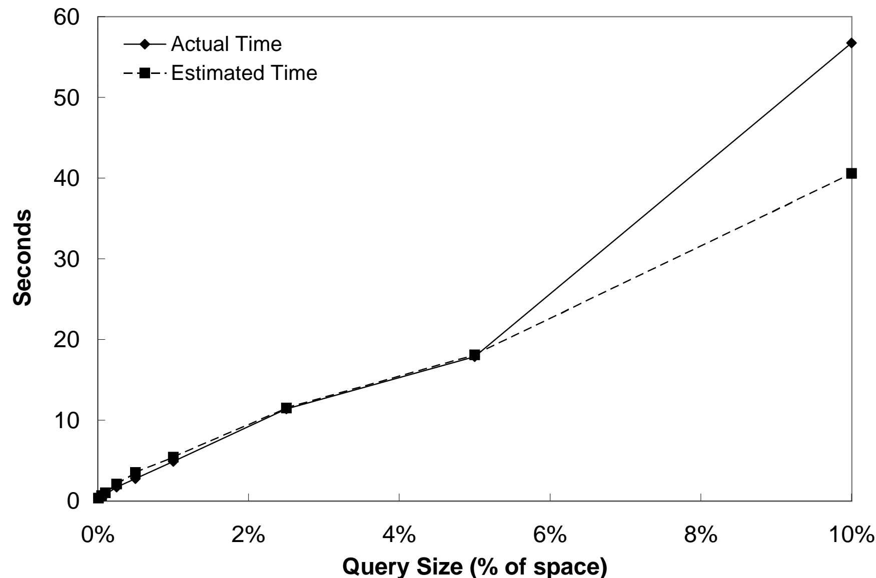


Figure 12: Execution times with a cold buffer pool for the Sequoia data set

The cost model is also more accurate for the Sequoia data set than it is for the synthetic data set. Queries on the Sequoia data set have longer execution times, so estimation accuracy is more important for this data set. On the other hand, the short execution times of the queries on the synthetic data set make small estimation errors appear more pronounced.

2.5.6 Effect of Histogram Construction Parameters

Next, we turn our attention to the effect of the different parameters of the SQ-histogram construction algorithm. The default histogram for this experiment uses 5 KBytes of memory, and is built starting with one 10-level quadtree using "maximum difference in the number of objects" to measure the variation in data distribution. We vary each of the histogram construction parameters in turn and show that the histogram is robust under all these variations. The errors shown in this section are average errors for all the queries of the workload.

Figure 16 shows the effect of the amount of memory available to a histogram on its accuracy. The figure shows the error in estimating s_{MBR} for the Sequoia data set using SQ-histograms and MinSkew partitioning occupying the same amount of memory. SQ-histograms are more accurate than MinSkew partitioning for the whole range of available memory. As expected, more available memory results in more estimation accuracy. Notice, though, that the error at 5

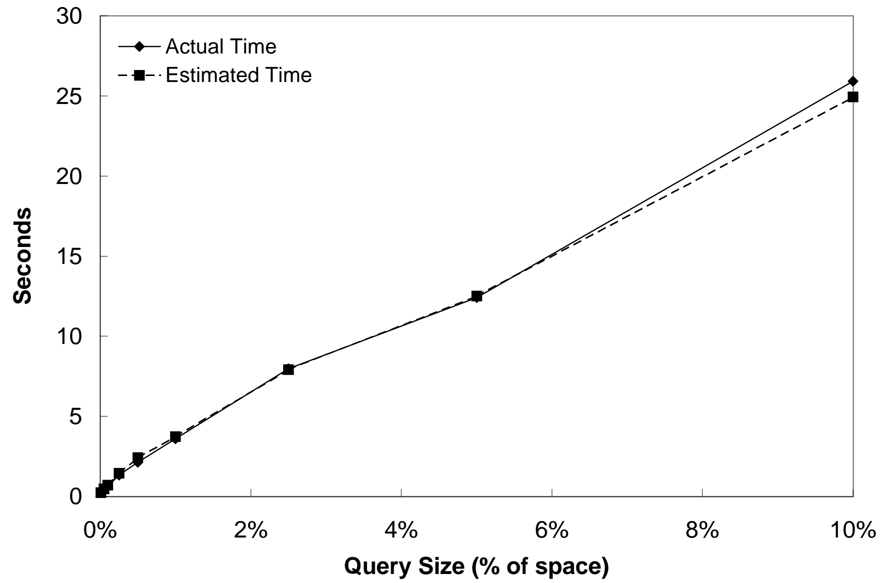


Figure 13: Execution times with a warm buffer pool for the Sequoia data set

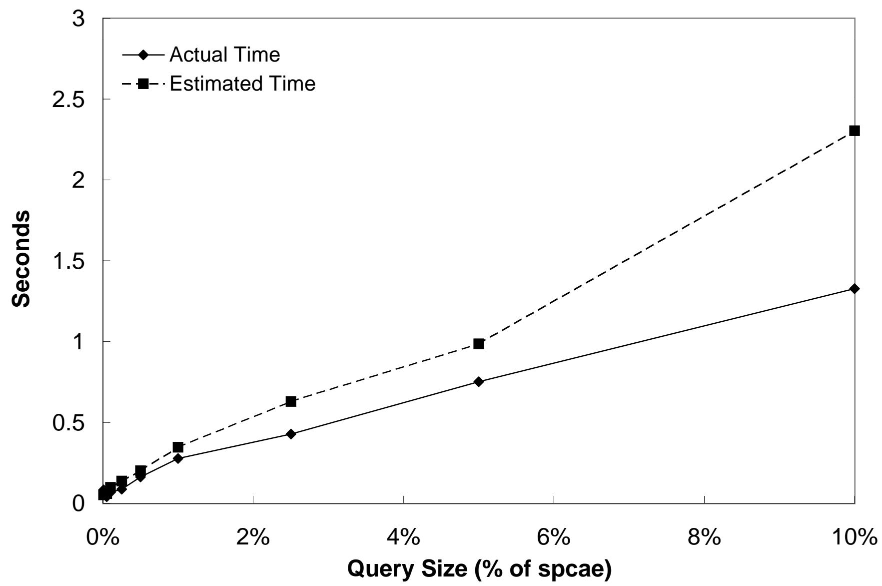


Figure 14: Execution times with a cold buffer pool for the synthetic data set

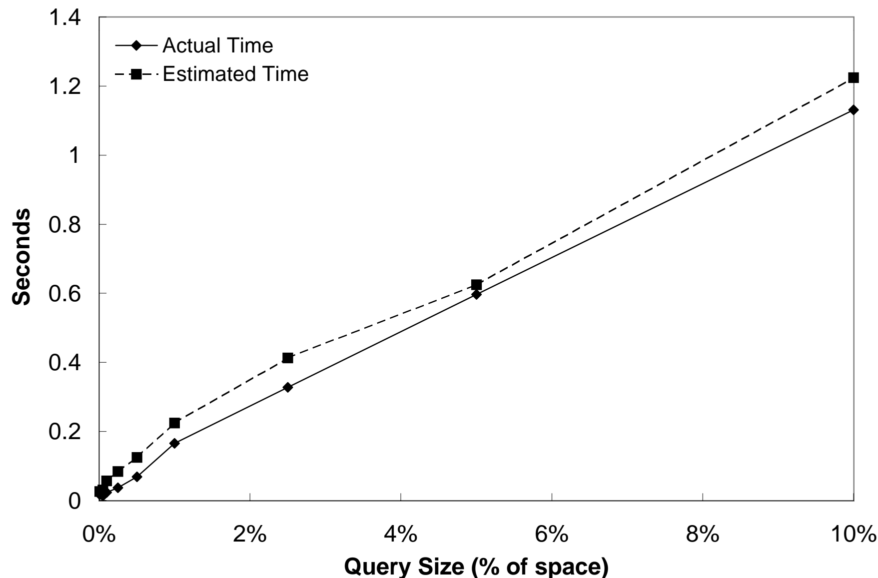


Figure 15: Execution times with a warm buffer pool for the synthetic data set

KBytes is already reasonable, and that the slope of the error beyond this point is small.

Figure 17 shows the effect of the number of levels in the initial complete quadtree on the accuracy of SQ-histograms in estimating s_{MBR} for the Sequoia and synthetic data sets. Starting with more quadtree levels is generally better, as it allows the histogram to consider the space at a finer granularity. Furthermore, using more levels allows for better separation of objects according to size. However, having too many levels may actually increase the error by creating a histogram with an unnecessarily large number of small buckets. The most important observation, though, is that the error is relatively flat for a wide range of initial quadtree levels. The histogram construction algorithm is not overly sensitive to this parameter.

Next, we compare SQ-histograms constructed using different measures of variation in the data distribution. We experiment with three different measures of variation. The first is the maximum difference between the number of objects in the different buckets. The second is the maximum difference between the number of objects in the different buckets *relative to* the maximum number of objects any of the buckets. The third is the variance of the number of objects in the buckets. We also try choosing the buckets to merge based on the total number of objects in these buckets. Under this scheme, we merge the buckets in which the total number of objects is minimum. This scheme tries to construct histograms where the buckets all have

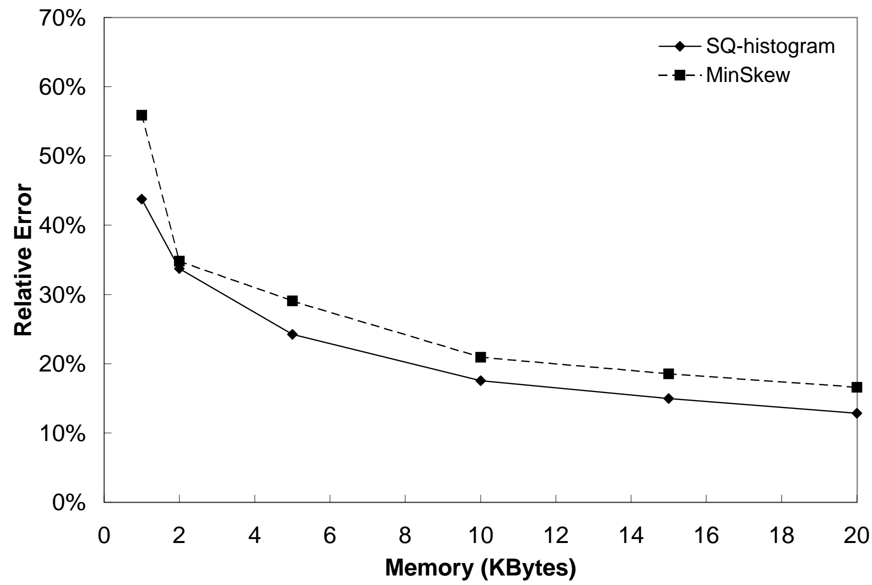


Figure 16: Effect of available memory on estimating s_{MBR} for the Sequoia data set

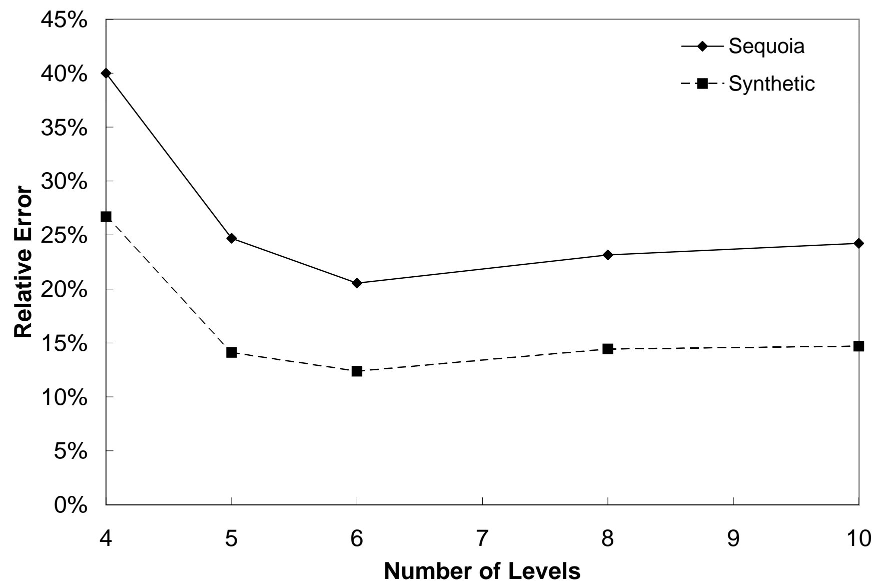


Figure 17: Effect of the number of levels in the initial quadtree on estimating s_{MBR}

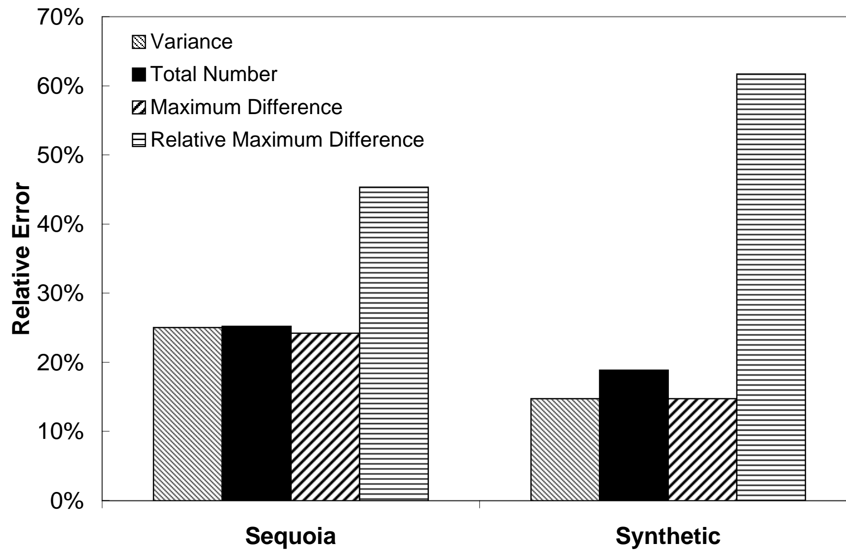


Figure 18: Effect of the measure of variation on estimating s_{MBR}

the same number of objects, similar to equi-depth histograms for traditional data [PIHS96]. Figure 18 presents the error in estimating s_{MBR} using SQ-histograms constructed using the different measures of variation. Maximum difference is the winner by a tiny margin. More importantly, we notice that the histogram is robust across three of the four methods.

In the interest of space, we do not present the results for starting with different numbers of quadtrees for different object complexities. The number of quadtrees does affect histogram accuracy, but the effect is small.

2.5.7 Using Sampling for Selectivity Estimation

In this section, we consider using sampling for selectivity estimation. Figure 19 presents the accuracy of using sampling to estimate the MBR selectivity and the actual selectivity for the Sequoia data set (similar results were obtained for the synthetic data set). The figure presents the errors for sample sizes of 100 and 200 random tuples. Sampling is very inaccurate for queries with low selectivity because most of the samples taken are negative samples (i.e., do not satisfy the selection predicate). Thus, the figure presents the average errors for all queries in the workloads with actual selectivities $> 1\%$.

With this number of samples, sampling is less accurate than SQ-histograms for estimating

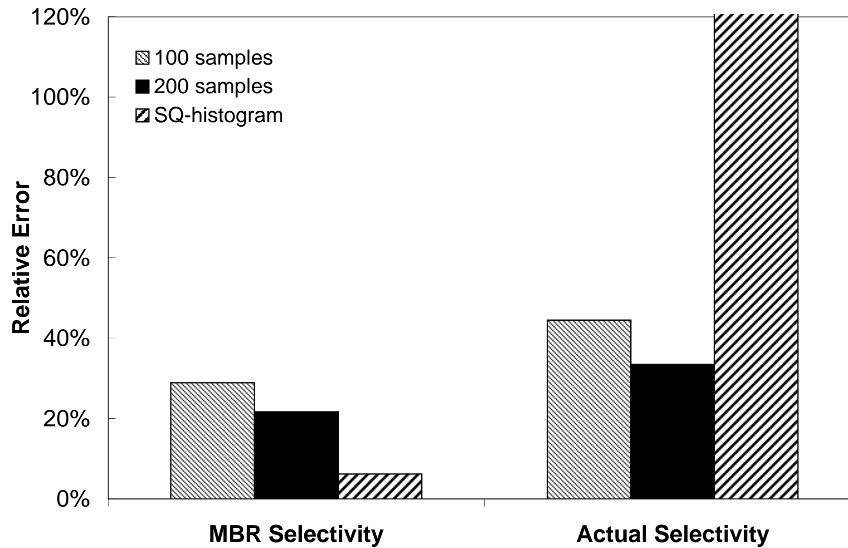


Figure 19: Selectivity estimation using sampling for the Sequoia data set

MBR selectivities. The key advantage of sampling is that, since it accesses and tests the actual data objects, it can be used to accurately estimate actual selectivities. Histograms provide only summary information that does not reflect the exact layout of the data objects, and hence cannot be used to estimate actual selectivities. Using the MBR selectivities estimated using the histograms as estimates of the actual selectivities leads to large errors (shown in the figure).

The disadvantage of sampling is its cost. Sampling involves the I/O cost of fetching the sampled tuples, as well as the high CPU cost of the exact geometry test for the objects in the sample. In our experiments, we found that taking a positive sample of one polygon (i.e., a sample where the polygon does overlap the query window) takes up to 25 ms when all the required indexes are buffered. A negative sample can often be detected by testing the MBRs of the query and polygon. In this case, the sample usually takes less than 1 ms if the indexes are in the buffer pool. Thus, the argument that sampling is expensive, which is often made in the context of traditional data, is more pronounced in the context of spatial data.

As expected, estimation accuracy increases with increasing the number of samples. Hence, one can reduce the error as desired by increasing the number of samples.

2.6 Conclusions

Accurate estimation of the cost of spatial selections requires taking into account the CPU and I/O costs of the refinement step. This requires estimating the MBR selectivity of the query and the average number of vertices in the candidate polygons identified by the filtering step.

SQ-histograms effectively estimate these two quantities and can be used to provide reasonably accurate cost estimates. SQ-histograms are also robust for a wide range of histogram construction parameters.

Sampling can also be used to estimate these two quantities. Sampling does not work well for very selective queries. For other queries, sampling offers the additional benefit of accurately estimating the actual selectivity of the query in addition to its MBR selectivity. However, sampling from spatial databases is expensive because each sample requires an expensive polygon overlap test.

Estimating the cost of spatial operations, in general, requires information about the location, size, and complexity of the data objects. We demonstrated how to effectively capture these properties using SQ-histograms, and how to use them for accurate estimation of the cost of spatial selections.

Chapter 3

Self-tuning Histograms: Building Histograms Without Looking at Data

3.1 Introduction

The previous chapter presented an example of building histograms for *non-traditional data types* (spatial data). In this chapter, we present an example of histograms for *traditional* data types, but built in a non-traditional way: without looking at the data.

Histograms are very useful for database systems because they represent information about the distribution of the data in a database. This information is primarily used by query optimizers for selectivity estimation, but it can also be used for other purposes such as approximate query processing, load balancing in parallel database systems, and guiding the process of sampling from a relation. Histograms are used in most commercial database systems such as Microsoft SQL Server, Oracle, and IBM DB/2.

While histograms impose very little cost at query optimization time, the cost of building them and maintaining or rebuilding them when the data is modified has to be considered when we choose the attributes or attribute combinations for which we build histograms. Building a histogram involves scanning or sampling the data, and sorting the data and partitioning it into buckets, or finding quantiles. For large databases, the cost is significant enough to prevent us from building all the histograms that we believe are useful. This problem is particularly striking for multi-dimensional histograms that capture joint distributions of correlated attributes [MD88, PI97].

Multi-dimensional histograms can be very useful for optimizing decision-support queries

since they provide valuable information that helps in estimating the selectivities of multi-attribute predicates on correlated attributes. Despite their potential, to the best of our knowledge, no commercial database system supports multi-dimensional histograms. The usual alternative to multi-dimensional histograms is to assume that the attributes are independent, which enables us to use a combination of one-dimensional histograms. This approach is efficient but *very* inaccurate. A more accurate approach that can help the query optimizer choose better query execution plans would be useful.

3.1.1 Self-tuning Histograms

We explore a novel approach that helps reduce the cost of building and maintaining histograms for large tables. Our approach is to build histograms *not* by examining the data but by using feedback information about the execution of the *queries on the database (the query workload)*. We start with an initial histogram built with whatever information we have about the distribution of the histogram attribute(s). For example, we will construct an initial two-dimensional histogram from two existing one-dimensional histograms assuming independence of the attributes. As queries are issued on the database, the query optimizer uses the histogram to estimate selectivities in the process of choosing query execution plans. Whenever a plan is executed, the query execution engine can count the number of tuples produced by each operator. Our approach is to use this "free" feedback information to refine the histogram. Whenever a query uses the histogram, we compare the estimated selectivity to the actual selectivity and refine the histogram based on the selectivity estimation error. This incremental refinement progressively reduces estimation errors and leads to a histogram that is accurate for query workloads similar to the workload used for refinement. We call histograms built using this process *self-tuning histograms*, or *ST-histograms* for short.

ST-histograms make it possible to build multi-dimensional histograms incrementally with little overhead, thus providing commercial systems with a low-cost approach to creating and

maintaining such histograms. ST-histograms have a low up-front cost because they are initialized without looking at the data. The refinement of ST-histograms is a simple low-cost procedure that leverages “free” information from the execution engine. Furthermore, we demonstrate that histogram refinement converges quickly. Thus, the overall cost of ST-histograms is much lower than that of traditional multi-dimensional histograms, yet the accuracy of both types of histograms is comparable.

An ST-histogram can be refined *on-line* or *off-line* (Figure 20). In the on-line mode, the module executing a range selection immediately updates the histogram. In the off-line mode, the execution module writes every selection range and its result size to a *workload log*. Tools available with commercial database systems, such as Profiler in Microsoft SQL Server, can accomplish such logging. The workload log is used to refine the histogram in a batch at a later time. On-line refinement ensures that the histogram reflects the most up-to-date feedback information but it imposes more overhead during query execution than off-line refinement and can also cause the histogram to become a high-contention hot spot.

The overhead imposed by histogram refinement, whether on-line or off-line, can easily be tailored. In particular, the histogram need not be refined in response to every single selection that uses it. We can choose to refine the histogram only for selections with a high selectivity estimation error. We can also skip refining the histogram during periods of high load or when there is contention for accessing it.

Refinement of ST-histograms makes them automatically adapt to database updates. Refinement brings an ST-histogram closer to the actual data distribution, whether the estimation error driving this refinement is due to the initial inaccuracy of the histogram or to modifications in the underlying data. Another advantage of ST-histograms is that their accuracy depends on how often they are used. The more an ST-histogram is used, the more it is refined, and the more accurate it becomes.

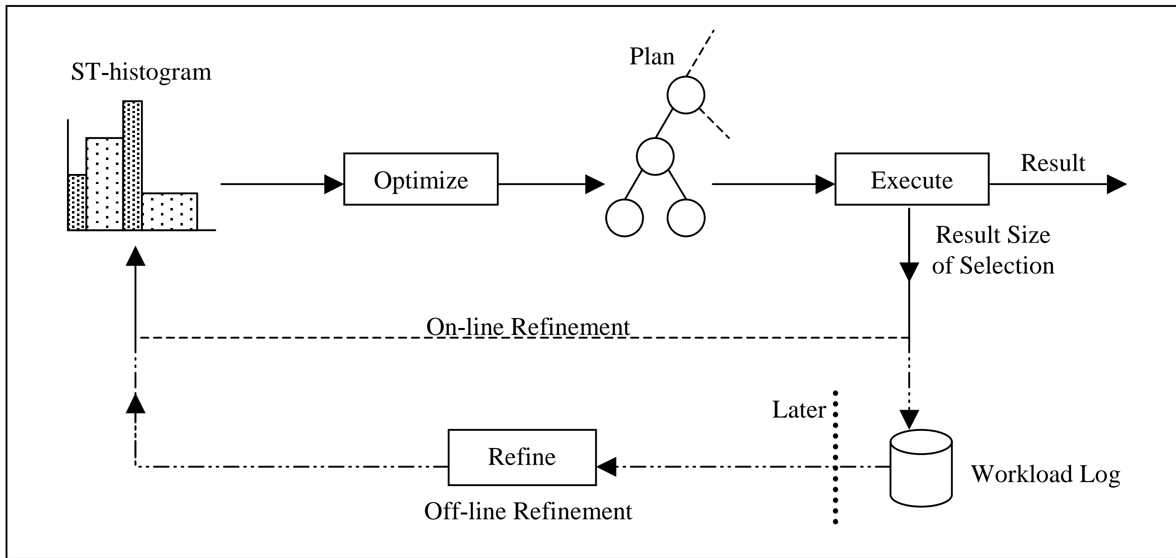


Figure 20: On-line and off-line refinement of ST-histograms

3.1.2 Applications of Self-tuning Histograms

One can expect traditional histograms built by looking at the data to be more accurate than ST-histograms that "learn" the data distribution without ever looking at the data. Nevertheless, ST-histograms are suitable for a wide range of applications.

Multi-dimensional ST-histograms are particularly attractive. Traditional multi-dimensional histograms, such as MHIST-p histograms [PI97], are significantly more expensive than traditional one-dimensional histograms, increasing the value of the savings in cost offered by ST-histograms. Furthermore, as our experiments in Section 3.5 demonstrate, ST-histograms are competitive in terms of accuracy with MHIST-p histograms for many data distributions. Multi-dimensional ST-histograms can be initialized using traditional one-dimensional histograms and subsequently refined to provide a cheap and efficient way of capturing the joint distribution of multiple attributes. The other, admittedly inexpensive, alternative of assuming independence has been repeatedly demonstrated to be inaccurate (see, for example, [PI97] and Section 3.5). Furthermore, note that building traditional histograms is an off-line process, meaning that histograms cannot be used until the system incurs the full cost of completely building them. This is not true of ST-histograms. Finally, note that ST-histograms make it possible to inexpensively

build not only two-dimensional, but also n-dimensional histograms.

ST-histograms are also a suitable alternative when there is not enough time for updating database statistics to allow building all the desired histograms in the traditional way. This may happen in data warehouses that are updated periodically with huge amounts of data. The sheer data size may prohibit rebuilding all the desired histograms during the batch window. This very same data size makes ST-histograms an attractive option, because examining the *workload* to build histograms will be cheaper than examining the data and can be tailored to a given time budget.

The technique of ST-histograms can be an integral part of database servers as we move toward self-tuning database systems. If a self-tuning database system decides that a histogram on some attribute or attribute combination may improve performance, it can start by building an ST-histogram. The low cost of ST-histograms allows the system to experiment more extensively and try out more histograms than if traditional histograms were the only choice. Subsequently, the system can construct a traditional histogram if the ST-histogram does not provide the required accuracy.

Finally, an intriguing possible application of ST-histograms will be for applications that involve queries on remote data sources. With recent trends in database usage, query optimizers will have to optimize queries involving remote data sources not under their direct control, e.g., queries involving data sources accessed over the Internet. Accessing the data and building traditional histograms for such data sources may not be easy or even possible. Query results, on the other hand, are always available from the remote sources, making the technique of ST-histograms an attractive option.

The rest of this chapter is organized as follows. In Section 3.2, we present an overview of related work. Section 3.3 describes one-dimensional ST-histograms and introduces the basic concepts that lead toward Section 3.4, where we describe multi-dimensional ST-histograms. Section 3.5 presents an experimental evaluation of our proposed techniques using real and synthetic data sets. Section 3.6 contains concluding remarks.

3.2 Related Work

Histograms were introduced in [Kooi80], and most commercial database systems now use one-dimensional equi-depth histograms for selectivity estimation. More accurate histograms have been proposed in [PIHS96]. The techniques proposed in [PIHS96] are extended to multiple dimensions in [PI97]. A novel approach for building histograms based on wavelets is presented in [MVW98].

A major disadvantage of histograms is the cost of building and maintaining them. Some work has addressed this shortcoming. [MRL98] proposes a one-pass algorithm for computing approximate quantiles that could be used to build approximate equi-depth histograms in one pass over the data. [GMP97] focuses on reducing the cost of maintaining equi-depth and compressed histograms. These two papers present techniques for lowering the cost of histograms by examining the data and processing it in better, novel ways. Our approach is not to examine the data at all, but to build histograms using feedback from the query execution engine. However, our technique for refining ST-histograms shares commonalities with the split and merge algorithm proposed in [GMP97]. This relationship is further discussed in Section 3.3.

The concept of using feedback from the query execution engine to estimate data distributions is introduced in [CR94]. In that paper, the data distribution is represented as a linear combination of *model functions*. Feedback information is used to adjust the weighting coefficients of this linear combination by a method known as the *recursive least square error method*. The paper only considers one-dimensional distributions. It remains an open problem whether one can find suitable multi-dimensional model functions, or whether the recursive least square error method would work well for multi-dimensional distributions. In contrast, we show how our technique can be used to construct multi-dimensional histograms as well as one-dimensional histograms. Furthermore, our work is easily integrated into existing systems because we use the same histogram data structures that are currently supported in commercial systems.

A different type of feedback from the execution engine to the optimizer is proposed in [KD98]. In that paper, the execution engine invokes the query optimizer to re-optimize a query if it believes, based on statistics collected during execution, that this will result in a better

query execution plan.

Subsequent to publishing the work presented in this chapter in [AC99], Bruno, Chaudhuri, and Gravano developed a new type of multi-dimensional histogram built using feedback information from query execution, which they call *ST-Holes* [BCG01]. ST-Holes histograms allow the histogram buckets to be completely contained within other buckets, which means that the histogram can be conceptually viewed as a tree structure (this bears some resemblance to the SQ-histograms that we propose for spatial data in Chapter 2). Furthermore, while the ST-histograms that we propose are refined based on selectivity estimation errors, ST-Holes histograms are refined based on observing the values of the tuples in the query result streams. They can, therefore, be more accurate than ST-histograms, but at the cost of higher refinement overhead.

3.3 One-dimensional ST-histograms

Although we propose ST-histograms mainly as a low cost alternative to traditional *multi*-dimensional histograms, the fundamentals of ST-histograms are best introduced using *one*-dimensional ST-histograms (ST-histograms for single attributes). One-dimensional ST-histograms are similar in structure to traditional histograms. A one-dimensional ST-histogram consists of a set of buckets. Each bucket, b, stores the range that it represents, $[low(b), high(b)]$, and the number of tuples in this range, or the *frequency*, $freq(b)$. Adjacent buckets share the bucket endpoints, and the ranges of all the buckets together cover the entire range of values of the histogram attribute. We assume that the refinement of ST-histograms is driven by feedback from *range selection queries*.

The life cycle of an ST-histogram consists of two stages. First, the histogram is *initialized* assuming the data is uniformly distributed. An ST-histogram always assumes that the data is uniformly distributed until the feedback observations contradict this assumption. Next, the histogram is *refined* based on feedback observations. The process of refinement can be broken down further into two parts: (a) refining individual *bucket frequencies*, and (b) *restructuring* the histogram, i.e., moving the *bucket boundaries*. The refinement process is driven by a query

workload consisting of multiple queries. The bucket frequencies are updated with every range selection on the histogram attribute, while the bucket boundaries are updated by *periodically restructuring* the histogram. Histogram refinement can be viewed as relaxing the uniformity assumption in response to feedback information. We describe these steps in detail in the rest of this section.

3.3.1 Initial Histogram

To build an ST-histogram, h, on an attribute, a, we need to know the required number of histogram buckets, B, the number of tuples in the relation, T, and the minimum and maximum values of attribute a, min and max. The B buckets of the initial histogram are evenly spaced between min and max. At the time of initializing the histogram structure, we have no feedback information. Therefore, we make the uniformity assumption and assign each of the buckets a frequency of T/B tuples (with some provision for rounding).

The parameter T can be looked up from system catalogs maintained for the database. However, the system may not store minimum and maximum values of attributes in its catalogs. Fortunately, the precise value of the minimum and maximum is not critical. Therefore, the initialization phase of ST-histograms can exploit additional sources to obtain estimates for these values that may subsequently be refined. For example, domain constraints on the column or the minimum and maximum values referenced in the query workload can be used for such estimation.

3.3.2 Refining Bucket Frequencies

The bucket frequencies of an ST-histogram are refined (updated) with feedback information from the queries of the workload. For every selection on the histogram attribute, we compute the *absolute estimation error*, which is the difference between the estimated and actual result sizes. Based on this error, we refine the frequencies of the buckets that were used in estimation.

The key problem is deciding how to distribute the “blame” for the estimation error among the histogram buckets that overlap the selection range of a given query. In an ST-histogram,

```
algorithm UpdateFreq
Inputs: h, rangelow, rangehigh, act
Outputs: h with updated bucket frequencies

begin
    Get the set of k buckets overlapping the selection range, {b_1, b_2, ..., b_k};
    est = Estimated result size of the range selection using histogram h;
    esterr = act - est;        /* Compute the absolute estimation error. */
    /* Distribute the error among the k buckets in proportion to their frequencies. */
    for i = 1 to k do
        frac = (min(rangehigh, high(b_i)) - max(rangelow, low(b_i)) + 1) / (high(b_i) - low(b_i) + 1);
        freq(b_i) = max(freq(b_i) + (α * esterr * frac * freq(b_i)) / est, 0);
    end for
end UpdateFreq;
```

$$frac = \frac{\min(rangehigh, high(b_i)) - \max(rangelow, low(b_i)) + 1}{high(b_i) - low(b_i) + 1};$$

$$freq(b_i) = \max\left(freq(b_i) + \frac{\alpha * esterr * frac * freq(b_i)}{est},\ 0\right);$$

Figure 21: Algorithm for updating bucket frequencies in one-dimensional ST-histograms

error in estimation may be due to incorrect frequencies in any of the buckets that overlap the selection range. This is different from traditional histograms in which, if the histogram has been built using a full scan of data and has not been degraded in accuracy by database updates, the estimation error can result only from the leftmost or rightmost buckets, and only if they partially overlap the selection range. In traditional histograms, buckets that are totally contained in the selection range do not contribute to the error.

The change in frequency of any ST-histogram bucket should depend on how much it contributes to the estimation error. We use the heuristic that buckets with higher frequencies contribute more to the estimation error than buckets with lower frequencies. Specifically, we assign the "blame" for the error to the buckets used for estimation *in proportion to their current frequencies*.

We multiply the estimation error by a *damping factor*, α, between 0 and 1 to make sure that bucket frequencies are not modified too much in response to errors, as this may lead to over-sensitive or unstable histograms. This damping factor is analogous to the *learning rate* used in many artificial intelligence algorithms.

Figure 21 presents the algorithm for updating the bucket frequencies of an ST-histogram, h, in response to a range selection, $[rangelow, rangehigh]$, with actual result size act. This

algorithm is used for both on-line and off-line refinement.

The algorithm first determines the histogram buckets that overlap the selection range, whether they partially overlap the range or are totally contained in it, and the estimated result size. The query optimizer usually obtains this information during query optimization, so we can save some effort by retaining this information for subsequently refining bucket frequencies.

Next, the algorithm computes the absolute estimation error, denoted by $esterr$. The error formula distinguishes between over-estimation, indicated by a negative error and requiring the bucket frequencies to be lowered, and under-estimation, indicated by a positive error and requiring the bucket frequencies to be raised.

As mentioned earlier, the blame for this error is assigned to histogram buckets in proportion to the frequencies that they contribute to the estimated result size. We assume that each bucket contains all possible values in the range that it represents, and we approximate all frequencies in a bucket by their average (i.e., we make the *continuous values* and *uniform frequencies* assumptions [PIHS96]). Under these assumptions, the contribution of a histogram bucket to the estimated result size is equal to its frequency times the fraction of the bucket overlapping the selection range. This fraction is the length of the interval where the bucket overlaps the selection range divided by the length of the interval represented by the bucket ($frac$ in Figure 21). This fraction can be less than 1 only for the leftmost and rightmost buckets used in estimation, b_1 and b_k, and it always equals 1 for all other buckets used in estimation.

To distribute the error among the histogram buckets used for estimation in proportion to frequency, each bucket is assigned a portion of the absolute estimation error, $esterr$, equal to its contribution to the result size, $frac * freq(b_i)$, divided by the total result size, est, damped by a damping factor, α. We experimentally demonstrate in Section 3.5 that the refinement process is robust across a wide range of values for α, and we recommend using values of α in the range 0.5 to 1.

3.3.3 Restructuring

Refining bucket frequencies is not enough to get an accurate histogram. The frequencies in a bucket are approximated by their average. If there is a large variation in frequency within a bucket, the average frequency is a poor approximation of the individual frequencies, no matter how accurate this average frequency is. Specifically, high-frequency values will be contained in high-frequency buckets, but they may be grouped with low-frequency values in these buckets. Thus, in addition to refining the bucket frequencies, we must also *restructure* the buckets, i.e., move the bucket boundaries. The goal of restructuring is to get a better partitioning that avoids grouping high-frequency and low-frequency values in the same buckets. Ideally, we would like to make high-frequency buckets as narrow as possible. In the limit, this approach separates out high-frequency values in singleton buckets of their own, a common objective in histogram construction [PIHS96].

We assume that the frequency refinement procedure described above is accurate, so high-frequency values will be contained in buckets that have a high frequency. However, these buckets may also contain low-frequency values. Thus, we choose buckets that currently have high frequencies and *split* them into several buckets. Splitting causes the separation of high-frequency and low-frequency values into different buckets, and the frequency refinement procedure later adjusts the frequencies of these new buckets. In order to ensure that the number of buckets assigned to the ST-histogram does not increase due to splitting, we need a mechanism to reclaim buckets as well.

Our mechanism for reclaiming buckets is to use a *merging* step that groups a run of consecutive buckets with similar frequencies into one bucket. Thus, our approach is to *restructure* the histogram periodically by merging buckets and using the buckets thus freed to split high frequency buckets.

Restructuring may be triggered using a variety of heuristics. We choose a simple scheme in which the restructuring process is invoked after every R selections that use the histogram. The parameter R is called the *restructuring interval*.

To merge buckets with similar frequencies, we first have to decide how to quantify "similar

frequencies". We assume that two bucket frequencies are similar if the difference between them is less than m percent of the number of tuples in the relation, T, where m is a parameter that we call the *merge threshold*. Our experiments show that $m \leq 1\%$ is typically a suitable choice. We use a greedy strategy to form runs of adjacent buckets with similar frequencies, and we collapse each run into a single bucket. We repeat this step until no adjacent buckets or runs of buckets have similar frequencies, so no further merging is possible.

We also need to decide which "high frequency" buckets to split. We choose to split the s percent of the histogram buckets with the highest frequencies, where s is a parameter that we call the *split threshold*. In our experiments, we use $s = 10\%$. Our splitting heuristic distributes the reclaimed buckets among the high frequency buckets in proportion to their frequency. The higher the frequency of a bucket, the more extra buckets it gets.

Figure 22 presents the algorithm for restructuring an ST-histogram, h, of B buckets on a relation with T tuples. The first step in histogram restructuring is to greedily find runs of consecutive buckets with similar frequencies to merge. The algorithm repeatedly finds the pair of adjacent runs of buckets such that the maximum difference in frequency between a bucket in the first run and a bucket in the second run is the minimum over all pairs of adjacent runs. The two runs are merged into one if this difference is less than the threshold $m * T$, and we stop looking for runs to merge if it is not.

This process results in a number of runs of several consecutive buckets. Each run is replaced with one bucket spanning the entire range of the run, and with a frequency equal to the total frequency of all the buckets in the run. This frees a number of buckets to allocate to the high frequency histogram buckets during splitting.

Splitting starts by identifying the s percent of the buckets that have the highest frequencies and are not singleton buckets (buckets that span only one value). We avoid splitting buckets that have been chosen for merging since their selection indicates that they have similar frequencies to their neighbors. The extra buckets freed by merging are distributed among the buckets being split in proportion to their frequencies. A bucket being split, b_i, gets $freq(b_i)/totalfreq$ of the extra buckets, where $totalfreq$ is the total frequency of the buckets being split. To split a bucket, it is replaced with itself plus the extra buckets assigned to it. These new buckets are

algorithm RestructureHist
Inputs: h
Outputs: restructured h

begin
/* Find buckets with similar frequencies to merge. */
Initialize B runs of buckets such that each run contains one histogram bucket;
repeat
For every two consecutive runs of buckets, find the maximum difference in frequency between a bucket in the first run and a bucket in the second run;
Let $mindiff$ be the minimum of all these maximum differences;
if ($mindiff \leq m * T$) **then**
Merge the two runs of buckets corresponding to $mindiff$ into one run;
end if
until ($mindiff > m * T$);

/* Assign the extra buckets freed by merging to the high frequency buckets. */
$k = s * B$;
Find the set, $\{b_1, b_2, \ldots, b_k\}$, of buckets with the k highest frequencies that were not chosen to be merged with other buckets in the merging step and are not singleton buckets;
Assign the buckets freed by merging to the k buckets in this set in proportion to their current frequencies;

/* Construct the restructured histogram by merging and splitting. */
Merge each previously formed run of buckets into one bucket spanning the range represented by all the buckets in the run and having a frequency equal to the sum of their frequencies;
Split the k buckets chosen for splitting, giving each one the number of extra buckets assigned to it earlier. For each bucket being split, the new buckets are evenly spaced in the range spanned by the old bucket and the frequency of the old bucket is equally distributed among them;
end RestructureHist;

Figure 22: Algorithm for restructuring one-dimensional ST-histograms

evenly spaced in the range spanned by the old bucket, and the frequency of the old bucket is evenly distributed among them.

Splitting and merging are used in [GMP97] to redistribute histogram buckets in the context of maintaining approximate equi-depth and compressed histograms. The algorithm in [GMP97] merges *pairs* of buckets whose total frequency is less than a threshold, whereas our algorithm merges *runs* of buckets based on the differences in their frequency. Our algorithm assigns the freed buckets to the buckets being split in proportion to the frequencies of the latter, whereas the algorithm in [GMP97] merges only one pair of buckets at a time and can, therefore, split only one bucket into two. A key difference between the two approaches is that in [GMP97], a sample of the tuples of the relation is continuously maintained (called the *backing sample*), and buckets are split at their approximate medians computed from this sample. On the other hand, our approach does not examine the data at any point, so we do not have information similar to that represented in the backing sample of [GMP97]. Hence, our restructuring algorithm splits buckets at evenly spaced intervals, without using any information about the data distribution within a bucket.

Figure 23 presents an example of histogram restructuring. In this example, the merge threshold is such that restructuring algorithm merges buckets if the difference between their frequencies is within 3. The algorithm identifies two runs of buckets to be merged, buckets 1 and 2, and buckets 4 to 6. Merging these runs frees three buckets to assign to high frequency histogram buckets. The split threshold is such that we split the two buckets with the highest frequencies, buckets 8 and 10. Assigning the extra buckets to these two buckets in proportion to frequency means that bucket 8 gets two extra buckets and bucket 10 gets one extra bucket.

Splitting may unnecessarily separate values with similar, low frequencies into different buckets. Such runs of buckets with similar low frequencies would be merged during subsequent restructuring. Notice that splitting distorts the frequency of a bucket by distributing it among the new buckets. This means that the histogram may lose some of its accuracy by restructuring. This accuracy is restored when the bucket frequencies are refined through subsequent feedback.

In summary, our model is as follows: The frequency refinement process is applied to the

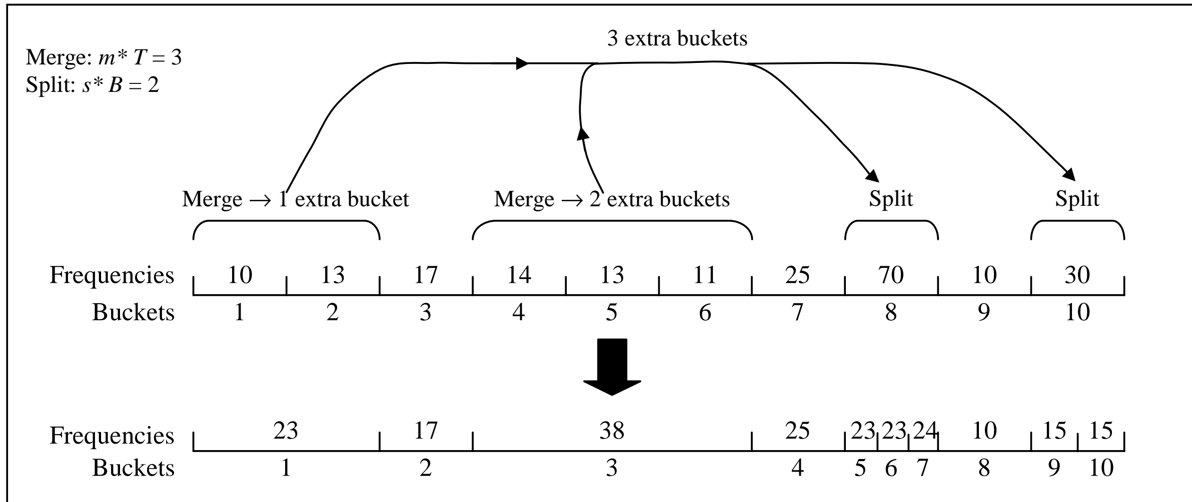


Figure 23: Example of ST-histogram restructuring

histogram after every query, and the refined frequency information is periodically used to restructure the histogram. Restructuring may reduce accuracy by distributing frequencies among buckets during splitting but frequency refinement restores, and hopefully increases, histogram accuracy.

3.4 Multi-dimensional ST-histograms

In this section, we present multi-dimensional (i.e., multi-attribute) ST-histograms. Our goal is to build histograms representing the joint distribution of multiple attributes of a single relation. These histograms will be used to estimate the result size of conjunctive range selections on these attributes, and are refined based on feedback from these selections. Using accurate one-dimensional histograms for all the attributes is not enough, because they do not reflect the *correlation* between attributes. In this section, we discuss the special considerations for multi-dimensional histograms.

Working in multiple dimensions raises the issue of how to partition the multi-dimensional space into histogram buckets. We addressed this issue in Chapter 2 in the context of building histograms for spatial data, and our solution there was to use a quadtree partitioning of the space. For ST-histograms, we adopt a different solution.

The effectiveness of ST-histograms stems from their ability to pinpoint the buckets contributing to the estimation error and "learn" the data distribution. The partitioning we choose must efficiently support this learning process. It must also be a partitioning that is easy to construct and maintain, because we want the cost of ST-histograms to remain as low as possible. To achieve these objectives, we use a *grid partitioning* of the multi-dimensional space. Each dimension of the space is partitioned into a number of partitions. The partitions of a dimension may vary in size, but the partitioning of the space is always fully described by the partitioning of the dimensions.

We choose a grid partitioning due to its simplicity and low cost, even though it does not offer as much flexibility in grouping values into buckets as other partitionings such as, for example, the MHIST-p histogram partitioning [PI97]. The simplicity of a grid partitioning allows our histograms to have more buckets for a given amount of memory. It is easier for ST-histograms to infer the data distribution from feedback information when working with a simple high-resolution representation of the distribution than it is when working with a complex low-resolution representation. Furthermore, we doubt that the simple feedback information used for refinement can be used to glean enough information about the data distribution to justify a more complex partitioning.

Each dimension, i, of an n-dimensional ST-histogram is partitioned into B_i partitions. B_i does not necessarily equal B_j for $i \neq j$. The partitioning of the space is described by n arrays, one per dimension, which we call the *scales* [NHS84]. Each array element of the scales represents the range of one partition, $[low, high]$. In addition to the scales, a multi-dimensional ST-histogram has an n-dimensional matrix representing the grid cell frequencies, which we call the *frequency matrix*. Figure 24 presents an example of a 5×5 two-dimensional ST-histogram and a range selection that uses it.

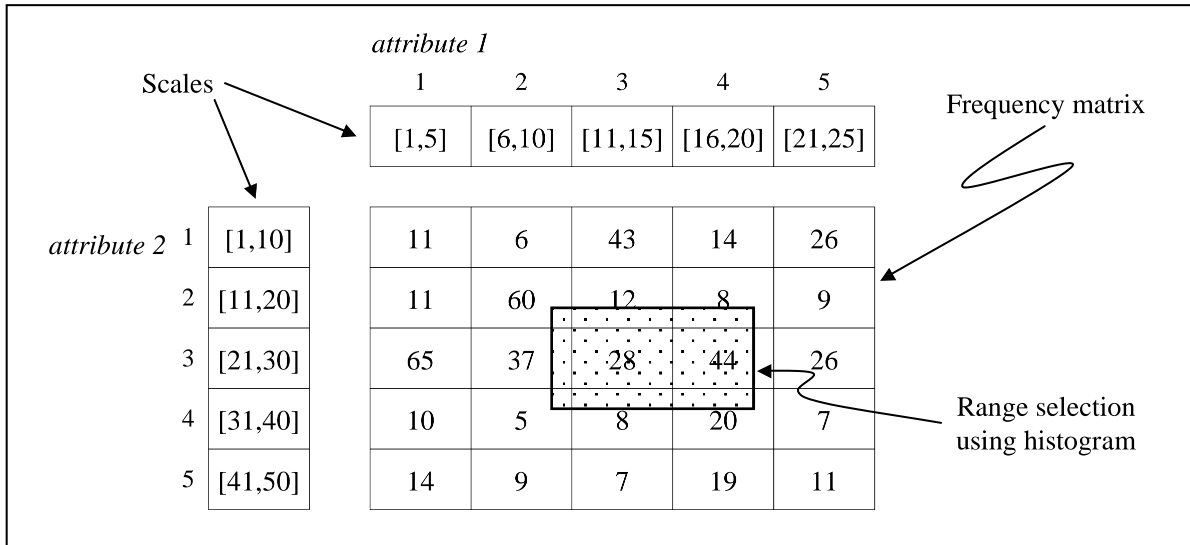


Figure 24: A two-dimensional ST-histogram and a range selection using it

3.4.1 Initial Histogram

To build an ST-histogram on attributes $a_1, a_2, \ldots, a_n$, we can start by assuming complete uniformity and independence, or we can use existing one-dimensional histograms but assume independence of the attributes as the starting point.

If we start with the uniformity and independence assumption, we need to know the minimum and maximum values of each attribute a_i, min_i and max_i. We also need to specify the number of partitions for each dimension, $B_1, B_2, \ldots, B_n$. Each dimension, i, is partitioned into B_i equally spaced partitions, and the T tuples of the relation are evenly distributed among all the buckets of the frequency matrix. This technique is an extension of one-dimensional ST-histogram initialization.

Another way of building multi-dimensional ST-histograms is to start with traditional one-dimensional histograms on all the multi-dimensional histogram attributes. Such one-dimensional histograms, if they are available, provide a better starting point than assuming uniformity and independence. In this case, we initialize the scales by partitioning the space along the bucket boundaries of the one-dimensional histograms, and we initialize the frequency matrix using the bucket frequencies of the one-dimensional histograms and assuming that the

attributes are independent. Under the independence assumption, the initial frequency of a cell of the frequency matrix is given by:

$$freq\,[j_1, j_2, \ldots, j_n] = \frac{1}{T^{n-1}} \prod_{i=1}^{n} freq_i[j_i]$$

where $freq_i[j_i]$ is the frequency of bucket j_i of the histogram for dimension i.

3.4.2 Refining Bucket Frequencies

The algorithm for refining bucket frequencies in the multi-dimensional case is identical to the one-dimensional algorithm presented in Figure 21, except for two differences. First, finding the histogram buckets that overlap a selection range now requires examining a multi-dimensional structure. Second, a bucket is now a multi-dimensional cell in the frequency matrix, so the fraction of a bucket overlapping the selection range is equal to the *volume of the region* where the bucket overlaps the selection range divided by volume of the region represented by the whole bucket (Figure 24).

3.4.3 Restructuring

Periodic restructuring is needed only for multi-dimensional ST-histograms initialized assuming uniformity and independence. ST-histograms initialized using traditional one-dimensional histograms do not need to be periodically restructured, assuming that the one-dimensional histograms are accurate. This is based on the assumption that the partitioning of an accurate traditional one-dimensional histogram built by looking at the data is more accurate when used for multi-dimensional ST-histograms than a partitioning built by splitting and merging.

As in the one-dimensional case, restructuring in the multi-dimensional case is based on merging buckets with similar frequencies and splitting high frequency buckets. The required parameters are also the same, namely the restructuring interval, R, the merge threshold, m, and the split threshold, s. Restructuring changes the partitioning of the multi-dimensional space *one dimension at a time*. The dimensions are processed in any order, and the partition boundaries of each dimension are modified independent of other dimensions. The algorithm for restructuring

one dimension of the multi-dimensional ST-histogram is similar to the algorithm in Figure 22. However, merging and splitting in multiple dimensions present some additional problems.

For an n-dimensional ST-histogram, every partition of the scales in any dimension identifies an $(n-1)$-dimensional "slice" of the grid (e.g., a row or a column in a two-dimensional histogram). Thus, merging two partitions of the scales requires merging two slices of the frequency matrix, each containing several buckets. Every bucket from the first slice is merged with the corresponding bucket from the second slice. To decide whether or not to merge two slices, we find the maximum difference in frequency between any two corresponding buckets that would be merged if these two slices are merged. We merge the two slices only if this difference is within $m * T$ tuples. We use this method to identify not just *pairs*, but rather *runs* of partitions of the scales to merge.

The high frequency partitions of any dimension are split by assigning them the extra partitions freed by merging *in the same dimension*. Thus, restructuring does not change the number of partitions in any dimension. To decide which partitions in a particular dimension to split and how many extra partitions each one of these partitions that are being split gets, we use the *marginal frequency distribution* along this dimension. The marginal frequency of a partition is the total frequency of all buckets in the slice of the frequency matrix that it identifies. Thus, the marginal frequency of partition j_i in dimension i is given by:

$$f_i(j_i) = \sum_{j_1=1}^{B_1} \cdots \sum_{j_{i-1}=1}^{B_{i-1}} \sum_{j_{i+1}=1}^{B_{i+1}} \cdots \sum_{j_n=1}^{B_n} freq\left[j_1, j_2, \ldots, j_{i-1}, j_i, j_{i+1}, \ldots, j_n\right]$$

As in the one-dimensional case, we split the s percent of the partitions in any dimension with the highest marginal frequencies, and we assign them the extra partitions in proportion to their current marginal frequencies.

Figure 25 demonstrates restructuring the histogram in Figure 24 along the vertical dimension (attribute 2). In this example, the merge threshold is such that we merge two partitions if the maximum difference in frequency between buckets in their slices that would be merged is within 5. This condition leads us to merge partitions 4 and 5. The split threshold is such that we split one partition along the vertical dimension. We compute the marginal frequency distribution along the vertical dimension and identify the partition with the maximum marginal

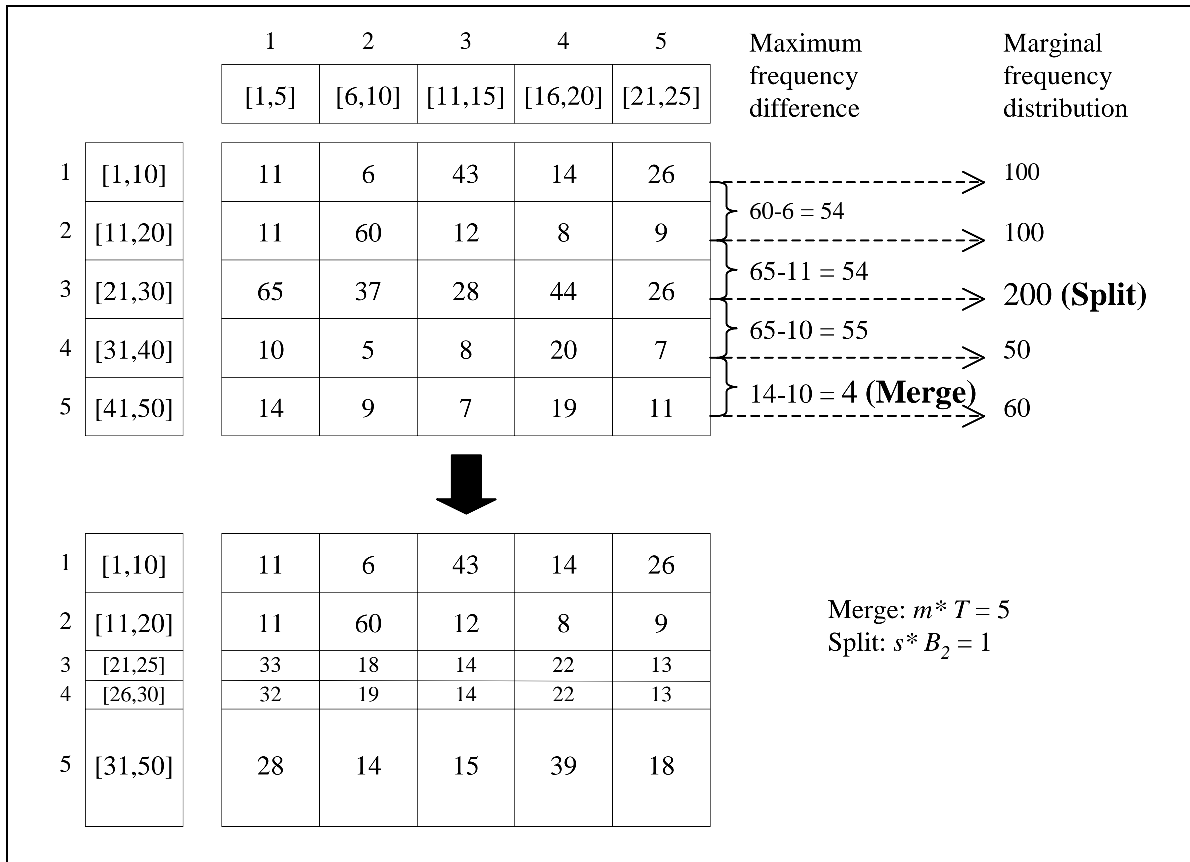


Figure 25: Restructuring the vertical dimension of a two-dimensional ST-histogram

frequency, partition 3. Merging and splitting (with some provisions for rounding) result in the shown histogram.

3.5 Experimental Evaluation

In this section, we present an experimental evaluation of the accuracy of ST-histograms. We demonstrate that ST-histograms are accurate for a wide range of data distributions, for workloads with different access patterns, and for a wide range of values for the histogram parameters. We also show that ST-histograms adapt well to database updates, and that histogram refinement converges rapidly. Finally, we study the effectiveness of ST-histograms over real data sets.

3.5.1 Experimental Setup

Data Sets

We use both synthetic and real data sets in our experiments. We study the accuracy and sensitivity of ST-histograms using synthetic integer data sets of one to three dimensions. The one-dimensional data sets each have 100,000 tuples and the multi-dimensional data sets have 500,000 tuples. Each dimension in a data set has V distinct values drawn randomly (according to a uniform distribution) from the domain of integers from 1 to 1000. $V =$ 200, 100, and 10, for one, two, and three dimensions, respectively. For multi-dimensional data sets, the number of distinct values and the domains from which these values are chosen are identical for all dimensions, and the value sets of all dimensions are generated independently.

Frequencies are generated according to the *Zipfian* distribution [Zipf49] with parameter $z =$ 0, 0.5, 1, 2, and 3. The parameter z controls the skew of the distribution. The greater the value of z, the more skewed the distribution, with $z = 0$ representing a uniform distribution (no skew). For one-dimensional data sets, the frequencies are assigned at random to the values. For multi-dimensional data sets, the frequencies are assigned at random to *combinations* of values using the technique proposed in [PI97], namely assigning the generated frequencies to randomly chosen cells in the joint frequency distribution matrix.

We also study the accuracy of ST-histograms on two real data sets: point of sale data for a drugstore chain and income data from the U.S. Census Bureau. These real data sets are described in detail in Section 3.5.9.

Query Workloads

We use workloads consisting of random *range selection* queries in one or more dimensions. Each workload consists of 2000 independent selection queries.

Most experiments use workloads with *random queries*, in which the end points in each dimension of each selection range are independently generated from a uniform distribution over the entire domain of the attribute being queried.

Some experiments use workloads with *locality of reference*. The values used for the end

points of the selection ranges in these workloads are generated from *piecewise uniform* distributions in which there is an 80% probability of choosing a value from a *locality range* that is 20% of the domain of the attribute being queried. The locality ranges for the different dimensions are independently chosen at random according to a uniform distribution.

Histograms

Unless otherwise stated, we use 100, 50, and 15 buckets per dimension for ST-histograms in one, two, and three dimensions, respectively. For multi-dimensional ST-histograms, we use the same number of buckets in all dimensions, resulting in two- and three-dimensional histograms with a total of 2500 and 3375 buckets, respectively. The one-, two-, and three-dimensional ST-histograms occupy 1.2, 10.5, and 13.5 KBytes of memory, respectively.

Our traditional histograms of choice are MaxDiff(V,A) histograms for one dimension, and MHIST-2 MaxDiff(V,A) histograms for multiple dimensions. These histograms were recommended in [PIHS96] and [PI97] for their accuracy and ease of construction. We compare the accuracy of ST-histograms to traditional histograms of these types occupying the same amount of memory.

We allocate more memory to histograms that most previous works, such as [PIHS96], [PI97], and [MVW98]. We believe that current trends in memory technology justify this increased memory allocation. We also demonstrate in Section 3.5.7 that our techniques are effective across a wide range of memory allocations.

Note that the cost of building and maintaining traditional histograms is a function of the size of the relation (or the size of the sample used to build the histogram). In contrast, the cost of ST-histograms is independent of the data size and depends on the size of the query workload used for refinement.

Refinement Parameters

Unless otherwise stated, the parameters we use for frequency refinement and histogram restructuring (Sections 3.3.2 and 3.3.3) are as follows:

- Damping factor: $\alpha = 0.5$ for one-dimensional histograms, and $\alpha = 1$ for multi-dimensional histograms.
- Restructuring interval: $R = 200$ queries.
- Merge threshold: $m = 0.025\%$.
- Split threshold: $s = 10\%$.

Error Metric

We use the *average absolute estimation error* to measure the accuracy of the different histograms. The absolute error in estimating the result size of one query, q, is defined as:

$$e_q = |\text{estimated result size} - \text{actual result size}|$$

For a workload of M queries, the average absolute estimation error is defined as:

$$E = \frac{\sum_{i=1}^{M} e_i}{M}$$

Our workloads all have $M = 2000$ queries. To put the absolute error values in perspective, we present these values as a *percent of the size (i.e., number of tuples) of the relation being queried.*

The other error metric commonly used to measure estimation accuracy is the *relative* estimation error defined as:

$$e = \frac{|\text{estimated result size} - \text{actual result size}|}{\text{actual result size}}$$

We used this error metric in our experiments on spatial data in Chapter 2. However, this error metric may sometimes be misleading when evaluating histograms. A disadvantage of the relative error is that it can be very large for queries with small result sizes because the denominator of the error formula is small. If a histogram is inaccurate for a few queries with small result sizes in a query workload, the average relative estimation error for the entire workload will be large no matter how accurate the histogram is for the remaining queries in the workload.

Moreover, these queries with small result sizes that distort the average estimation error are queries for which estimation accuracy may not be important, since the number of tuples involved

is small. It is more important for query optimization purposes to have accurate selectivity estimates for queries with *large* result sizes. Some commercial database systems (e.g., IBM DB/2) even use histograms that isolate high-frequency values in singleton buckets to ensure selectivity estimation accuracy for queries with large result sizes in which these values appear. The goal of ST-histogram refinement is also to accurately capture high-frequency values. ST-histograms may not accurately capture all low-frequency values, especially for highly skewed data distributions, but this does not limit their usefulness for query optimization since they are still accurate for queries with large result sizes.

An important issue in evaluating ST-histograms is choosing a workload with respect to which the accuracy of a histogram is measured. An ST-histogram is refined – or "trained" – using feedback from a particular *refinement workload*. It would be unrealistic to replay the exact workload after refinement and use it to evaluate the histogram. Instead, we evaluate the histogram using a different but *statistically similar* query workload. We call this the *test workload*. We cannot expect the workload issued before refinement to be repeated exactly after refinement, but we can reasonably expect a workload with similar statistical characteristics. The ultimate test of accuracy is whether the ST-histogram is accurate for the test workload.

Unless otherwise stated, our experiments use *off-line* histogram refinement. To summarize, our steps for verifying the effectiveness of ST-histograms for some particular data set are as follows:

1. Initialize an ST-histogram for the data set.
2. Issue the query workload that will be used to refine the histogram (the *refinement workload*) and generate a *workload log*.
3. Refine the histogram off-line based on the generated workload log.
4. After refinement, issue the refinement workload again and compute the estimation error. Verify that the error after refinement is less than the error before refinement.
5. Issue a different query workload in which the queries have the same distribution as the

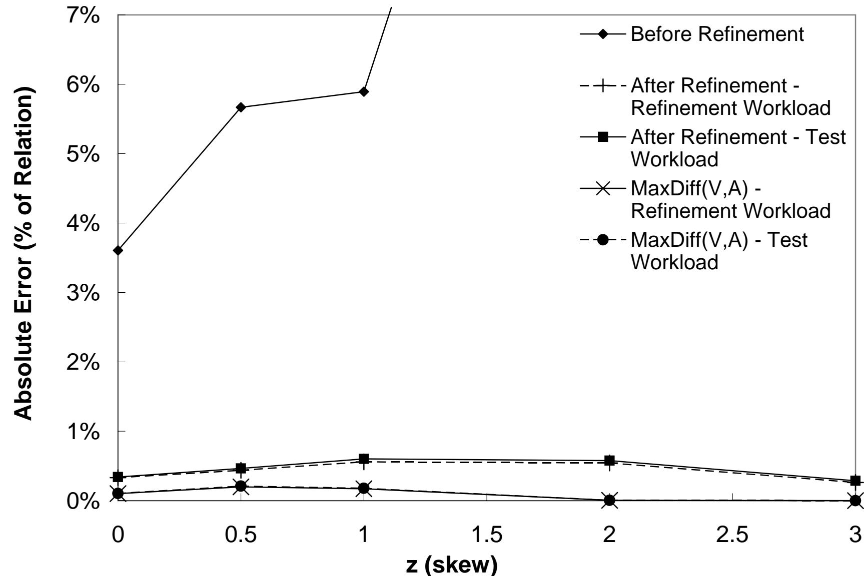


Figure 26: Accuracy of one-dimensional ST-histograms

refinement workload. This is the *test workload*. The measure of accuracy of the ST-histogram is the average absolute estimation error for this test workload.

3.5.2 Accuracy of One-dimensional ST-histograms

In this section, we study the effectiveness of one-dimensional ST-histograms for a wide range of data skew (z) using workloads with random queries and the procedure outlined above.

Figure 26 presents estimation errors for workloads with random queries on one-dimensional data sets with varying z. Each point in the figure is the average error for the 2000 queries constituting a particular workload. For each data set, the figure presents the average absolute estimation error for the refinement workload using the initial ST-histogram constructed assuming uniformity. Assuming uniformity is the most common approach when there is no information about the data distribution. Thus, our first question is whether ST-histograms can do better than this assumption (i.e., whether an ST-histogram *after* refinement will be more accurate than it was *before* refinement).

For each data set, Figure 26 presents the estimation error for the refinement workload when this workload is issued again after histogram refinement. Histogram refinement clearly improves accuracy. Thus, ST-histograms are more accurate than assuming uniformity.

The figure also shows, for each data set, the estimation error using the refined ST-histogram for a test workload with random queries that is different from the refinement workload. ST-histograms are as accurate for the test workloads as they are for the refinement workloads when these workloads are issued again after refinement.

The figure also presents the estimation errors for the refinement and test workloads using traditional MaxDiff(V,A) histograms occupying the same amount of memory as the ST-histograms. The figure shows that MaxDiff(V,A) histograms are more accurate than ST-histograms. This is expected because MaxDiff(V,A) histograms are built based on the true distribution determined by examining the data. ST-histograms have a lower cost than MaxDiff(V,A) histograms. This lower cost comes at the expense of lower accuracy. However, an important point to note is that the estimation errors using refined ST-histograms are close to the errors using MaxDiff(V,A) histograms, and are small enough for query optimization purposes.

Throughout our experiments, we observe that ST-histograms and traditional histograms have the same accuracy for the test workloads as they do for the refinement workloads when they are issued again after refinement. Furthermore, we are ultimately interested in the accuracy of histograms on the test workloads. Thus, in all remaining experiments, we only present the accuracy of refined ST-histograms and traditional histograms on the test workloads.

3.5.3 Accuracy of Multi-Dimensional ST-histograms

In this section, we show that multi-dimensional ST-histograms initialized using traditional one-dimensional histograms are more accurate than using these one-dimensional histograms and assuming independence. We also compare the performance of these ST-histograms and MHIST-2 histograms. We demonstrate that such ST-histograms are more accurate than MHIST-2 histograms for low values of z (i.e., low correlation). This is an important result because it indicates that ST-histograms are better than MHIST-2 histograms in both cost and accuracy for data distributions with low correlation. We only present the results of experiments with ST-histograms initialized using traditional one-dimensional histograms, not the less accurate ST-histograms initialized assuming uniformity and independence.

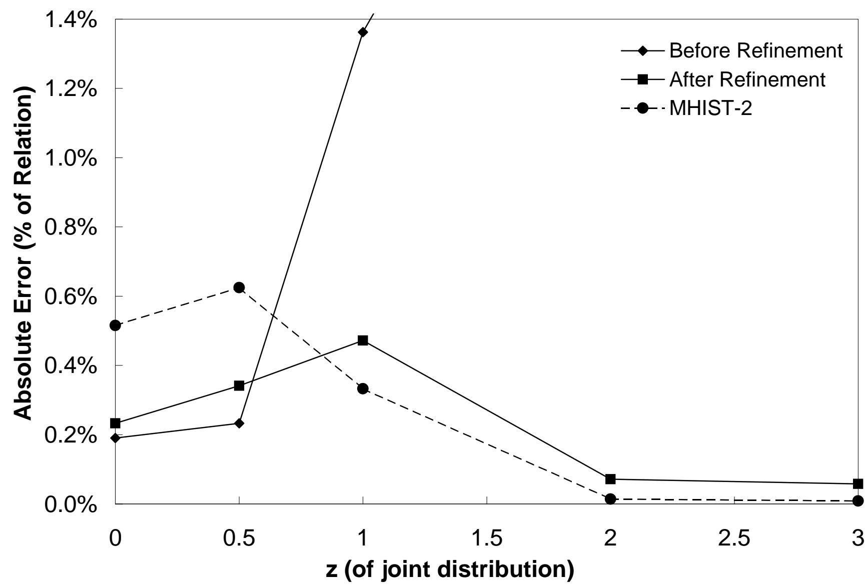


Figure 27: Accuracy of two-dimensional ST-histograms starting with MaxDiff(V,A)

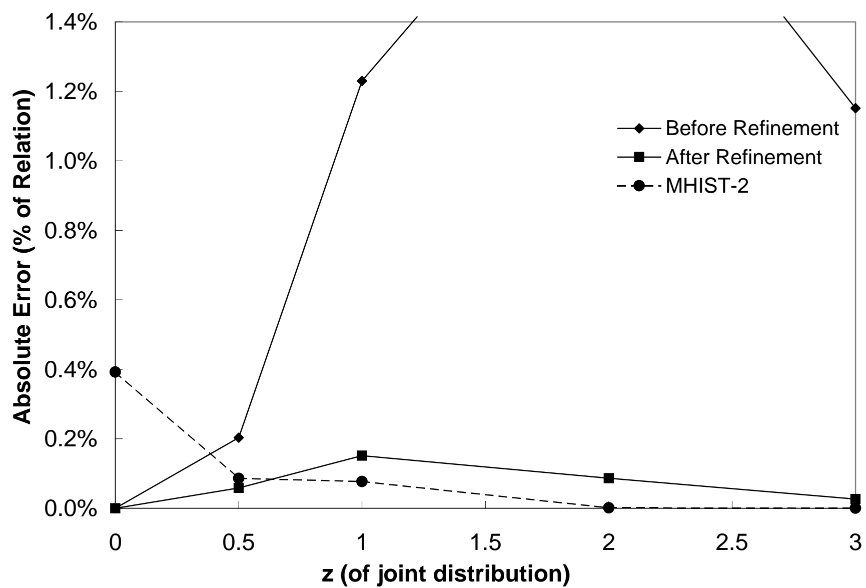


Figure 28: Accuracy of three-dimensional ST-histograms starting with MaxDiff(V,A)

Figures 27 and 28 present the results of using multi-dimensional ST-histograms initialized using MaxDiff(V,A) histograms and assuming independence for workloads with random queries on two- and three-dimensional data sets with varying z. The figures show the estimation errors of ST-histograms on the refinement workloads when they are issued before histogram refinement. This is equivalent to using the one-dimensional MaxDiff(V,A) histograms used to initialize the ST-histograms and assuming independence. The figures also show the estimation errors of ST-histograms on the test workloads issued *after* histogram refinement. Furthermore, the figures show the estimation errors of traditional multi-dimensional MHIST-2 MaxDiff(V,A) histograms on the test workloads.

The refined ST-histograms are more accurate than assuming independence, and the benefit of using them (i.e., the reduction in error) increases as z increases. ST-histograms are not as accurate as MHIST-2 histograms for high z, although they are close. This is because inferring joint data distributions based on simple feedback information becomes increasingly difficult with increasing correlation. As expected, MHIST-2 histograms are very accurate for high z [PI97], but we must bear in mind that the cost of building multi-dimensional MHIST-2 histograms is higher than the cost of building one-dimensional MaxDiff(V,A) histograms. Furthermore, this cost increases with increasing dimensionality.

Note that ST-histograms are more accurate than MHIST-2 histograms for low values of z. This is because MHIST-2 histograms use a complex partitioning of the space (as compared to ST-histograms). Representing this complex partitioning requires MHIST-2 histograms to have complex buckets that consume more memory than ST-histogram buckets. Consequently, ST-histograms have more buckets than MHIST-2 histograms occupying the same amount of memory. For low values of z, the complex partitioning used by MHIST-2 histograms does not increase accuracy because the joint data distribution is close to uniform so any partitioning works well. On the other hand, the large number of buckets in ST-histograms allows them to represent the distribution at a finer granularity leading to higher accuracy. For such data, ST-histograms are both cheaper and more accurate than traditional histograms.

Table 4 presents the estimation errors of ST-histograms initialized using different types of

z	Equi-width		Equi-depth		MaxDiff(V,A)	
	Before	After	Before	After	Before	After
0	0.17%	0.28%	0.24%	0.31%	0.19%	0.21%
0.5	0.23%	0.25%	0.32%	0.41%	0.23%	0.32%
1	1.35%	0.40%	1.39%	0.40%	1.36%	0.45%
2	3.22%	2.59%	2.91%	0.24%	2.61%	0.06%
3	2.83%	1.37%	2.10%	0.17%	2.03%	0.06%

Table 4: Two-dimensional ST-histograms initialized using different types of traditional one-dimensional histograms

traditional one-dimensional histograms. The table presents the errors for workloads with random queries on two-dimensional data sets with varying z. The errors are shown for the refinement workloads when they are issued before histogram refinement, and for the test workloads when they are issued after histogram refinement. All one-dimensional histograms have 50 buckets. In addition to MaxDiff(V,A) histograms, the table presents the errors when we start with equi-width histograms, which are the simplest type of histograms, and when we start with equi-depth histograms, which are used by many commercial database systems. The table shows that ST-histograms are effective for all these types of one-dimensional histograms.

Next, we study the *sensitivity* of ST-histograms to variations in their operating conditions. For all these experiments, we use synthetic data sets with $z = 1$ and ST-histograms initialized assuming uniformity and independence.

3.5.4 Effect of Locality of Reference in the Query Workload

An interesting issue is studying the performance of ST-histograms on workloads with locality of reference in accessing the data. Locality of reference is a fundamental concept underlying database accesses, so one would expect real life workloads to have such locality. Moreover, workloads with purely random queries provide feedback information about the entire distribution, while workloads with locality of reference provide most of their feedback about a small part of the distribution. We would like to know how effective this type of feedback is for histogram refinement. In this section, we demonstrate that ST-histograms perform well for workloads with locality of reference. We also demonstrate that histogram refinement adapts to

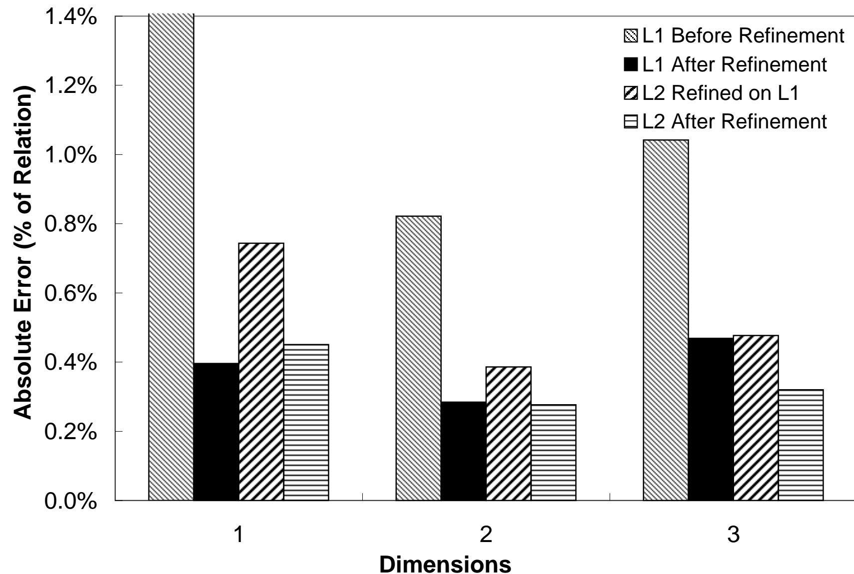


Figure 29: Accuracy of ST-histograms for workloads with locality of reference

changes in the locality range of the workload.

Figure 29 presents the estimation errors for workloads with an 80%-20% locality for one- to three-dimensional data sets with $z = 1$. For each dimensionality, the first bar represents the error for the refinement workload when it is issued before ST-histogram refinement (which is equivalent to assuming uniformity and independence). The refinement workload has a particular locality range, $L1$. The second bar shows the error for a test workload with the same locality range as the refinement workload, $L1$, when it is issued after histogram refinement. As expected, histogram refinement improves estimation accuracy.

Next, we keep the refined histogram and change the locality of reference of the query workload. We issue a new workload with a *different* locality range, $L2$. The third bar for each dimensionality in Figure 29 shows the error for this workload using the histogram refined using a workload with locality range $L1$. The refined ST-histogram is not as accurate for the workload with locality range $L2$ as it is for the workload with locality range $L1$. However, it is better than assuming uniformity and independence (the unrefined ST-histogram). This means that histogram refinement was still able to infer some information from the 20% of the queries in the refinement workload that lie outside the locality range, $L1$.

Next, we refine the histogram on the new workload with locality range $L2$, and we issue a

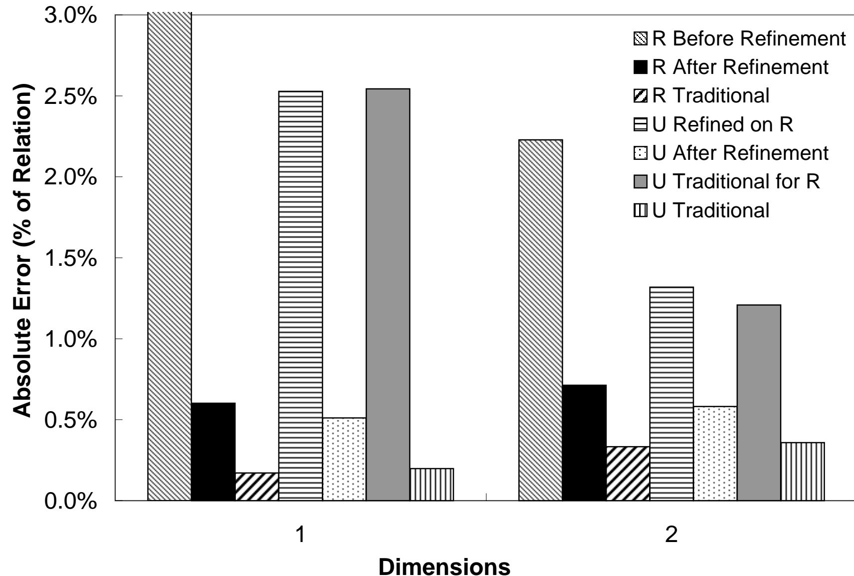


Figure 30: Accuracy of ST-histograms in adapting to database updates

different test workload that has the same locality range. The fourth bar for each dimensionality in the figure shows the error for this test workload. We see that refinement is able to restore histogram accuracy. Thus, ST-histograms adapt well to changes in the locality of reference of the workload.

3.5.5 Adapting to Database Updates

In this section, we demonstrate that although ST-histograms do not examine the data, the feedback mechanism enables these histograms to adapt to updates in the underlying relation.

Figure 30 presents the estimation errors for one- and two-dimensional data sets with $z = 1$ using workloads with random queries. The first bar for each dimensionality presents the estimation error for the refinement workload on the original relation, R, using the ST-histogram before refinement. The second bar shows the error for the test workload using the refined ST-histogram, and the third bar shows the error for this workload using a traditional MaxDiff(V,A) or MHIST-2 histogram. As expected, refined ST-histograms are more accurate than unrefined ST-histograms but less accurate than traditional histograms.

We update the relation by deleting a random 25% of its tuples and inserting an equal number of tuples following a Zipfian distribution with $z = 1$. We denote this updated relation by U.

Dims	1		2		3	
z	No Restruct.	Restruct.	No Restruct.	Restruct.	No Restruct.	Restruct.
0	0.41%	0.34%	0.44%	0.46%	0.95%	0.95%
0.5	0.46%	0.46%	0.44%	0.44%	0.97%	0.91%
1	0.83%	0.60%	0.72%	0.71%	1.49%	1.19%
2	1.12%	0.58%	1.08%	0.32%	1.66%	1.40%
3	0.50%	0.29%	1.33%	0.27%	1.51%	0.34%

Table 5: Accuracy of ST-histograms with and without periodic restructuring

We retain the traditional and ST-histograms built for R and issue a test workload on U. The fourth bar for each dimensionality in Figure 30 shows the error for this workload using the ST-histogram refined for R. The histogram is not as accurate for U as it is for R, but it is still more accurate than assuming uniformity and independence.

Next, we refine the ST-histogram using the workload issued on the updated relation, U, and we issue a different test workload on this relation. The fifth bar for each dimensionality in the figure shows the error for this test workload using the refined ST-histogram. Refinement restores the accuracy of the ST-histogram and adapts it to the updates in the relation.

The last two bars for each dimensionality present the estimation errors for the final test workload issued on U using traditional histograms built for R and U, respectively. Traditional histograms do not adapt to database updates. The accuracy of the histogram on the original relation when used for the updated relation is much lower than the accuracy of a histogram built on the updated relation.

Most database systems deal with this inaccuracy by periodically rebuilding histograms in response to database updates, incurring the full cost of building a histogram with each periodic rebuild. This makes a low-cost alternative such as ST-histograms even more desirable.

3.5.6 Refinement Parameters

In this section, we investigate the effect of the refinement parameters: R, m, and s for restructuring, and α for updating bucket frequencies.

Table 5 presents the estimation errors for test workloads consisting of random queries using

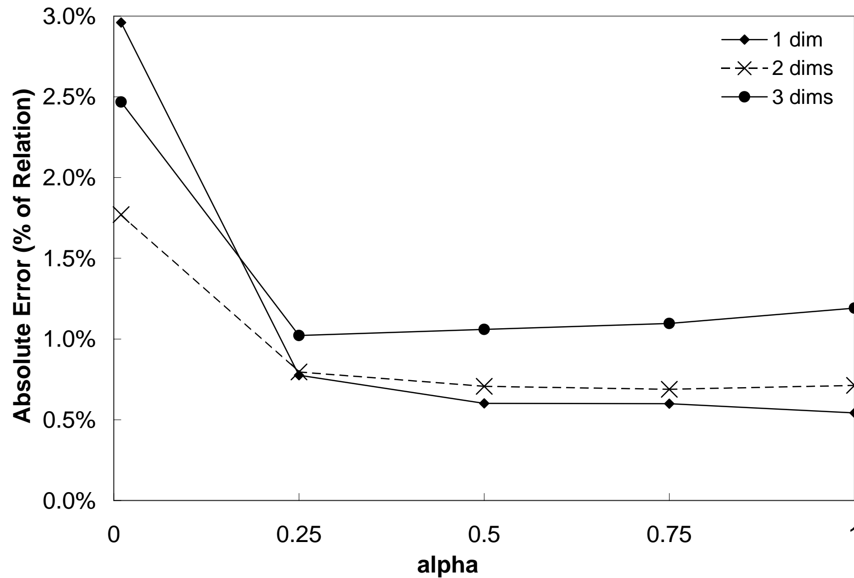


Figure 31: Effect of α on the accuracy of ST-histograms

ST-histograms that have been refined off-line using other statistically similar refinement workloads for one- to three-dimensional data sets with varying z. For each data set, the error is presented if the histogram is not restructured during refinement, and if it is restructured with $R = 200$, $m = 0.025\%$, and $s = 10\%$. Restructuring has no benefit for low z because there are no high-frequency values that restructuring can isolate. However, as z increases the need for restructuring becomes evident.

Figure 31 presents the estimation errors for test workloads consisting of random queries on one- to three-dimensional data sets with $z = 1$ using ST-histograms that have been refined off-line using other statistically similar refinement workloads with $\alpha = 0.01$ to 1. The estimation errors are relatively flat for a wide range of α. Thus, the benefit of trying to find the optimal α is minimal, and we recommend using a fixed α or varying it according to a simple algorithm. For our experiments we use $\alpha = 0.5$ for the one-dimensional case and $\alpha = 1$ for the multi-dimensional case.

3.5.7 Effect of Available Memory

Figures 32–34 present the estimation errors for test workloads consisting of random queries on data sets with $z = 1$ in one- to three-dimensions using ST-histograms that have been refined

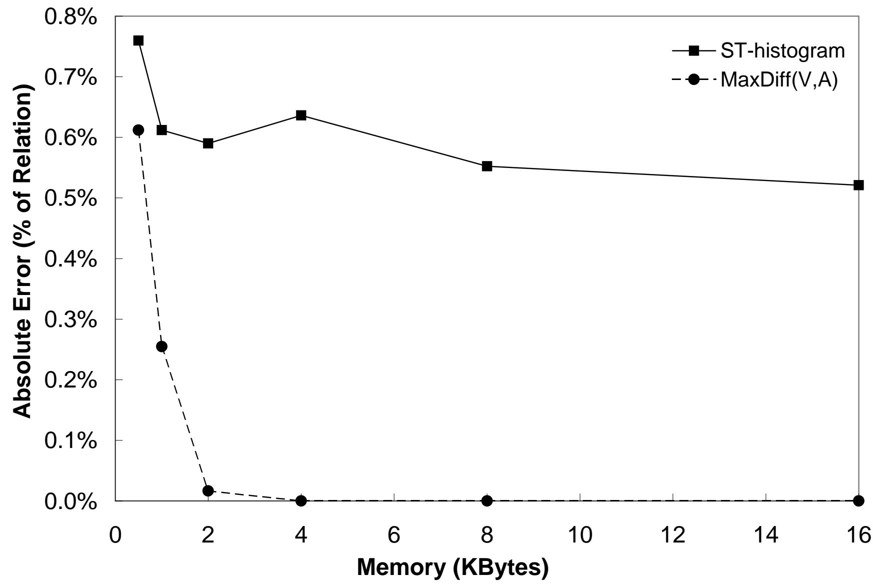


Figure 32: Effect of available memory on the accuracy of one-dimensional ST-histograms

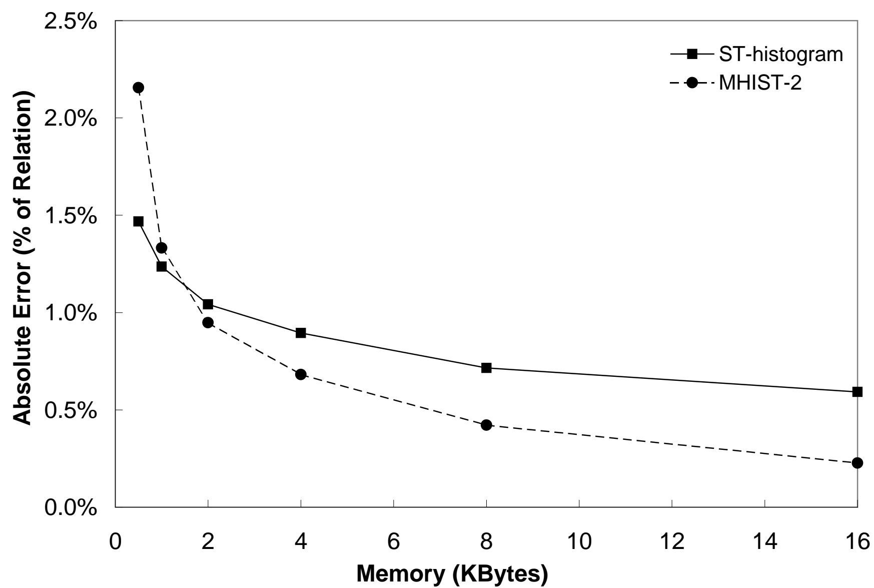


Figure 33: Effect of available memory on the accuracy of two-dimensional ST-histograms

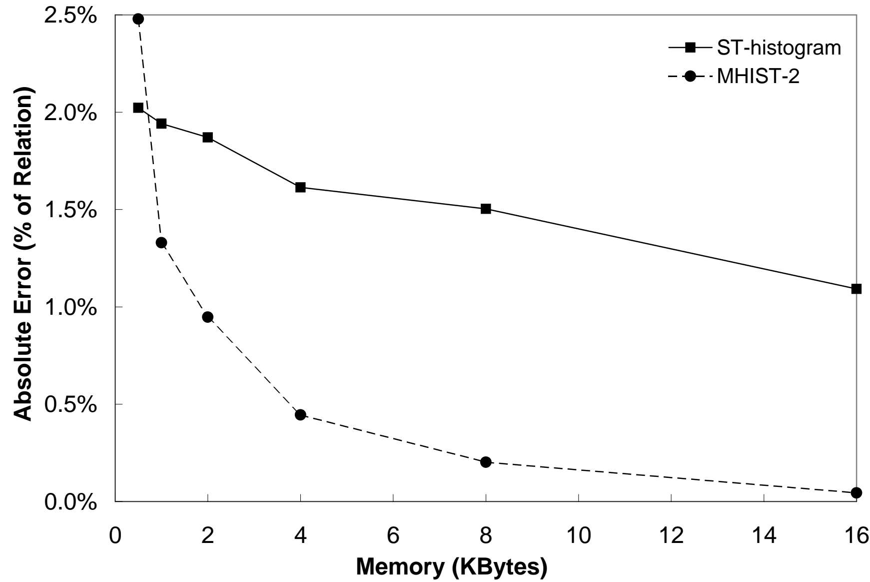


Figure 34: Effect of available memory on the accuracy of three-dimensional ST-histograms

using other refinement workloads, and using traditional histograms occupying the same amount of memory as the ST-histograms. The errors are presented for histograms using 0.5 to 16 KBytes of memory.

Increasing the memory allocated to both ST-histograms and traditional histograms results in a decrease in estimation error. However, the error drops more for traditional histograms, and the drop is faster. Traditional histograms are able to make better use of extra available memory than ST-histograms. This is because ST-histograms are refined based on simple, coarse-grained feedback information. This simple feedback information does not allow ST-histogram refinement to accurately partition the space into more buckets that cover smaller areas. For example, refining the frequency of a bucket based on a query that covers many other buckets is not as accurate as refining its frequency based on a query that is totally contained in this bucket. Also, splitting or merging a bucket that has been used by a small number of queries and, therefore, reflects a limited amount of feedback information is not as accurate as splitting or merging a bucket that has been used by many queries.

Traditional histograms do not suffer from this drawback. Since they examine the data, they can gather as much information as needed about the data distribution to accurately partition the space into as many buckets as required. These histograms, therefore, make better use of extra

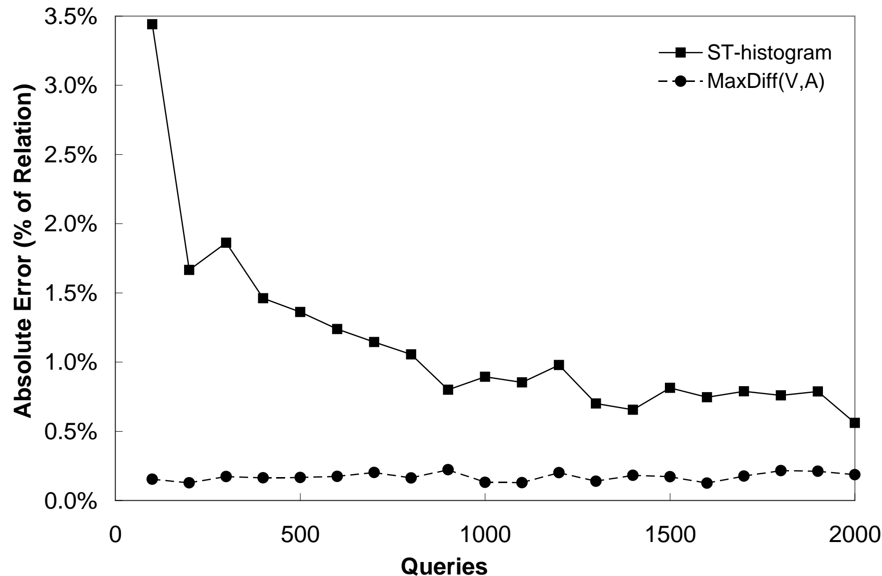


Figure 35: On-line refinement of one-dimensional ST-histograms

available memory.

For the multi-dimensional case, ST-histograms are better than MHIST-2 histograms when the amount of available memory is small. An MHIST-2 histogram has fewer buckets than an ST-histogram using the same amount of memory because MHIST-2 histograms use a more complex partitioning of the space that requires more memory to represent. For low memory, the MHIST-2 histogram has too few buckets to cleverly partition the space. The ST-histogram makes better use of what little memory is available by partitioning the space into more buckets and capturing the distribution at a finer granularity.

3.5.8 On-line Refinement and Convergence

The histogram refinement process and the resulting refined ST-histograms are the same whether we use on-line or off-line refinement. Thus, even though our experiments use off-line refinement, our conclusions are valid for on-line refinement as well. In this section, we switch to on-line refinement to study convergence. Convergence is important for both off-line and on-line refinement, but it is more important and easier to observe for on-line refinement.

We issue a workload consisting of random range queries one query at a time, recording the estimation error and incrementally refining the ST-histogram after each query. To study the

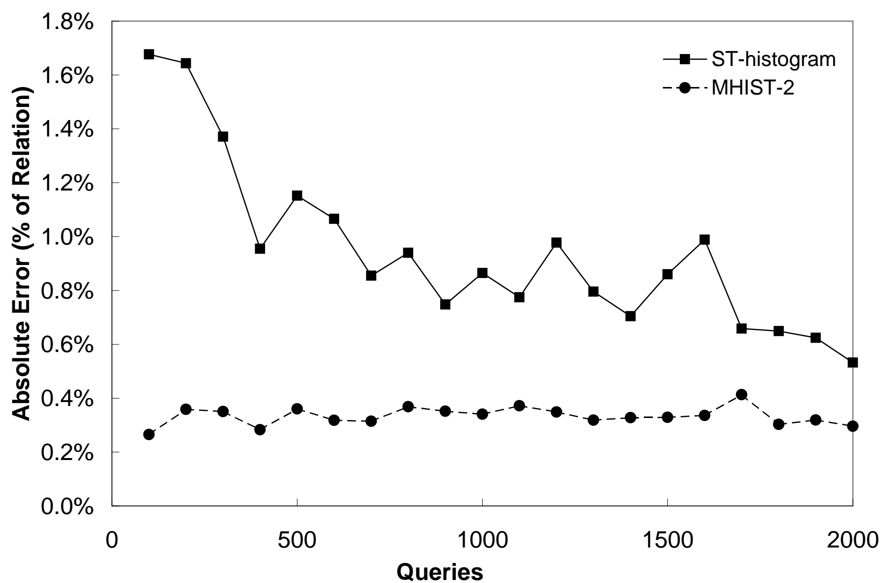


Figure 36: On-line refinement of two-dimensional ST-histograms

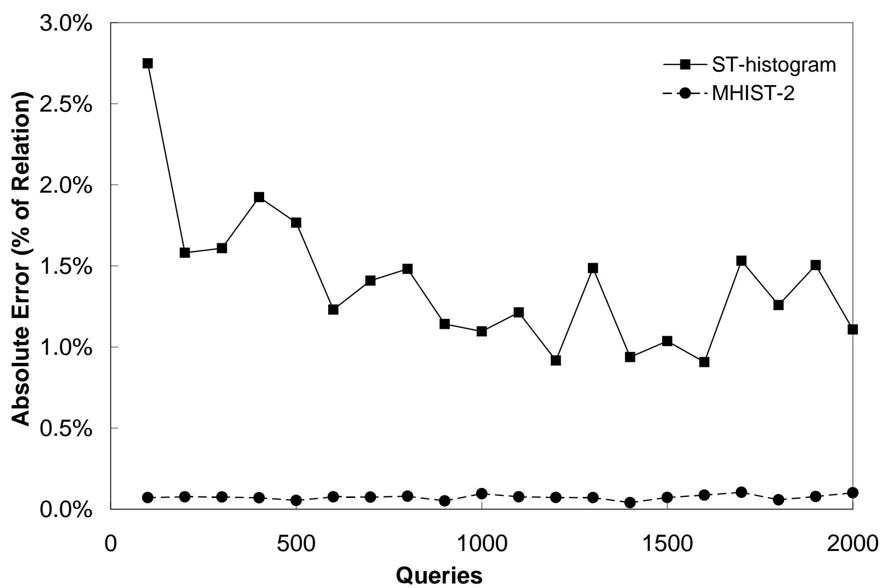


Figure 37: On-line refinement of three-dimensional ST-histograms

convergence of ST-histogram refinement, we compute the average error of every 100 queries. Figures 35–37 present these errors for ST-histograms and traditional histograms for one- to three-dimensional data sets with $z = 1$.

The figures show that ST-histogram refinement converges fairly rapidly. These results support our argument that ST-histograms have a low cost. The simple histogram refinement process has to be performed only a small number of times before the histogram becomes sufficiently accurate.

3.5.9 Accuracy of ST-histograms on Real Data

In this section, we study the accuracy of two- and three-dimensional ST-histograms initialized using traditional one-dimensional MaxDiff(V,A) histograms on two real data sets.

The first real data set consists of point of sale data for the five days from 1/12/1997 to 1/16/1997 from several stores belonging to Pharma, a major drugstore chain in Japan [HKMY98][1]. This data set has 1,056,892 rows. Each row contains the time, quantity, and retail price per item of a particular purchase. Our two-dimensional queries are random range queries on (time, quantity), and our three-dimensional queries are random range queries on (time, quantity, retail price). The two-dimensional queries have an average result size of 362.8, and the three-dimensional queries have an average result size of 2.6.

Figures 38 and 39 present the estimation errors for two- and three-dimensional test workloads consisting of random queries on this data. The figures present the estimation errors using ST-histograms that have been initialized using traditional MaxDiff(V,A) histograms on the attributes and refined using refinement workloads consisting of random range queries. The figures also present the estimation errors using traditional MHIST-2 histograms occupying the same amount of memory as the ST-histograms. The errors are presented for histograms occupying 0.5 to 16 KBytes of memory.

[1] Thanks to Raghu Ramakrishnan for providing us with this data, and to the Pharma company for allowing us to use it.

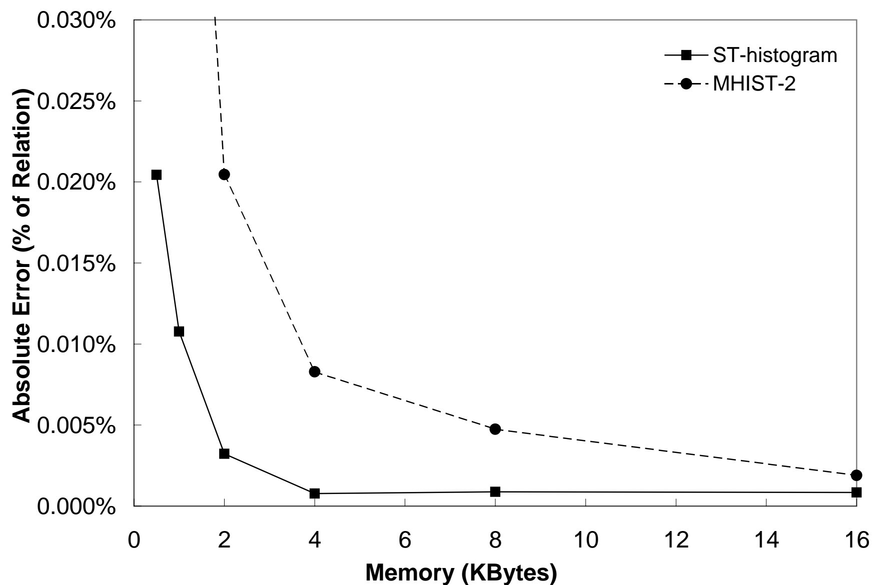


Figure 38: Two-dimensional ST-histograms starting with MaxDiff(V,A) – Point of sale data

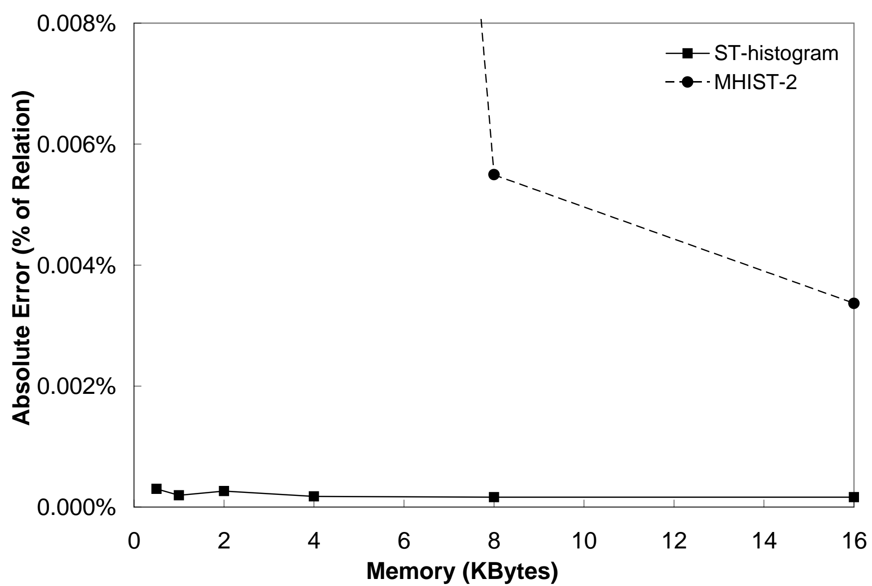


Figure 39: Three-dimensional ST-histograms starting with MaxDiff(V,A) – Point of sale data

The figures show that, for this data set and these workloads, ST-histograms are more accurate than traditional MHIST-2 histograms. This is for the same reason that caused multi-dimensional ST-histograms to be more accurate than traditional MHIST-2 histograms for data with low correlation in Section 3.5.3. The data distributions for this data set are not very skewed and the correlation between attributes is reasonably low. However, the distributions have many distinct values, 8,311 for the two-dimensional case and 280,372 for the three-dimensional case. For such distributions, the complex partitioning used by MHIST-2 histograms does not increase accuracy because the joint data distribution is close to uniform so all partitionings are almost equivalent. On the other hand, the larger number of buckets available to ST-histograms for a given amount of memory allows these histograms to represent the distribution at a finer granularity, leading to higher estimation accuracy. The figures also show that more memory results in more accuracy for both traditional and ST-histograms.

The second real data set that we use for our experiments consists of income data from the U.S. Census Bureau obtained from the University of California, Irvine, Knowledge Discovery in Databases Archive [Census]. This data set has 299,285 rows. Each row contains the wages per hour, age, and capital gains earned for a particular individual in a particular year. Our two-dimensional queries are random range queries on (wages, age), and our three-dimensional queries are random range queries on (wages, age, capital gains). The two-dimensional queries have an average result size of 1548.3, and the three-dimensional queries have an average result size of 22.7. The two-dimensional data distribution has 6,965 distinct values and the three-dimensional data distribution has 10,216 distinct values.

Figures 40 and 41 present the same information for this data set as presented in Figures 38 and 39. For most memory allocations, ST-histograms are more accurate than MHIST-2 histograms for the same reason noted above. However, Figures 40 and 41 also illustrate that more memory does not necessarily result in more accuracy for ST-histograms. This effect is quite pronounced for the two-dimensional case when increasing the available memory from 8 KBytes to 16 KBytes.

As discussed in Section 3.5.7, ST-histograms cannot always partition the space accurately

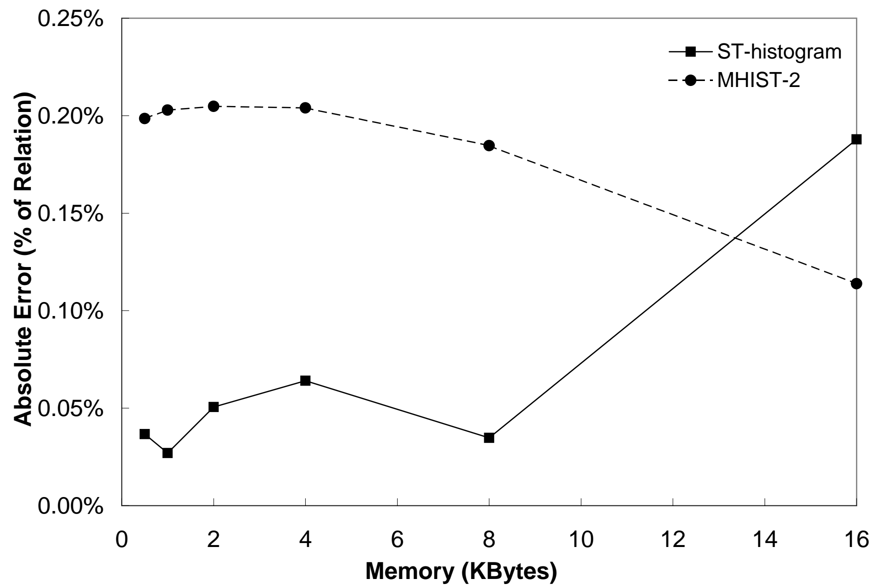


Figure 40: Two-dimensional ST-histograms starting with MaxDiff(V,A) – Census data

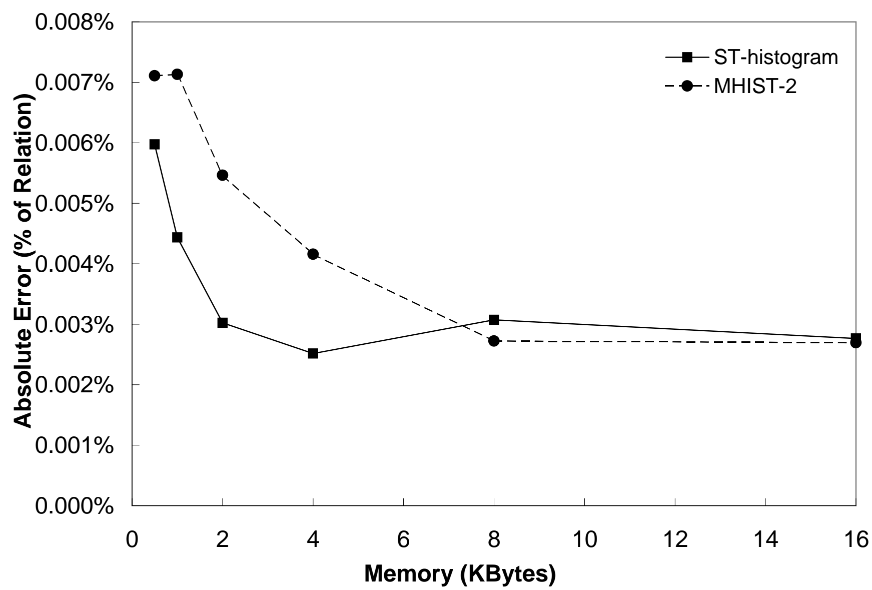


Figure 41: Three-dimensional ST-histograms starting with MaxDiff(V,A) – Census data

into more buckets that cover smaller areas using the simple feedback information used for histogram refinement. Figures 40 and 41 show a case in which having more buckets actually *reduces* accuracy. ST-histograms with more buckets may have to refine bucket frequencies based on queries that span a large number of buckets. Allocating the "blame" for estimation errors to individual buckets in this case can be inaccurate. ST-histograms with more buckets may also have to make restructuring decisions after only a few queries have "hit" each bucket. In this case, the information reflected in the buckets may not be an accurate basis for restructuring.

To avoid such problems, we can employ some heuristics when refining ST-histograms. For example, we can refine bucket frequencies only for queries that span $\leq k$ buckets, for some threshold k. Also, we can trigger histogram restructuring only when the average number of queries per bucket is $> l$, for some threshold l. These issues present interesting areas of future work.

3.6 Conclusions

ST-histograms are a novel way of building histograms at a low cost based on feedback from the query execution engine *without* looking at the data. Multi-dimensional ST-histograms are particularly attractive, since they provide a low-cost alternative to traditional multi-dimensional structures proposed in the literature that are often prohibitively expensive for large databases (true of many data warehouses). Multi-dimensional ST-histograms are almost as accurate as traditional MHIST-2 histograms for a wide range of data distributions, and sometimes more accurate, while costing much less to build and maintain.

It is possible to take advantage of the low cost of ST-histograms but still get the extra accuracy achievable with traditional histograms when it is needed. We can start by initializing an ST-histogram and refining it for a preset number of queries representative of the workload on the database. If this training sequence fails to reduce the selectivity estimation error using the histogram to an acceptably low level, we can consider building a more accurate but more expensive traditional histogram.

Chapter 4

Estimating the Selectivity of XML Path Expressions for Internet-scale Applications

4.1 Introduction

In this chapter, we move our discussion from querying data that is stored in a database system in a traditional setting to querying *Internet data*. We present selectivity estimation techniques that can help a query optimizer optimize queries posed against data on the Internet.

Data on the Internet is increasingly presented in the *extensible markup language (XML)* format [BPS98]. The standardized, simple, self-describing nature of this format opens the door for novel Internet-scale applications that integrate and query XML data from numerous sources all over the Internet.

An example of such an Internet-scale application is the Niagara Internet query system [NDM+01]. Niagara allows a user to pose queries against "all the XML documents on the Internet," using an integrated search engine to find XML documents that are relevant to a query based on the path expressions that appear in the query. Another Internet-scale query processor is Xyleme [Xyl01], which aims to build an indexed, queryable XML repository of all the information on the World Wide Web.

As an example of the queries that can be handled by an Internet-scale query processor, consider the following query expressed in the XQuery language [CFR+01]:

```
FOR $n_au IN document("*")//novel/author
    $p_au IN document("*")//play/author
WHERE $n_au/text()=$p_au/text()
```

```
RETURN $n_au
```

This is a join query that asks for writers who have written both a play and a novel. `document("*")` means that the query should be executed against all the known XML data on the Internet. In Niagara, the integrated search engine would find all XML documents relevant to this query by finding all documents that contain the path `novel/author` or `play/author`.

Optimizing a query like this one requires estimating the selectivities of the path expressions it contains. For example, the query optimizer may need to estimate the selectivities of the path expressions `novel/author` and `play/author` (i.e., the number of `author` elements reachable by each path) to choose the more selective path expression as the outer data stream of the join. Path expressions are essential to querying XML, so estimating the selectivity of these path expressions is essential to XML query optimization.

Estimating the selectivity of XML path expressions requires having database statistics that contain information about the *structure* of the XML data. These statistics must fit in a small amount of memory because the query optimizer may consult them many times in the course of optimizing a query. The goal here is not to conserve memory, but rather to conserve *query optimization time*. The statistics must be small enough to be processed efficiently in the short time available for query optimization. To quote examples from commercial relational database systems, Microsoft SQL Server 2000 uses 200 buckets per column for its histograms [DS00], and IBM DB/2 uses 20 quantiles and 10 frequent values per column [DB2].

The structure of XML data with its many degrees of freedom can be much richer than the value distribution of a column in a relational database. As such, we should assume that the statistics used for estimating the selectivity of XML path expressions will be allocated more memory than histograms in relational database systems. However, these statistics must still be small enough to allow for fast selectivity estimation.

Ensuring that statistics do not consume too much memory is particularly important for the Internet-scale applications that we focus on. It may be safe to assume that the structure of a single typical XML document can be captured in a small amount of memory. However, when considering Internet-scale applications that handle large amounts of XML data with widely

varying structure, we cannot assume that the overall structure of all the XML data handled can be captured in a small amount of memory. The statistics used for selectivity estimation therefore have to be *summarized* so that they fit in the available memory.

We present techniques for building database statistics that capture the structure of complex XML data in a small amount of memory and for using these statistics to estimate the selectivity of XML path expressions. Our focus is on *simple path expressions*.

A simple path expression is a sequence of *tags* that represents a navigation through the tree structure of the XML data starting anywhere in the tree (not necessarily at the root). In abbreviated XPath syntax [CD99], which we use throughout this chapter, a simple path expression of length n is expressed as $//t_1/t_2/\cdots/t_n$. This path expression specifies finding a tag t_1 anywhere in the document, and nested in it finding a tag t_2, and so on until we find a tag t_n. Our focus is estimating the number of t_n elements reached by this navigation. Note that we assume an unordered model of XML and that we do not consider navigations based on IDREF attributes.

We propose two techniques for capturing the structure of XML data for estimating the selectivity of path expressions. The first technique is to construct a tree representing the structure of the XML data, which we call the *path tree*. We then summarize this tree to ensure that it fits in the available memory by deleting low-frequency nodes and replacing them with nodes representing the information contained in the deleted nodes at a coarser granularity. The second technique is to store all paths in the data up to a certain length and their frequency of occurrence in a table of paths that we call the *Markov table*. We summarize the Markov table by deleting low-frequency paths, and we combine the paths of limited length in the Markov table to estimate the selectivity of longer paths.

The rest of this chapter is organized as follows. Section 4.2 presents an overview of related work. Section 4.3 describes path trees. Section 4.4 describes Markov tables. Section 4.5 presents an experimental evaluation of the proposed techniques. Section 4.6 contains concluding remarks.

4.2 Related Work

Estimating the selectivity of XML path expressions is related to estimating the selectivity of substring predicates, which has been addressed in several papers [KVI96, JNS99, JKNS99, CKKM00]. These papers all address the issue of estimating the frequency of occurrence of one or more substrings in a string database. They all use variants of the *pruned suffix tree* data structure. A suffix tree is a trie that stores all strings in the database and all their suffixes. A pruned suffix tree is a suffix tree in which nodes corresponding to low frequency strings are pruned so that the tree fits in the available memory. [KVI96] and [JNS99] address the problem of estimating the selectivity of single substrings. [JKNS99] addresses the problem of estimating the selectivity of multiple substring predicates combined in conjunctive queries. [CKKM00] addresses the problem of estimating the selectivity of multiple substring predicates combined in arbitrary boolean queries.

The techniques in [JNS99] and [CKKM00] are the basis for techniques developed in [CJK$^+$01] for estimating the selectivity of *twig queries*. Like path expressions, twig queries specify a navigation through the structure of XML documents or other tree-structured data. Twig queries are more general than the simple path expressions that we consider. They can specify navigations based on *branching* path expressions, and they can specify specific *data values* that must be found at the ends of the path expressions (rather than navigating based only on the structure of the XML data).

The techniques developed in [CJK$^+$01] are the best previously known techniques that can be applied to the problem of estimating the selectivity of XML path expressions, even though they solve a more general problem. We restricted the data structures developed in [CJK$^+$01] for twig queries to the simpler problem of estimating the selectivity of path expressions by storing the minimum amount of information needed for selectivity estimation and no information about data values or path branching. We found that our data structures were able to give more accurate selectivity estimates for this simpler but very common case. The techniques developed in [CJK$^+$01] are described in more detail in Section 4.5.1.

Estimating the selectivity of XML path expressions requires summarizing the structure of

the XML data. This can be done using *DataGuides* [GW97] or *T-indexes* [MS99]. These methods construct graphs that represent structural summaries of the data. These graphs can be used to estimate the selectivity of path expressions by annotating their nodes with the selectivity of the paths that they represent. For tree-structured XML data, the graphs constructed by both these methods are identical, and they have the same structure as our un-summarized path trees. However, the problem of summarizing these graphs if they do not fit in the available memory is not addressed in [GW97] or [MS99]. On the other hand, the techniques we develop can summarize path trees to fit in any amount of memory.

Summarizing DataGuides is addressed in [GW99]. In that paper, a DataGuide is summarized by finding common labels in the paths represented in it or similar sets of objects reachable by these paths. This summary is not suitable for selectivity estimation because the frequency of occurrence of the paths does not play a role in summarization. Also, no summarization is possible if all the labels are distinct, and no guarantees can be made on the size of the summarized data guide.

The Lore semi-structured database system can store and query XML data [MAG+97, GMW99]. The Lore query optimizer estimates the selectivity of XML path expressions by storing selectivity information for all paths in the database of length up to k, where k is a tuning parameter [MW99]. This approach is valid but not scalable because the memory required for storing all paths of length up to k grows as the database grows. The Markov table approach that we propose also builds a table of all paths in the database up to a certain length, but this table is summarized if it overgrows the available memory. We also provide a method of combining the paths of limited length stored in the Markov table to obtain accurate selectivity estimates for longer path expressions.

Navigating through a graph-structured database using path expressions has been considered in the context of object-oriented databases. Some cost models for query optimization in object-oriented database systems do depend on the selectivity of path expressions, but no methods for accurately estimating this selectivity are proposed. See, for example, [ODE95], [GGT95], [GGT96], or [BF97].

Subsequent to publishing the work presented in this chapter in [AAN01], Polyzotis and

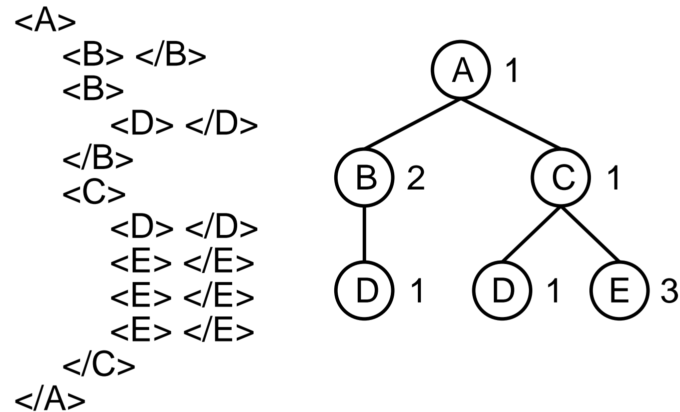


Figure 42: An XML document and its path tree

Garofalakis proposed a graph synopsis structure, which they call XSKETCH, that can be used to estimate the selectivity of XML path expressions [PG02]. This structure exploits notions of localized graph stability to approximate the path and branching distributions of XML data. It is shown to be more accurate than the Markov tables that we propose for some data sets given the same amount of storage.

4.3 Path Trees

In this section, we describe *path trees* that represent the structure of XML data and present techniques for summarizing these trees. We also describe using summarized path trees for selectivity estimation.

A path tree is a tree representing the structure of an XML document. Every node in the path tree represents a path starting from the root of the XML document. The root node of the path tree represents the root element of the document. A path tree node has a child node for every distinct tag name of an XML element directly nested in any of the elements reachable by the path it represents. Every path tree node is labeled with the *tag name* of the elements reachable by the path it represents and with the number of such elements, which we call the *frequency* of the node. Figure 42 presents an XML document and its path tree.

The path tree of an XML document can be constructed in one scan of the document using

an event-based XML parser [SAX] and a stack that grows to the maximum nesting depth of the XML elements in the document. To construct a path tree for multiple XML documents, we create an artificial root node for all the XML data so that we can view it as a single tree.

A path tree contains all the information required for selectivity estimation. To estimate the selectivity of a query path expression, we scan the tree looking for all nodes with tags that match the first tag of the path expression. From every such node, we try to navigate down the tree following child pointers and matching tags in the path expression with tags in the path tree. This will lead us to a set of path tree nodes which all correspond to the query path expression. The selectivity of the query path expression is the total frequency of these nodes. This algorithm is $O(n)$, where n is the number of nodes in the path tree.

The problem with a path tree is that it may be larger than the available memory, so we need to summarize it. To summarize a path tree, we delete the nodes with the lowest frequencies from anywhere in the tree. We try to preserve some of the information represented in the deleted nodes at a coarser granularity by adding nodes to the path tree that represent groups of deleted nodes. These nodes that we add have the special tag name "$*$", which stands for "any tag name", so we call them *$*$-nodes* ("star nodes"). Next, we present four methods of summarizing path trees that differ in the amount of information they try to preserve in the $*$-nodes.

4.3.1 Sibling-$*$

In the first method for summarizing path trees, which we call *sibling-$*$*, we repeatedly choose the path tree node with the lowest frequency and mark it for deletion. This can be done in $O(n \log n)$ using a priority queue. Marking a node for deletion does not reduce the size of the path tree. However, when we mark a node, `A`, for deletion, we check its siblings to see if they contain a node, `B`, that is either a $*$-node or a regular node that has been marked for deletion. If we find such a node, nodes `A` and `B` are *coalesced* into one $*$-node, which reduces the size of the path tree.

Each $*$-node represents multiple sibling nodes deleted from the path tree. The parent of a $*$-node is the parent of the deleted nodes it represents, and the children of these deleted nodes

become children of the $*$-node. When a node, A, is coalesced with a $*$-node, the children of A become children of the $*$-node. Some of the children of A may have the same tag name as children of the $*$-node. Since these children with common tag names are now siblings, they are coalesced together, further reducing the size of the path tree. Coalescing the children of coalesced nodes is repeated recursively if needed.

Since both $*$-nodes and nodes with regular tags may be coalesced during summarization, all path tree nodes store the number of nodes in the original un-summarized path tree that they represent and the total frequency of these nodes. $*$-nodes always represent multiple nodes in the original path tree that have been deleted, while nodes with regular tags can represent either single nodes in the original path tree or multiple nodes with the same tag name that have been coalesced because their parents were coalesced.

During path tree summarization, we do not consider $*$-nodes as candidates for deletion. Coalesced nodes with regular tag names are deleted only if their *total* frequency is the lowest frequency in the path tree.

When the size of the path tree is reduced enough so that it fits in the available memory, we traverse the tree and compute, for every $*$-node, the average frequency of the multiple deleted nodes that it represents. This is the frequency that is used for selectivity estimation.

Nodes with regular tag names (i.e., not $*$) can represent multiple nodes in the original path tree if they are coalesced when their parents are coalesced. For such nodes, we do *not* always use the average frequency for selectivity estimation. We use the average frequency in some cases, but we use the total frequency in other cases. Thus, both the total frequency and the number of nodes represented have to be retained so that the average frequency can be computed when needed. Details are presented in the next section.

Figures 43 and 44 present a path tree of 12 nodes and its sibling-$*$ summarization to 9 nodes. The nodes of this tree are marked for deletion in the order A, I, J, E, H, D, C, G. Sibling nodes that can be coalesced are identified when marking J, D, and G. Coalescing G and H allows us to coalesce the two K nodes, saving us an extra node. The summarized path tree retains both the total frequency of the K node and the number of nodes it represents.

The $*$-nodes in sibling-$*$ summarization try to preserve the exact position of the deleted

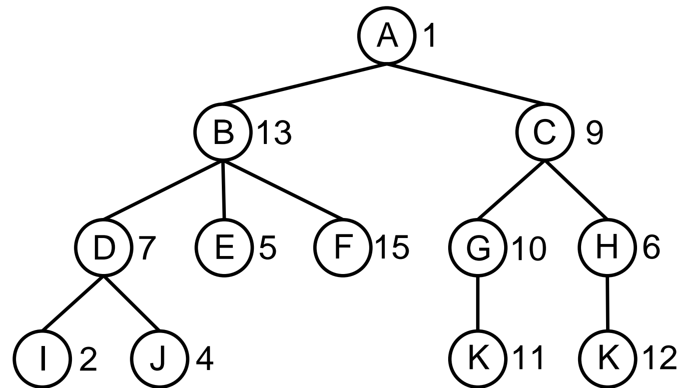


Figure 43: An example path tree

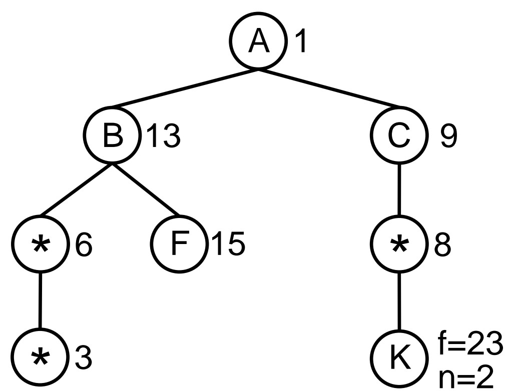


Figure 44: The sibling-$*$ summarization of the path tree in Figure 43

nodes in the original path tree. The cost of preserving this exact information is that we may need to delete up to $2n$ nodes to reduce the size of the tree by n nodes. Furthermore, the deleted nodes are not always the ones with the lowest frequencies. A low-frequency node may be "spared" from deletion if it is the only node marked for deletion among its siblings.

4.3.2 Selectivity Estimation

To estimate the selectivity of a query path expression using a summarized path tree, we try to match the tags in the path expression with tags in the path tree to find all path tree nodes to which the path expression leads. The estimated selectivity is the total frequency of all these nodes. When we cannot match a tag in the path expression to a path tree node with a regular tag, we try to match it to a $*$-node that can take its place.

Tags in any position of the query path expression can be matched to $*$-nodes. For example, the path expression A/B/C would match all of A/$*$/C, A/$*$/$*$, and $*$/B/$*$. To find all the matches for a query path expression, we traverse the path tree looking for nodes whose tags match tags in any position of the path expression and start navigating from these nodes, matching with $*$-nodes when necessary. We allow matches with any number of $*$-nodes as long as they include at least one node with a regular tag name. We do not allow matches consisting entirely of $*$-nodes because there is not enough confidence in such matches.

When we match a tag from the query path expression with a $*$-node, we are making the assumption that this tag was present in the original path tree but was deleted and replaced with the $*$-node. We are essentially assuming that all query path expressions ask for paths that exist in the data, so we aggressively try to match them in the summarized path tree.

If a match of the query path expression in the path tree ends at a node, A, with a regular tag that represents multiple coalesced nodes of the original path tree, we check whether or not this match took us through a $*$-node. If the match took us through one or more $*$-nodes, node A contributes the *average* frequency of the nodes it represents to the estimated selectivity. A $*$-node encountered during the match represents multiple deleted nodes from the original path tree. We assume that if we were using the original un-summarized path tree, the match would

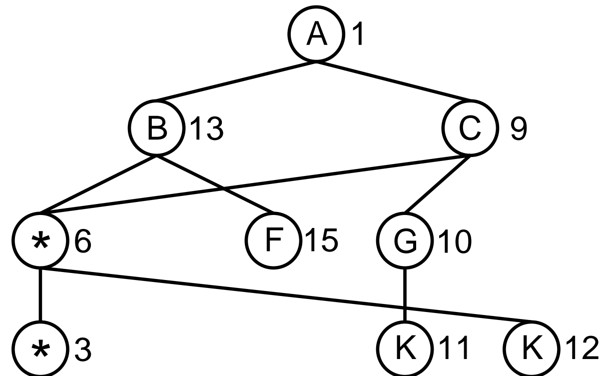


Figure 45: The level-∗ summarization of the path tree in Figure 43

have taken us through only one of the nodes represented by the ∗-node and ended at only one of the nodes represented by node A, so node A contributes its average frequency. On the other hand, if the match in the summarized path tree did not take us through a ∗-node, then the match in the un-summarized path tree would have taken us to *all* the nodes represented by node A, so node A contributes the *total* frequency of the nodes it represents to the estimated selectivity. This explains why nodes with regular tags that represent multiple coalesced nodes of the original path tree need to retain both the total frequency and the number of nodes they represent.

4.3.3 Level-∗

The second method for summarizing path trees, which we call *level-∗*, has a ∗-node for every level of the path tree representing all deleted nodes at this level. As before, we delete the lowest frequency path tree nodes. All nodes deleted at any given level of the path tree are coalesced into the ∗-node for this level. The parents of these nodes become parents of the ∗-node and the children of these nodes become children of the ∗-node. This means that the path tree can become a *dag*. However, we can still us the same selectivity estimation algorithm as for sibling-∗. As in sibling-∗, when the children of a deleted node are added to the children of the corresponding ∗-node, any nodes that become siblings that have the same tag must be coalesced. Figure 45 shows the level-∗ summarization of the path tree in Figure 43 to 9 nodes.

The ∗-nodes in level-∗ summarization preserve only the level in the path tree of the deleted

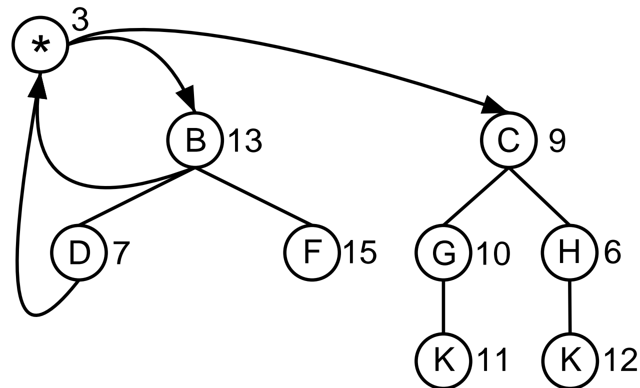


Figure 46: The global-$*$ summarization of the path tree in Figure 43

nodes, not their exact position as in sibling-$*$. Hence, level-$*$ usually deletes fewer nodes than sibling-$*$ to reduce the size of the tree by the same amount. To reduce the size of a path tree by n nodes, level-$*$ needs to delete no more than $n + l$ nodes, where l is the number of levels in the tree.

4.3.4 Global-$*$

The third method for summarizing path trees is the *global-*$*$ method, in which a single $*$-node represents all low-frequency nodes deleted from anywhere in the path tree. The parents of the deleted nodes become parents of the $*$-node and their children become children of the $*$-node, so the path tree can become a *cyclic graph* with cycles involving the global $*$-node. Nevertheless, we can still use the same selectivity estimation algorithm as for sibling-$*$ and level-$*$. Figure 46 shows the global-$*$ summarization of the path tree in Figure 43 to 9 nodes.

Global-$*$ preserves less information about the deleted nodes than sibling-$*$ or level-$*$, so it has to delete fewer nodes. To reduce the size of a path tree by n nodes, global-$*$ deletes $n + 1$ nodes.

4.3.5 No-$*$

The final path tree summarization method, which we call *no-*$*$, does not rely on $*$-nodes at all to represent deleted nodes. In the no-$*$ method, low-frequency nodes are simply deleted and not replaced with $*$-nodes. The path tree can become a *forest* with many roots. To reduce the size of a path tree by n nodes, no-$*$ deletes exactly n nodes. This is only one node less than global-$*$.

The fundamental difference between no-$*$ and the methods that use $*$-nodes is not the amount of memory saved, but rather the effect of the absence of $*$-nodes on selectivity estimation. When matching a query path expression in a path tree summarized with no-$*$, if any tag in the path expression is not found, we assume that the entire path expression does not exist. No-$*$ conservatively assumes that nodes that do not exist in the summarized path tree did not exist in the original path tree. Methods that use $*$-nodes, on the other hand, aggressively assume whenever possible that nodes that do not exist in the summarized path tree *did* exist in the original path tree but were deleted and replaced with $*$-nodes. The characteristics of the query workload determine which of these two assumptions is more accurate.

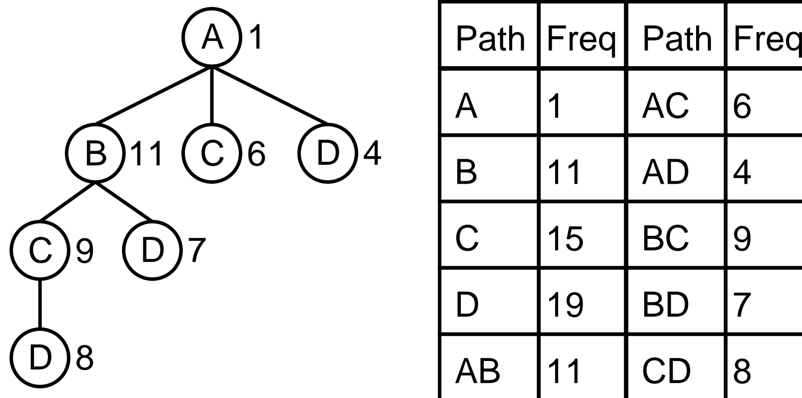
4.4 Markov Tables

In this section, we describe an alternative method of representing the structure of XML data for selectivity estimation.

We construct a table of all the distinct paths in the data of length up to m and their frequency, where m is a parameter ≥ 2. The table provides selectivity estimates for all path expressions of length up to m. To estimate the selectivity of longer path expressions, we combine several paths of length m using the formula

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

where $f(t_1/t_2/\cdots/t_n)$ is the frequency of the path $t_1/t_2/\cdots/t_n$. $f(t_1/t_2/\cdots/t_k)$ for any $k \leq m$ is obtained by a lookup in the table of paths. This can be done in $O(1)$ by implementing the table of paths as a hash table.



Path	Freq	Path	Freq
A	1	AC	6
B	11	AD	4
C	15	BC	9
D	19	BD	7
AB	11	CD	8

Figure 47: A path tree and the corresponding Markov table for $m = 2$

For example, if $m = 3$ and the query path expression is A/B/C/D, the formula used would be:

$$f(\texttt{A/B/C/D}) = f(\texttt{A/B/C})\frac{f(\texttt{B/C/D})}{f(\texttt{B/C})}$$

The fraction $f(\texttt{B/C/D})/f(\texttt{B/C})$ can be interpreted as the average number of D elements contained in all B/C paths.

In this approach, we are assuming that a tag in any path in the XML data depends only on the $m - 1$ tags preceding it. We are, in effect, modeling the paths in the XML data as a *Markov process* of order $m - 1$, so we call the table of paths that we use the *Markov table*. We can think of the Markov table as a compact representation of the very sparse transition matrix of this order $m - 1$ Markov process. Figure 47 presents a path tree and its corresponding Markov table for $m = 2$.

The "short memory" assumption made in this approach is very intuitive, and we expect it to hold for most XML data, even for $m = 2$ or 3. It was suggested very early on that the short memory assumption holds for text in natural languages [Sha48, Sha51]. This assumption is also the basis for *maximal overlap parsing*, which is used in estimating the selectivity of string queries in [JNS99] and [CJK$^+$01].

Markov tables represent an accurate approximation of the structure of the XML data based on the short memory assumption, but they may not fit in the available memory. As we did for path trees, we summarize Markov tables by deleting low-frequency paths.

Low-frequency paths of length 1 or 2 that are deleted from the Markov table are replaced with special $*$-*paths* ("star paths") that preserve some of the information lost by deletion. These $*$-paths are very similar to the $*$-nodes used in path tree summarization. Low-frequency paths of length greater than 2 (for $m > 2$) are discarded and not replaced with $*$-paths. If estimating the selectivity of a query path expression using Equation 1 involves looking up a path of length > 2 that is not found in the Markov table, we switch to using Equation 1 with paths of length 1 and 2 (i.e., with $m = 2$). This corresponds to using a Markov process of order 1.

Next, we describe three methods of summarizing Markov tables that differ in the way they use $*$-paths to handle deleted paths of length 1 and 2. For all these methods, deleted paths of length greater than 2 are discarded and not replaced with $*$-paths.

4.4.1 Suffix-$*$

The first method for summarizing Markov tables, which we call *suffix-$*$*, has two special $*$-paths: a path, $*$, representing all deleted paths of length 1, and a path, $*/*$, representing all deleted paths of length 2. When deleting a low-frequency path of length 1, it is added to the path $*$. When deleting a low-frequency path of length 2, we do *not* add it to the path $*/*$ right away.

We keep a set of deleted paths of length 2, S_D. When we delete a path of length 2, say `A/B`, we look for any path in the set S_D that starts with the same tag as the path being deleted, in this case `A`. If no such path is found, we remove the path being deleted, `A/B`, from the Markov table and add it to S_D. If we find such a path in S_D, say `A/C`, we remove `A/B` from the Markov table, remove `A/C` from S_D, and add a new path `A/`$*$ that represents these two paths to the Markov table.

`A/`$*$ represents all deleted paths that start with the tag `A`. We call the path `A/`$*$ a *suffix-$*$ path*. When we delete a path of length 2, before we check the set S_D, we check the Markov table to see if there is a suffix-$*$ path that has the same starting tag as the path being deleted. If we find such a path, the path being deleted is combined with it. In our example, if we delete `A/D`, we would combine it with `A/`$*$.

Suffix-$*$ paths in the Markov table are considered for deletion based on the *total* frequency of the deleted paths that they represent. When a suffix-$*$ path is deleted, it is added to the path $*/*$.

In our example, paths A/B and A/C individually qualify for deletion because of their low frequency. Their total frequency when they are combined into A/$*$ may be high enough to prevent them from being deleted. If at some point during summarization the total frequency of A/$*$ is the lowest frequency in the Markov table, it is deleted and added to the path $*/*$. In this case, paths A/B and A/C could not use the "second chance" provided to them by A/$*$.

This summarization algorithm is a greedy algorithm that may miss opportunities for combining paths. However, it is simple and practical, and it achieves good results.

At the end of summarization, paths still remaining in S_D are added to the path $*/*$, and the average frequencies of all $*$-paths are computed. When selectivity estimation uses a $*$-path, it uses the average frequency of all deleted paths represented by this $*$-path.

To estimate the selectivity of a query path expression, we try to use Equation 1 with the maximum m in the Markov table. If any of the required paths is not found in the table, we switch to using Equation 1 with $m = 2$ for the entire path expression. In this case, if a required path of length 1 is not found, we use the frequency of the path $*$. If a required path of length 2, say A/B, is not found, we look for a path A/$*$. If we find it, we use its frequency. Otherwise we use the frequency of the path $*/*$. If all the paths used for estimation are $*$-paths, we estimate the selectivity of the query path expression to be zero, because we consider that there is not enough confidence in the result of Equation 1 in this case.

4.4.2 Global-$*$

The second method for summarizing Markov tables, which we call *global-*$*$, has only two $*$-paths: a path, $*$, representing all deleted paths of length 1, and a path, $*/*$, representing all deleted paths of length 2. When deleting a low-frequency path of length 1 or 2, it is immediately added to the appropriate $*$-path. Selectivity estimation is the same as for suffix-$*$. Global-$*$ does not preserve as much information about deleted paths as suffix-$*$, but it may delete fewer paths

to summarize the Markov table.

4.4.3 No-$*$

The final method for summarizing Markov tables, which we call *no-*$*$, does not use $*$-paths. Low-frequency paths are simply discarded.

When using Equation 1 for selectivity estimation using a Markov table summarized with no-$*$, if any of the required paths for $m = 2$ is not found, we estimate a selectivity of zero. No-$*$ for Markov tables is similar to no-$*$ for path trees. It conservatively assumes that paths that do not exist in the summarized Markov table did not exist in the original Markov table.

4.5 Experimental Evaluation

In this section, we present an experimental evaluation of our proposed techniques using real and synthetic data sets. We determine the best summarization methods for path trees and Markov tables and the conditions under which each technique wins over the other. We also compare our proposed techniques to the best known alternative: the pruned suffix trees of [CJK$^+$01].

4.5.1 Estimation Using Pruned Suffix Trees

To estimate the selectivity of path expressions and the more general twig queries, [CJK$^+$01] proposes building a trie that represents all the path expressions in the data. For every root-to-leaf path in the data, the path and all its suffixes are stored in the trie. Every possible path expression in the data is, therefore, represented by a trie node. Every trie node contains the total number of times that the path it represents appears in the data. To prune the trie so that it fits in the available memory, the low-frequency nodes are deleted. To avoid deleting internal trie nodes without deleting their descendants, pruning is done based on the total frequency of the node and all its descendants.

To estimate the selectivity of a query path expression, the *maximal (i.e., longest) sub-paths* of this path expression that appear in the trie are determined, and their frequencies are combined

in a way that is similar to the way we combine frequencies in our Markov table technique.

In [CJK$^+$01], every node of the trie stores a *hash signature* of the set of nodes that the path expression it represents is rooted at. These hash signatures are used to combine the selectivities of multiple paths to estimate the selectivity of branching path expressions, or *twig queries*. Since we do not consider branching path expressions, we do not store hash signatures in the nodes of the trie that we use in our experiments. The trie with set hash signatures is referred to in [CJK$^+$01] as a *correlated suffix tree*. We refer to the trie that does not include set hash signatures as the *pruned suffix tree*.

The twig queries considered in [CJK$^+$01] are more general than the path queries that we consider. Twig queries involve branching path expressions, and values in their leaves, both of which we do not consider. We consider the simpler but very common case of XML path expressions. We *expect* the techniques we develop for this case to work better than the techniques developed for the more general case of twig queries. The purpose of our comparison is to validate and quantify this expectation.

Note that Markov tables bear some similarity to pruned suffix trees. A key difference between these two techniques is that Markov tables only store paths of length up to m while pruned suffix trees store paths that may be of any length and that may contain tags that are not needed due to the "short memory" property. Furthermore, the summarization methods for Markov tables are very different from the summarization method for pruned suffix trees.

4.5.2 Data Sets

We present the results of experiments on one synthetic and one real data set. The synthetic data set has 1,000,000 XML elements. Its un-summarized path tree has 3197 nodes and 6 levels, requiring 38 KBytes of memory. The average fanout of the internal nodes of this path tree is 4.6. The un-summarized Markov tables for this data set for $m = 2$ and 3 require 60 KBytes and 110 KBytes, respectively. The frequencies of the path tree nodes follow a Zipfian distribution with skew parameter $z = 1$ [Zipf49]. The Zipfian frequencies are assigned in ascending order to the path tree nodes in breadth first order (i.e., the root node has the lowest frequency and the

Data Set	Random Paths	Random Tags
Synthetic	491	71
DBLP	36,060	343

Table 6: Average result sizes of the XML query workloads

rightmost leaf node has the highest frequency). 50% of the internal nodes of this path tree have repeated tag names, which introduces "Markovian memory" in the data. For example, if two internal nodes of the path tree have tag name A, and only one of these nodes has a child node B, then if we are at a node A, whether or not this node has a child B will depend on *which* A node this is, which in turn depends on how we got to this node from the root node. The details of our synthetic data generation process can be found in Appendix A and in [ANZ01].

The real data set is the DBLP bibliography database [DBLP], which has 1,399,765 XML elements. Its un-summarized path tree has 5883 nodes and 6 levels, requiring 69 KBytes of memory. The un-summarized Markov tables for this data set for $m = 2$ and 3 require 20 KBytes and 98 KBytes, respectively.

4.5.3 Query Workloads

We present results for two workloads for each data set. Each workload consists of 1000 query path expressions having a random number of tags between 1 and 4.

The query path expressions in the first workload, which we call the *random paths* workload, consist of paths that are chosen at random from the path tree of the data set. Thus, all queries have non-zero result sizes. This workload models a user who knows the structure of the data well, and so asks for paths that exist in the data.

The query path expressions in the second workload, which we call the *random tags* workload, consist of random concatenations of the tags that appear in the data set. In this workload, most query path expressions of length 2 or more have a result size of zero. This workload models a user who knows very little about the structure of the data.

The average result sizes of the two workloads on each of the two data sets are given in Table 6.

We also experimented with two other types of workloads for every data set: a *frequency weighted* workload and a *result weighted* workload. The query path expressions for both these workloads are generated by choosing a tag name at random according to the frequency distribution of the tag names in the data set (i.e., the more frequent a tag name is, the more likely it is to be chosen).

In the frequency weighted workload, this chosen tag gets placed in some random position of the query path expression. In the result weighted workload, the chosen tag always becomes the *final* tag of the query path expression (i.e., the "target" tag). The remaining tags of the query path expressions in both workloads are generated as in the random paths workload. This ensures that these query path expressions ask for paths that do exist in the data.

These two workloads represent two ways in which the query distribution can follow the data distribution for XML data. For all our experiments, we found that the conclusions from these two workloads were the same as from the random paths workload. Thus, in the interest of space, we do not present results for these two workloads.

4.5.4 Performance Evaluation Method

We present the performance of the different selectivity estimation techniques in terms of their *average absolute error* for all queries in the workload. The conclusions from the relative error are the same, but the relative error is not defined for many queries in the random tags workloads because their actual result size is 0.

For a given data set and query workload, we vary the available memory for the different selectivity estimation techniques from 5 KBytes to 50 KBytes and present the average absolute error for the 1000 queries in the workload for each technique at each memory setting.

In all the data structures used for estimation, tag names are not stored as character strings. Instead, we hash the tag names and store their hash values. This conserves memory because a tag requires one integer of storage regardless of its length.

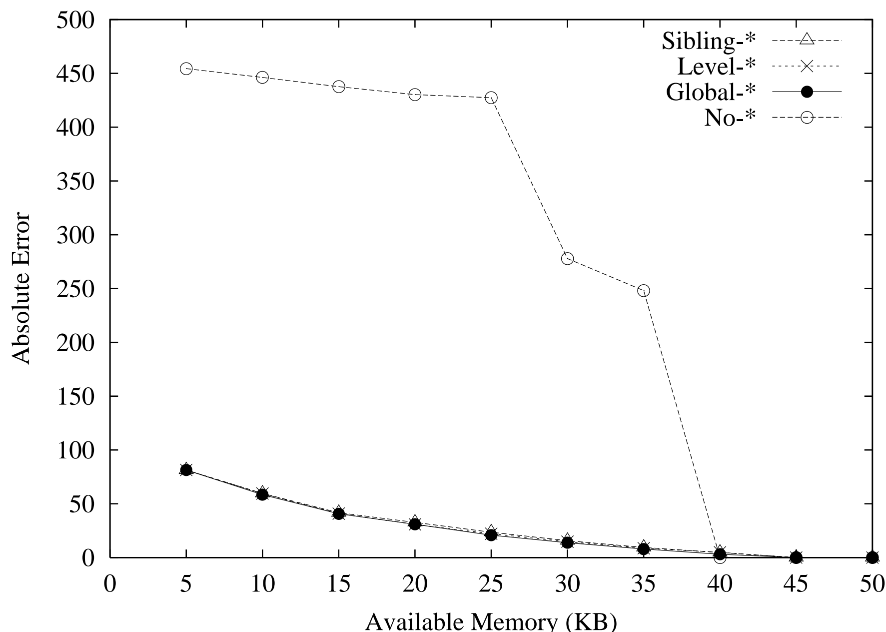


Figure 48: Path tree summarization for the synthetic data set and random paths workload

4.5.5 Summarizing Path Trees

In this section, we illustrate the best summarization methods for path trees. Figures 48 and 49 present the average absolute error in selectivity estimation using path trees summarized in different ways for the random paths and random tags workloads on the synthetic data set, respectively.

Figure 48 shows that, for the random paths workload, all summarization methods that use $*$-nodes have similar performance, and they are all better than the no-$*$ method. The methods using $*$-nodes are better than no-$*$ because the query path expressions ask for paths that exist in the data, so the aggressive assumption that these methods make about nodes not in the summarized path tree are mostly valid and result in higher accuracy. Since all methods using $*$-nodes have similar performance, we conclude that the more detailed information maintained by the more complex sibling-$*$ and level-$*$ methods does not translate into higher estimation accuracy. Hence, for workloads that ask for paths that exist in the data, global-$*$ is the best path tree summarization method.

The situation is different for the random tags workload in Figure 49. Since query path

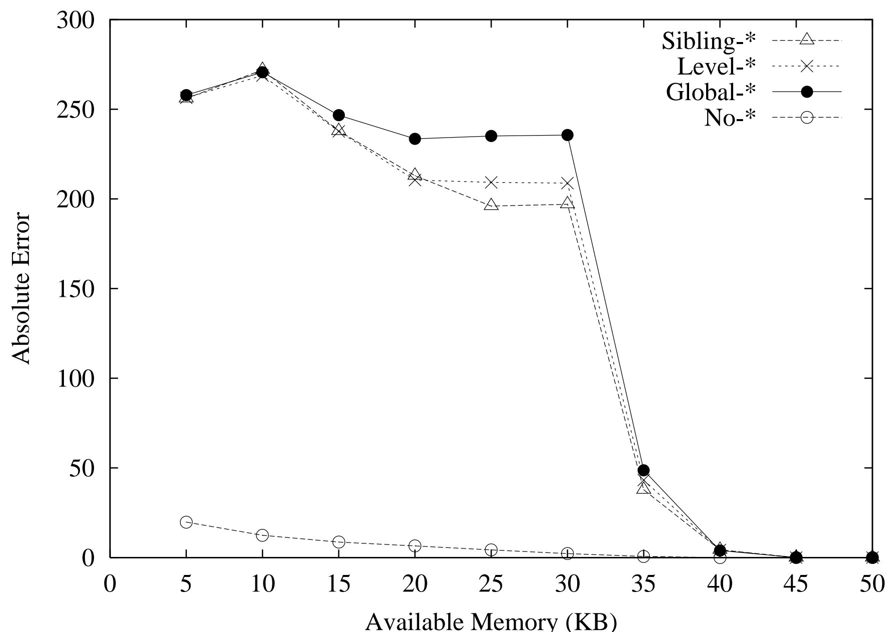


Figure 49: Path tree summarization for the synthetic data set and random tags workload

expressions in the random tags workload ask for paths that mostly do not exist in the data, the correct thing to do when we are unable to match the entire query path expression with nodes in the path tree is to estimate a selectivity of zero. This is what no-$*$ does. The methods that use $*$-nodes are misleading in this case because they allow us to match tags in the query path expression with $*$-nodes in the path tree even when the query path expression does not exist in the data. This results in significantly less accuracy than no-$*$. Hence, for workloads that ask for paths that do not exist in the data, no-$*$ is the best path tree summarization method.

4.5.6 Summarizing Markov Tables

In this section, we illustrate the best summarization methods for Markov tables. For all data sets and query workloads, we observe that un-summarized Markov tables with $m = 3$ are very accurate, so we only evaluate the performance of summarized Markov tables with $m = 2$ and 3, but not with $m > 3$. In general, the practical values of m are 2 and 3.

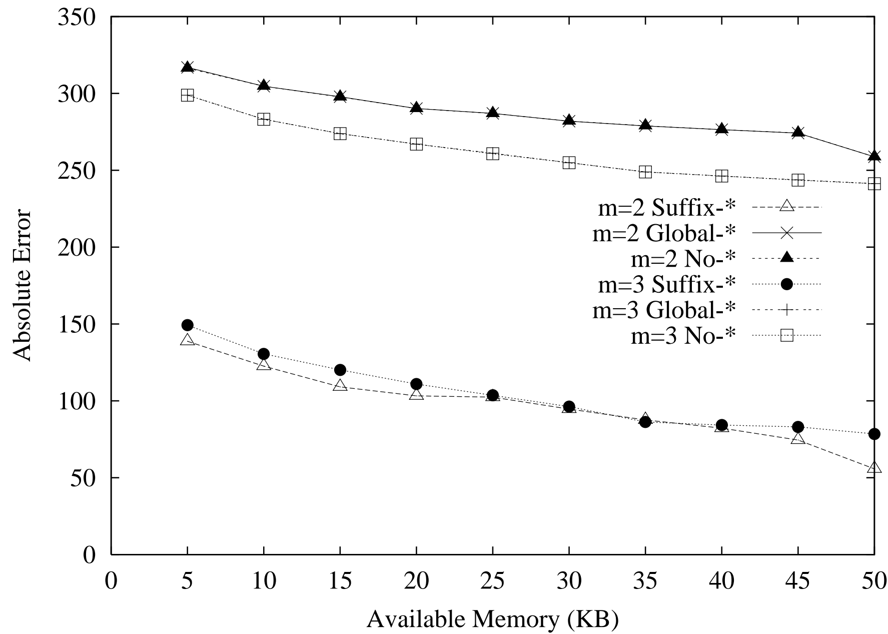


Figure 50: Markov table summarization for the synthetic data set and random paths workload

Figures 50 and 51 present the estimation accuracy using Markov tables summarized in different ways for the random paths and random tags workloads on the synthetic data set, respectively.

Figure 50 shows that, for the random paths workload, suffix-$*$ summarization is best. Unlike for path trees, the summarization method that preserves the most information about deleted paths works best for Markov tables. $m = 2$ and $m = 3$ have similar performance, so the conclusion is to use the simpler $m = 2$. Thus, the best Markov table approach for workloads that ask for paths that exist in the data is to use $m = 2$ and suffix-$*$ summarization.

Figure 51 shows that, for the random tags workload, $m = 2$ and global-$*$ or no-$*$ summarization are the best methods. No-$*$ works well for the same reason that it works well in path tree summarization for the random tags workload. Global-$*$ is similar in performance to no-$*$ because many of the query path expressions that ask for paths that do not exist in the data get matched entirely with the paths $*$ and $*/*$, so they have an estimated selectivity of zero, which is correct. The best Markov table approach for workloads that ask for paths that do not exist in the data is, therefore, to use $m = 2$ and the simpler no-$*$ summarization.

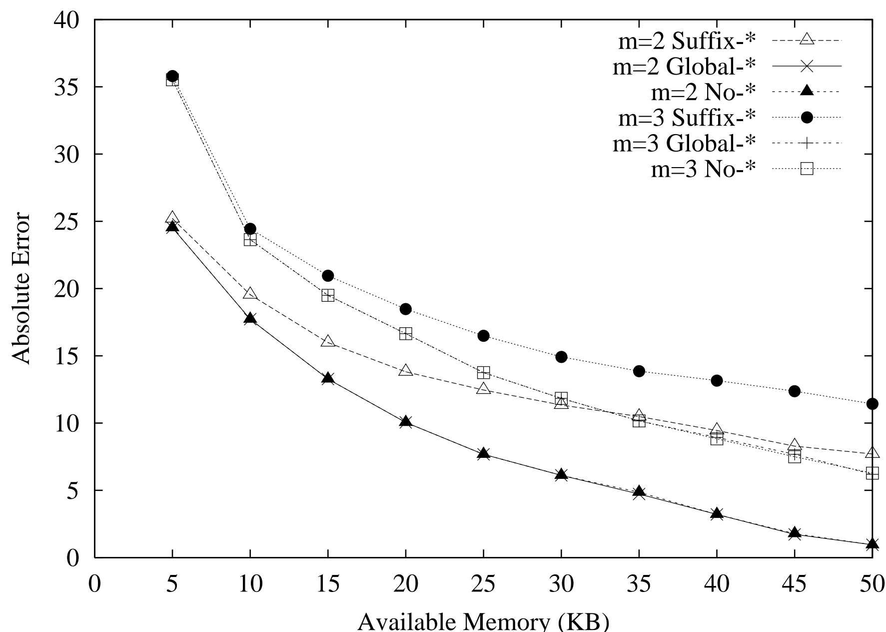


Figure 51: Markov table summarization for the synthetic data set and random tags workload

4.5.7 Estimation Accuracy

In this section, we compare the best techniques for path trees and Markov tables as identified in the previous sections. We also compare these techniques to the pruned suffix tree approach.

Figure 52 presents the selectivity estimation errors for the random paths workload on the synthetic data set using path trees summarized with global-$*$, Markov tables with $m = 2$ summarized with suffix-$*$, and pruned suffix trees. Figure 53 presents the selectivity estimation errors for the random tags workload on the synthetic data set using path trees summarized with no-$*$, Markov tables with $m = 2$ summarized with no-$*$, and pruned suffix trees. Figures 54 and 55 present the same information for the DBLP data set.

For the synthetic data set, path trees are the most accurate technique, and both path trees and Markov tables are more accurate than pruned suffix trees. For the DBLP data set, Markov tables are the most accurate technique, and they are much more accurate than pruned suffix trees. Path trees, on the other hand, are the least accurate technique for the DBLP data set. Their estimation error, which is too high to show in Figures 54 and 55, is usually greater than 100.

The DBLP data set represents bibliography information for many different conferences and

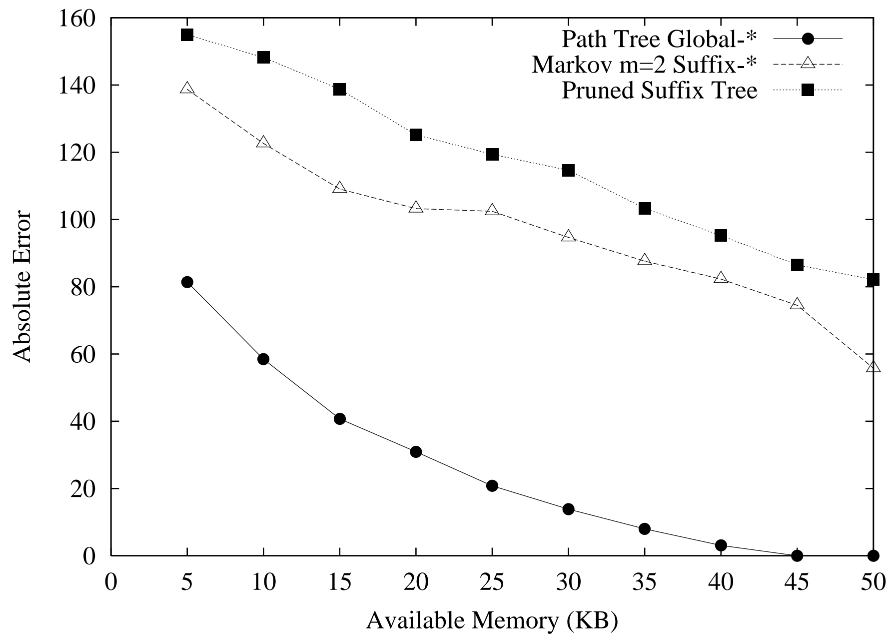


Figure 52: Estimation accuracy for the synthetic data set and random paths workload

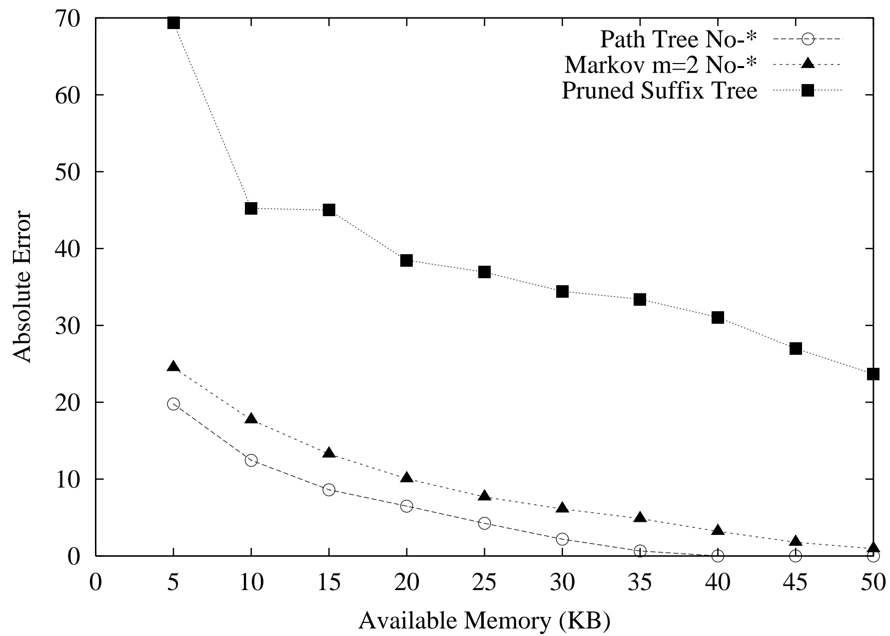


Figure 53: Estimation accuracy for the synthetic data set and random tags workload

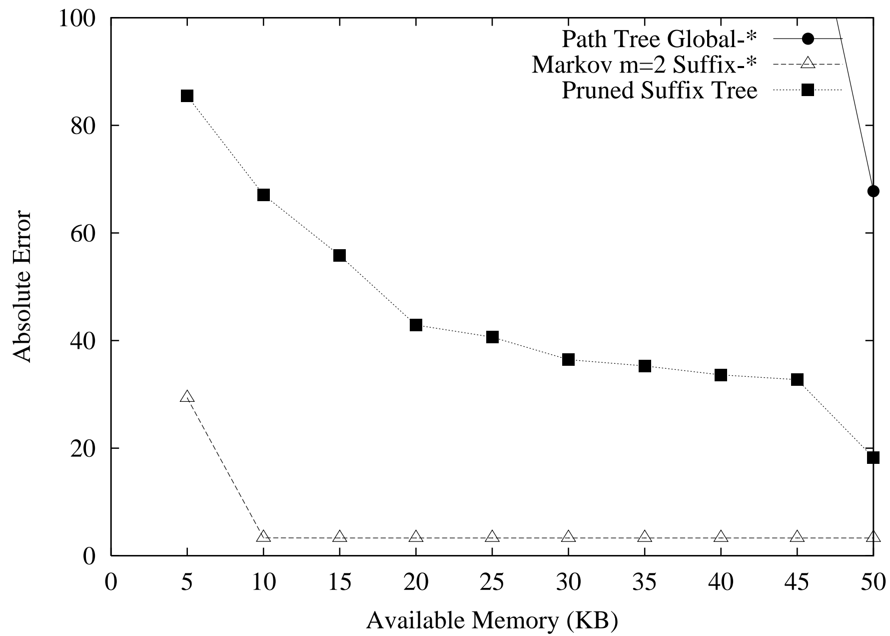


Figure 54: Estimation accuracy for the DBLP data set and random paths workload

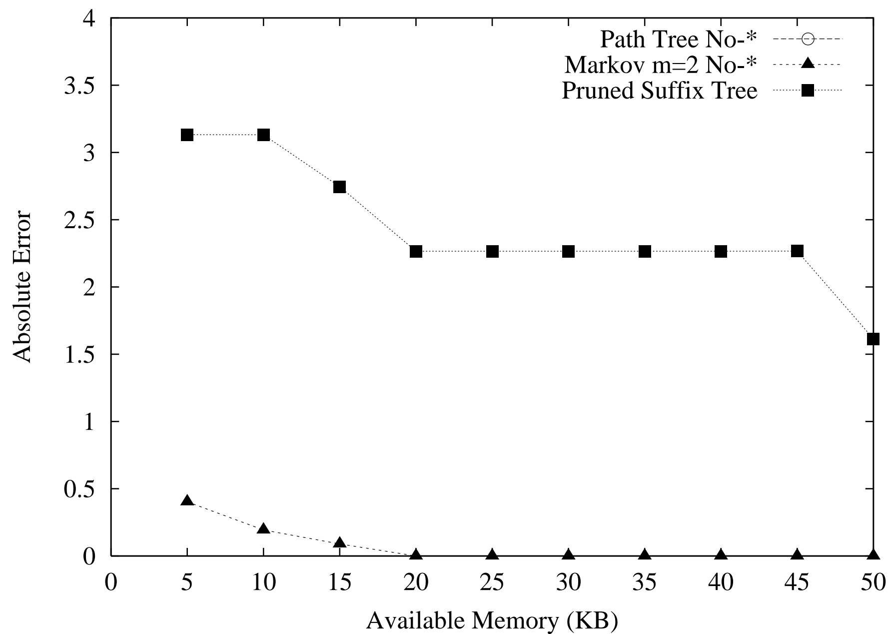


Figure 55: Estimation accuracy for the DBLP data set and random tags workload

journals. Each conference or journal is a different sub-tree of the path tree, but the structure of the data within each of these sub-trees is the same for any conference or journal. Path tree summarization cannot compactly represent the common structure of these sub-trees. On the other hand, Markov tables, and to a lesser extent pruned suffix trees, can effectively capture this common structure. For example, note that each bibliography entry in every conference or journal in the DBLP data set has one or more `author` elements. In the path tree, there will be an `author` node in the sub-tree corresponding to every conference or journal. Some of these nodes will have to be deleted during tree summarization resulting in a loss of accuracy. In the Markov table, on the other hand, there will only be one `author` path for all the `author` nodes in the path tree.

Thus, if the data has many common sub-structures, Markov tables should be used. If the data does not have many common sub-structures, path trees should be used. Choosing the appropriate selectivity estimation technique always results in higher accuracy than using pruned suffix trees.

4.6 Conclusions

The proliferation of XML on the Internet will enable novel applications that query "all the data on the Internet." The queries posed by these applications will involve navigating through the structure of the XML data using path expressions. Optimizing these queries will, therefore, require estimating the selectivity of these path expressions. We propose two techniques for summarizing the structure of large-scale XML data in a small amount of memory for estimating the selectivity of XML path expressions: path trees and Markov tables.

The correct selectivity estimation technique to use depends on the XML data whose structure is being summarized. If the data has a lot of common structures, Markov tables with $m = 2$ should be used. If the data does not have such common structures, path trees should be used.

The best way to summarize path trees and Markov tables depends on the characteristics of the query workload. If the query path expressions ask for paths that exist in the data, then the aggressive global-$*$ and suffix-$*$ techniques should be used for summarizing path trees and

Markov tables, respectively. If the query path expressions ask for paths that do not exist in the data, then the conservative no-$*$ technique should be used for both path trees and Markov tables.

The correct choice from our techniques always results in higher selectivity estimation accuracy than pruned suffix trees [CJK$^+$01], the best previously known alternative. The techniques in [CJK$^+$01] solve a more general problem, so their full power is not evident in the comparison with our proposed techniques that solve a simpler problem. It is an open question whether XML query optimizers will require selectivity information about simple path expressions, in which case our proposed techniques would be better, or about more complex path expressions involving branches and values, in which case the techniques in [CJK$^+$01] would be better. Answering this question is a possible area for future work.

At this time, we cannot conclusively determine the typical characteristics of XML data that will be available on the Internet, so we cannot recommend an overall best technique for selectivity estimation. However, if we were to make an educated guess, we would say that things like standard DTDs and schema libraries will result in a lot of common sub-structures. We would also guess that users typically know enough about the semantics of the tag names to ask for paths that generally do exist in the data. Thus, we would recommend Markov tables with $m = 2$ and suffix-$*$ summarization. If this guess proves to be wrong, we simply need to choose another one of our proposed techniques. In any case, developing a general framework for choosing the correct selectivity estimation technique is an interesting topic for future work.

Chapter 5

Conclusions

Accurate selectivity and cost models are essential for effective query optimization in database systems. Current advances in database research present new challenges that require the development of advanced selectivity and cost estimation techniques. This dissertation presented three such advanced selectivity and cost estimation techniques related to different trends in database research. Chapter 2 discussed accurately estimating the cost of spatial selections, Chapter 3 presented self-tuning histograms, and Chapter 4 discussed estimating the selectivity of XML path expressions.

In Chapter 2, we demonstrated that accurate estimation of the cost of spatial selections requires taking into account the CPU and I/O costs of both the filtering and the refinement steps of spatial query processing. Estimating the cost of the refinement step requires estimating the MBR selectivity of the spatial selection query and the average number of vertices in the candidate polygons identified by the filtering step. We developed a novel statistics data structure, which we call the SQ-histogram, that effectively estimates these two quantities.

Estimating the cost of spatial operations, in general, requires information about the location, size, and complexity of the data objects. We demonstrated how to effectively capture these properties using SQ-histograms, and how to use them for accurate estimation of the cost of spatial selections. This is an example of cost estimation techniques developed to provide support for *new data types* in database systems.

In Chapter 3, we introduced self-tuning histograms. Self-tuning histograms are a novel way of building histograms at a low cost based on feedback from the query execution engine without looking at the data. Multi-dimensional self-tuning histograms are particularly attractive, since they provide a low-cost alternative to traditional multi-dimensional structures proposed in the literature.

There is a current trend in database research toward building *self-tuning self-administering* database systems. Self-tuning histograms can be a useful tool for such systems. The low cost of self-tuning histograms can help a self-tuning self-administering database system experiment with building many different histograms on many different combinations of data columns. This is useful since the system cannot rely on an "expert" database administrator to decide which histograms to build.

In Chapter 4, we presented two techniques for estimating the selectivity of simple path expressions on complex large-scale XML data: path trees and Markov tables. Both techniques work by summarizing the structure of the XML data in a small amount of memory and using this summary for selectivity estimation. Markov tables are the technique of choice if the XML data has many common sub-structures, while path trees work best for data that does not have such sub-structures.

Path trees and Markov tables must be summarized so that they fit in the available memory. If the query path expressions ask for paths that exist in the data, we should use aggressive summarization techniques that replace information deleted from our structures with coarse grained information in $*$-nodes or $*$-paths. In particular, we should use global-$*$ summarization for path trees and suffix-$*$ summarization for Markov tables. If the query path expressions ask for paths that do not exist in the data, then we should use the conservative no-$*$ summarization technique. Path trees and Markov tables are examples of the selectivity estimation techniques required to provide support for *querying the Internet*.

Declarative query languages and automatic query optimization are key features of database systems. Query optimizers must, therefore, keep up with advances in different areas of database research. This requires that the selectivity and cost estimation techniques used by query optimizers also keep up with advances in database research. This dissertation presented different examples of how this can be done.

Bibliography

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proc. Int. Conf. on Very Large Data Bases*, pages 591–600, Rome, Italy, September 2001.

[AC99] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–192, Philadelphia, Pennsylvania, June 1999.

[AN00] Ashraf Aboulnaga and Jeffrey F. Naughton. Accurate estimation of the cost of spatial selections. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 123–134, San Diego, California, March 2000.

[ANZ01] Ashraf Aboulnaga, Jeffrey F. Naughton, and Chun Zhang. Generating synthetic complex-structured XML data. In *Proc. 4th Int. Workshop on the Web and Databases (WebDB'2001)*, pages 79–84, Santa Barbara, California, May 2001.

[Aoki99] Paul M. Aoki. How to avoid building DataBlades that know the value of everything and the cost of nothing. In *Proc. Int. Conf. on Scientific and Statistical Database Management*, pages 122–133, Cleveland, Ohio, July 1999.

[APR99] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, Philadelphia, Pennsylvania, June 1999.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 211–222, Santa Barbara, California, May 2001.

[BF95] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proc. Int. Conf. on Very Large Data Bases*, pages 299–310, Zurich, Switzerland, September 1995.

[BF97] Elisa Bertino and Paola Foscoli. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, May 1997.

[BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 197–208, Minneapolis, Minnesota, May 1994.

[BPS98] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen (eds.). Extensible markup language (XML) 1.0. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210/, February 1998.

[CD99] James Clark and Steve DeRose (eds.). XML path language (XPath) version 1.0. W3C Recommendation available at http://www.w3.org/TR/xpath/, November 1999.

[CDN+97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, and Dhaval Shah. The BUCKY object-relational benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135–146, Tucson, Arizona, May 1997.

[Census] Census-income database. University of California, Irvine, knowledge discovery in databases archive. http://kdd.ics.uci.edu/databases/census-income/census-income.html.

[CFR+01] Don Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu (eds.). XQuery: A query language for XML. W3C Working Draft available at http://www.w3.org/TR/xquery/, February 2001.

[CJK+01] Zhiyuan Chen, H.V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond Ng, and Divesh Srivastava. Counting twig matches in a tree. In *Proc. IEEE Int. Conf. on Data Engineering*, pages 595–604, Heidelberg, Germany, April 2001.

[CKKM00] Zhiyuan Chen, Flip Korn, Nick Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 216–225, Dallas, Texas, May 2000.

[CN97] Surajit Chaudhuri and Vivek Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proc. Int. Conf. on Very Large Data Bases*, pages 146–155, Athens, Greece, August 1997.

[CR94] Chung-Min Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 161–172, Minneapolis, Minnesota, May 1994.

[DB2] IBM DB/2 Universal Database administration guide: Performance. Available from http://www-4.ibm.com/software/data/db2/library/publications/.

[DBLP] DBLP Bibliography. http://www.informatik.uni-trier.de/~ley/db/.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer-Verlag, 1997.

[DD93] Chris J. Date and Hugh Darwen. *A Guide to the SQL Standard*. Addison-Wesley, Reading, Massachusetts, third edition, 1993.

[DS00] Kalen Delaney and Ron Soukup. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 4–13, Minneapolis, Minnesota, May 1994.

[GGT95] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. A cost model for clustered object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 323–334, Zurich, Switzerland, September 1995.

[GGT96] Georges Gardarin, Jean-Robert Gruser, and Zhao-Hui Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 390–401, Mumbai (Bombay), India, September 1996.

[GMP97] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *Proc. Int. Conf. on Very Large Data Bases*, pages 466–475, Athens, Greece, August 1997.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proc. 2nd Int. Workshop on the Web and Databases (WebDB'99)*, pages 25–30, Philadelphia, Pennsylvania, June 1999.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–54, Boston, Massachusetts, June 1984.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. Int. Conf. on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[GW99] Roy Goldman and Jennifer Widom. Approximate DataGuides. In *Proc. Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*, Jerusalem, Israel, January 1999.

[HKMY98] Yukinobu Hamuro, Naoki Katoh, Yasuyuki Matsuda, and Katsutoshi Yada. Mining pharmacy data helps to make profits. *Data Mining and Knowledge Discovery*, 2(4):391–398, December 1998.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. Int. Conf. on Very Large Data Bases*, pages 562–573, Zurich, Switzerland, September 1995.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[JKNS99] H.V. Jagadish, Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. Multidimensional substring selectivity estimation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 387–398, Edinburgh, Scotland, September 1999.

[JNS99] H.V. Jagadish, Raymond T. Ng, and Divesh Srivastava. Substring selectivity estimation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 249–260, Philadelphia, Pennsylvania, May 1999.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 106–117, Seattle, Washington, June 1998.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proc. 2nd Int. Conference on Information and Knowledge Management*, pages 490–499, Washington, DC, November 1993.

[Kooi80] Robert P. Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Case Western Reserve University, September 1980.

[KVI96] P. Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 282–293, Montreal, Canada, June 1996.

[LNS90] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–11, Atlantic City, New Jersey, May 1990.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 28–36, Chicago, Illinois, June 1988.

[MRL98] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 426–435, Seattle, Washington, June 1998.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proc. 7th Int. Conf. on Database Theory*, pages 277–295, Jerusalem, Israel, January 1999.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 448–459, Seattle, Washington, June 1998.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proc. Int. Conf. on Very Large Data Bases*, pages 315–326, Edinburgh, Scotland, September 1999.

[NDM+01] Jeffrey Naughton, David DeWitt, David Maier, et al. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[Niagara] The Niagara Project home page. http://www.cs.wisc.edu/niagara/.

[ODE95] Cetin Ozkan, Asuman Dogac, and Cem Evrendilek. A heuristic approach for optimization of path expressions. In *Proc. Int. Conf. on Database and Expert Systems Applications*, pages 522–534, London, United Kingdom, September 1995.

[O'R98] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 1998.

[Ore86] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 326–336, May 1986.

[P$^+$97] Jignesh Patel et al. Building a scalable geo-spatial database system: Technology, implementation, and evaluation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 336–347, Tucson, Arizona, May 1997.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 259–270, Montreal, Canada, June 1996.

[PG02] Neoklis Polyzotis and Minos Garofalakis. Statistical synopses for graph-structured XML databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Madison, Wisconsin, June 2002. (to appear).

[PI97] Viswanath Poosala and Yannis Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. Int. Conf. on Very Large Data Bases*, pages 486–495, Athens, Greece, August 1997.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 294–305, Montreal, Canada, May 1996.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc.*

ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pages 214–221, Washington, DC, May 1993.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, Boston, Massachusetts, May 1979.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[SAX] SAX 2.0: The simple API for XML. http://www.saxproject.org/.

[SFGM93] Michael Stonebraker, Jim Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 storage benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 2–11, Washington, D.C., May 1993.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[Sha51] Claude E. Shannon. Prediction and entropy of printed english. *The Bell System Technical Journal*, 30:50–64, January 1951.

[TS96] Yannis Theodoridis and Timos Sellis. A model for the prediction of R-tree performance. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 161–171, Montreal, Canada, June 1996.

[WVLP01] Min Wang, Jeffrey Scott Vitter, Lipyeow Lim, and Sriram Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Proc. Symposium on Spatial and Temporal Databases (SSTD 2001)*, pages 175–196, Redondo Beach, California, July 2001.

[Xyl01] Lucie Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2):40–47, June 2001.

[Zipf49] George K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Reading, Massachusetts, 1949.

Appendix A

Generating Synthetic Complex-structured XML Data

We developed a data generator that generates synthetic complex-structured XML documents for our experiments in Chapter 4. The data generator can generate tree-structured XML documents of arbitrary complexity, and it allows the user to specify a wide range of characteristics for the generated data by varying a number of simple and intuitive input parameters. The data generator can also generate values within the XML elements, but element values are of no interest to us in our experiments, so we do not discuss them any further. The data generator is described in [ANZ01] and is publicly available from [Niagara].

The data generator starts by generating a path tree (Section 4.3) that represents the structure of an XML document. It then assigns tag names to the nodes of this tree and specifies the frequency distribution of the XML elements represented by these nodes. Next, it uses the information in this path tree to generate the XML document.

A.1 Generating the Tree Structure of the Data

At the start of the data generation process, only the structure of the path tree is generated. Path tree nodes are generated without tag names or node frequencies. This information is added to the nodes later in the data generation process.

The input parameters of the data generator include parameters that control the structure of the generated path tree and allow us to generate arbitrarily complex trees. These parameters are the *number of levels in the path tree*, and for every level of the path tree except the lowest level the *minimum and maximum number of children* for nodes at this level. Nodes at the lowest level

of the path tree are all leaf nodes that have no children.

The data generator generates the path tree in a depth first manner, from root to leaves. The number of children of an internal path tree node at a particular level of the path tree is chosen at random according to a uniform distribution from the range specified by the input parameters of the data generator for the possible number of children of nodes at this level.

A.2 Tag Names – Recursion and Repetition

The next step in the data generation process is assigning tag names to the path tree nodes. This step starts by traversing the path tree in breadth first order and assigning each node a distinct tag name. The tag names used are `A`, `B`, `C`, . . . , `Z`, `AA`, `AB`, `AC`, etc.

For several reasons, it may be desirable to have *repeated tag names* in the path tree. First, it may be desirable to have *recursion* in the generated XML data (i.e., an XML element having the same tag name as its parent or one of its ancestors). Second, it may be desirable to have multiple XML elements with the same tag name reachable by different paths from the root. Third, repeating the tag names of internal path tree nodes allows us to increase the *Markovian memory* of the paths in the path tree.

An important property of paths in XML data is that the next tag encountered while traversing the path may depend on some of the previous tags encountered. Such paths can be modeled as a *Markov process*, and the number of previous tags that the next tag depends on is the *order* of this process. When the next tag depends on many previous tags, we say that the paths have a lot of Markovian memory. This property of "the future depending on the present and some part of the past" is often exhibited in real XML data.

When all nodes of the path tree have distinct tag names, the tag names of the children of a path tree node depend only on the tag name of this node and not on the tag names of its ancestors. In this case, paths in the path tree can be modeled as a Markov process of order 1. When there are repeated tag names in the internal nodes of the path tree, the tag names of the children of a path tree node may depend on the tag names of its ancestors as well as its own tag name. In this case, paths in the path tree can be modeled as a Markov process of order > 1, so

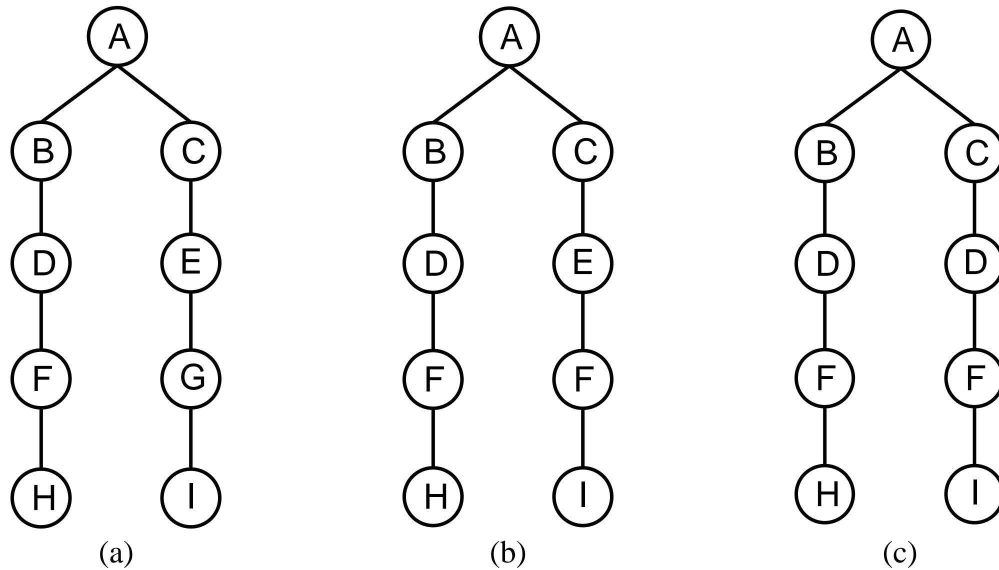


Figure 56: Markovian memory in the generated path tree due to repeated tag names

there is more Markovian memory in the paths of the path tree.

For example, consider the path trees in Figure 56. The nodes of the path tree in Figure 56(a) have distinct tag names. Knowing that node `F` has a child `H` does not require any information about its ancestors. Paths in this path tree can be modeled as a Markov process of order 1. In Figure 56(b), the internal nodes of the path tree have repeated tag names. In this case, determining whether node `F` has a child `H` requires knowing the tag name of its parent. A node `F` has a child `H` only if its parent is `D`. Paths in this path tree can be modeled as a Markov process of order 2. The path tree in Figure 56(c) has even more repetition in the tag names of its internal nodes. Paths in this path tree can be modeled as a Markov process of order 3. We know that a node `F` has a child `H` only if we know that its parent is `D` and its grand parent is `B`.

After assigning distinct tag names to the path tree nodes, the data generator introduces different kinds of recursion and repeated tag names into the path tree as specified by several input parameters. The data generator introduces *direct recursion*, in which some nodes have the same tag name as their parents, and *indirect recursion*, in which some nodes have the same tag name as one of their ancestors.

The data generator also introduces repeated tag names among *internal path tree nodes*, which increases the Markovian memory of the paths in the path tree, and among *leaf nodes*

of the path tree, which does not affect the Markovian memory of the paths in the path tree. The data generator can also introduce repeated tag names among nodes of the path tree without restricting whether these nodes are internal nodes or leaf nodes.

A.3 Element Frequency Distribution

The next step in the data generation process is specifying the frequency distribution of the path tree nodes. One of the input parameters to the data generator is the *total number of XML elements to generate*. This is the total frequency of all path tree nodes. The frequencies of the path tree nodes follow a Zipfian distribution [Zipf49]. The *skew parameter* of this distribution, z, is another input parameter of the data generator. $z = 0$ represents a uniform distribution, while $z > 0$ represents a skewed distribution in which some path tree nodes are more frequent than others.

Frequencies are assigned to the path tree nodes in *breadth first order*. An input parameter of the data generator specifies whether these frequencies are sorted in ascending order (i.e., the root node is the least frequent), descending order (i.e., the root node is the most frequent), or random order.

Figure 57 illustrates the different approaches for assigning frequencies to path tree nodes. The figure illustrates three path trees in which frequencies are assigned to nodes in ascending, descending, and random orders. The total number of XML elements that will be generated from these path trees is 31, and $z = 1$.

The path tree with tag name and node frequency information generated as outlined above fully specifies the structure of the XML data to be generated. The final step is for the data generator to generate the required XML document. The data generator performs a depth first traversal of the path tree and generates XML elements according to the specification given in the tree. To ensure that the generated XML document is well-formed, the data generator adds a top-level `<ROOT>` element to the document.

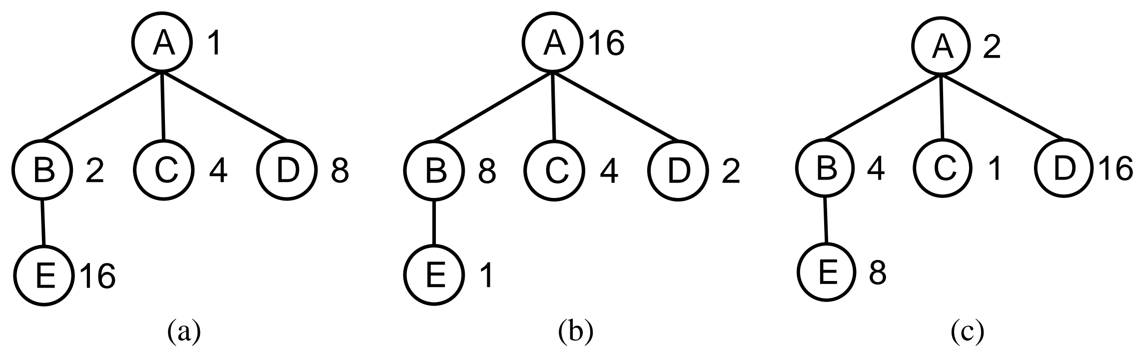


Figure 57: Assigning frequencies to path tree nodes in (a) ascending, (b) descending, and (c) random order