APPROXIMATION AND LEARNING TECHNIQUES IN DATABASE SYSTEMS

by

Min Wang

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jeffrey S. Vitter, Supervisor

_____
Pankaj K. Agarwal

_____
Lars Arge

_____
Jeffrey S. Chase

_____
Bala R. Iyer

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

1999

ABSTRACT

(Computer Science)

APPROXIMATION AND LEARNING TECHNIQUES IN DATABASE SYSTEMS

by

Min Wang

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Jeffrey S. Vitter, Supervisor

_____
Pankaj K. Agarwal

_____
Lars Arge

_____
Jeffrey S. Chase

_____
Bala R. Iyer

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

1999

Abstract

In this thesis, we study two important techniques that are widely used in database systems: approximation and learning.

Approximation has been an area of great interest and importance in database community. A classic example of using approximation in database systems is selectivity estimation. Another example is using approximation techniques to answer OLAP (On-Line Analytical Processing) queries, which is quite new and is initiated by our work.

In this thesis, we propose novel approximation techniques used in both selectivity estimation and approximate computation of OLAP aggregates. Our techniques are based on the powerful mathematical tool of wavelets and multiresolution analysis and are fundamentally different from traditional approaches. We present several methods that first attempt to use wavelets in the domain of database approximation. Our methods offer substantial improvements in accuracy and efficiency over existing methods.

We also develop efficient and scalable learning techniques for DBMSs to extract patterns from large databases in the context of data mining. Classification is an important and fundamental data mining problem. Almost all of the current classification algorithms have the restriction that the entire training set should fit in the internal memory to achieve efficiency. We present a novel classification algorithm (classifier) called MIND (MINing in Databases). MIND is scalable with respect to I/O efficiency, which is important since scalability is a key requirement for any data mining algorithm.

Acknowledgements

First of all, I would like to thank my advisor, Dr. Jeff Vitter, for guiding me through my thesis work. Jeff is very demanding. Looking back on my graduate years at Duke, one of the most unforgettable parts is Jeff's criticisms. They were hard to face at the beginning, but I learned the right ways to do research and the precise ways to present ideas when I went through them.

I was lucky to have the chance to work with Dr. Bala Iyer at IBM Santa Teresa Laboratory as a summer intern. Bala introduced me to several "real" database problems and has always been there whenever I have any questions.

I would like to thank Dr. Pankaj Agarwal for his valuable comments on my dissertation. I also would like to thank Dr. Lars Arge and Dr. Jeff Chase for serving on my committee.

My study at Duke was made enjoyable by many friends and fellow students. Thanks go to Joyce Chai, Hai Shao, Wei Li, Amy Xu, Yong Gao, Wendy Wang, Chong Xu, Yue Ma, and Paul Natsev.

I spent the last year of my graduate study at Stanford University as a visiting student. I would especially like to thank Dr. Rajeev Motwani for his generous help during my stay at Stanford.

I cannot express my gratitude to my husband Jingmin He and my son Taotao (Bill). Without their support and love, my dissertation would not have been possible.

I would like to give my deepest thanks to my parents and my brother for their endless support, love, and encouragement.

Credits

I would like to thank my co-authors for their collaboration while I was working on

portions of the material presented in this thesis.

The material in Chapter 4 is joint work with Yossi Matias and Jeff Vitter, and a version of the chapter appears as [53]. The material in Chapter 6 is joint work with Jeff Vitter and Bala Iyer, and a version of the chapter appears as [95]. The material in Chapter 7 is joint work with Jeff Vitter, and a version of the chapter appears as [94]. The material in Chapter 8 is joint work with Jeff Vitter and Bala Iyer, and a version of the chapter appears as [98]. The material in Chapter 9 is joint work with Bala Iyer and Jeff Vitter and a version of the chapter appears as [97].

My research was partly supported by an IBM Graduate Fellowship and Army Research Office MURI grant DAAH04–96–1–0013. I gratefully acknowledge the support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database technology has been used with great success in this information age. In many enterprises, large amount of data need to be maintained and managed in a very efficient way so that the business can remain competitive in today's rapidly changing market. Databases have been and still will be the major means that most enterprises can utilize to achieve their goals. However, database industry also faces challenges. It must be prepared to deal with new business requirements and continuously provide supplies of powerful database servers. For example, electronic commerce is the latest hot topic and it is generating huge revenue. All electronic commerce products require a powerful database back end that can provide quick answers to very complicated queries.

Enhancing and improving the database engines has always been the mission of all database companies. The quality of a database engine depends on several components, one of them is the query optimizer. To put it simply, all a database server can do is answering queries, and how efficient a query can be answered largely depends on how smart its query optimizer is, which in turn, depends on how accurate selectivity estimation can be done for the query. Accurate selectivity estimation is a very important problem and it is one of the major motivations for this thesis.

Another motivation for this thesis is to answer new types of queries efficiently. In recent years, many corporations have built or are building new unified decision-support databases called *data warehouses* on which users can carry out their business analysis. Users of data warehouses are typically interested in identifying business trends rather than looking at individual records in isolation. Processing OLAP (On-

Line Analytical Processing) queries efficiently is very important for data warehouses, and computation of data cube is a core operation for answering OLAP queries.

Both selectivity estimation and data cube computation can be viewed as representing the raw data in a way so that some useful information can be fetched to answer user queries efficiently. In principle, the required information can always be obtained from the raw data directly. The problem is efficiency. Since we usually cannot afford to process the raw data online, we need to preprocess and/or compress the raw data to obtain a transformed form through an approximation process, and from there we can do our online computation.

One of the major contributions of this thesis is to introduce a powerful mathematical tool—wavelet analysis—to do both selectivity estimation and data cube computation. Our methods offer substantial improvements in accuracy and efficiency over existing methods.

The third motivation for this thesis is to develop efficient and scalable learning techniques for DBMSs to extract patterns from large databases. We propose a scalable classification algorithm that can be built into RDBMSs directly.

We also investigate the problem of selectivity estimation for alphanumeric data.

In the following sections, we elaborate on each of the problems we address in this thesis in more details and highlight our approaches.

1.1 Wavelet-Based Approximation Techniques for Selectivity Estimation

Approximation techniques have become an area of increasing interest and importance in database community.

A classic example of using approximation techniques in database systems is *selectivity estimation*, that is, given a query P, we need to estimate the fraction of

records in the database that satisfy P. Several important components in a database management system (DBMS) require accurate estimation of the selectivity of a given query. For example, cost-based query optimizers use it to evaluate the costs of different query execution plans and choose the preferred one [76]. Other important motivations for good-quality selectivity estimations come from query profilers [63], approximate query processors [27], and load balancing mechanism [65].

Various histogram methods are used in today's DBMSs for numeric selectivity estimation. A traditional histogram consists of a number of buckets and approximates the data in one or more attributes of a relation by grouping attribute values into the buckets and storing summary statistics for each bucket. A well-known example is the equal-depth histogram [62]. The major drawback of the traditional histograms is that they are too simple to capture the many possible complex data distributions and the correlations among multiple attributes.

We introduce a new type of histograms based upon wavelets, a mathematical tool for hierarchical decomposition of functions, for selectivity estimation. Our study shows that the wavelet-based histogram offers substantial improvements in accuracy over the state-of-art histograms. One big advantage of our wavelet-based histogram is that it can handle any data distribution effectively. Another advantage is that it extends naturally to capture the joint distribution of multiple attributes.

1.2 Wavelet-Based Approximation Techniques for OLAP Applications

There has recently been an explosion of interest in the analysis of data in data warehouses in the field of On-Line Analytical Processing (OLAP). Data warehouses can be extremely large, yet obtaining quick answers to queries is important. In many situations, obtaining the exact answer to an OLAP query is prohibitively expensive in

terms of time and/or storage space. It can be advantageous to have fast, approximate query answers.

Selectivity estimation is completely internal to the database engine, and the quality (accuracy) of the approximation is observable by a user only indirectly in terms of the performance of the database system. Although a bad (inaccurate) estimation might cause a query to be executed slower, the engine still gives the correct query result. On the other hand, in OLAP applications, approximation plays a more direct role: the query result itself is an approximation. A bad estimation implies a query result with big error that might be unacceptable to a user. Therefore the nature and the quality of approximation become more salient.

Using the wavelet decomposition technique, we initiate the study of answering OLAP queries approximately. We present novel methods that provide approximate answers to high-dimensional OLAP aggregation queries in massive data sets in a time-efficient and space-efficient manner. We present I/O-efficient algorithms to construct an approximate and space-efficient representation of the underlying multidimensional data, based upon a multiresolution wavelet decomposition. In the on-line phase, each aggregation query can generally be answered using the compact representation in one I/O or a small number of I/Os, depending upon the desired accuracy.

1.3 Alphanumeric Selectivity Estimation

Almost all previous work in selectivity estimation dealt with the estimation of *numeric selectivity*, i.e., the query contains only numeric attributes. The general problem of estimating *alphanumeric selectivity* is much more difficult and has attracted attention only very recently, and the focus has been on the special case when only one column is involve [46, 41].

We consider the more general case when there are two correlated alphanumeric

columns. We develop efficient algorithms to build space-efficient storage structures that capture the correlations between the two columns.

1.4 Learning Techniques in Database Systems

Information technology has developed rapidly over the last three decades. To make decisions faster, many companies have combined data from various sources in relational databases [35]. The data contain patterns previously undeciphered that are valuable for business purposes. *Data mining* is the process of extracting valid, previously unknown, and ultimately comprehensible information from large databases and using it to make crucial business decisions.

Classification is an important and fundamental data mining problem [5]. It can be described as follows. The input data, also called the *training set*, consists of multiple examples (records), each having multiple attributes. Additionally, each example (record) is tagged with a special attribute called *class label*. The objective of classification is to analyze the input data and to develop an accurate description or model for each class using the attributes present in the training set. The model is used to classify future *test data* for which the class labels are unknown. It can also be used to develop a better understanding of each class in the data.

Classification has been studied extensively [99]. It is shown that immense data may be needed to train a classifier for good accuracy [21, 44]. However, most of the current classification algorithms have the restriction that the entire training set should fit in the internal memory. Even the state-of-the-art classifiers [55, 77] need an in-memory data structure of size $O(N)$, where N is the size of the training set, to achieve efficiency. In addition, all the existing classification algorithms take records in a file as the input training data, while most businesses store their data in relational databases [35]. Since extracting data from databases to files before running data

mining functions would require extra I/O costs, users of current data mining products as well as previous investigations [40, 39] have pointed to the need for the relational database managers to have these functions built in.

We propose a novel classification algorithm (classifier) called MIND (MINing in Database). MIND is truly scalable with respect to I/O efficiency, which is important since scalability is a key requirement for any data mining algorithm. MIND can be phrased in such a way that its implementation is very easy using the extended relational calculus SQL, and this in turn allows the classifier to be built into a relational database system directly.

1.5 Outline

This thesis is organized as follows: Chapter 2–7 study wavelet-based approximation techniques. In Chapter 2 we define the selectivity estimation problem and review previous work. Chapter 3 is a tutorial on wavelets and multiresolution analysis. In Chapter 4 we describe how to use wavelets to do selectivity estimation. In Chapter 5–7, we study wavelet-based techniques for efficiently and approximately answering typical OLAP aggregation queries. Chapter 5 introduces the problem and reviews previous work. In Chapter 6 we describe our method for approximately answering OLAP queries when the underlying multidimensional data are dense. In Chapter 7 we extend our techniques to solve the more difficult case when the underlying data are sparse.

In Chapter 8 we present our method for alphanumeric selectivity estimation in the presence of column correlations.

We describe our scalable classification algorithm MIND and its implementation in Chapter 9.

Chapter 2

Selectivity Estimation

A relational database consists of a collection of *relations*, or *tables*. Each row of a table represents one *record* (*tuple*) whose *attributes* are indicated by the *columns*. A Relational DataBase Management System (RDBMS) consists of a collection of relational databases and provides an environment that is both convenient and efficient for users to store, retrieve and analyze information. A user can access a database by issuing queries against the database tables.

One of the most important components of a RDBMS is its *optimizer* that does *query optimization*. Query optimization is the process of selecting the most efficient plan (called *query plan*) for executing a query. The plan selection usually involves (at least) two tasks. First, the given query is transformed (rewritten) to another equivalent form that can be executed more efficiently. The transform is done using relational algebra, which is not the topic of this thesis.

The second and most important task is to select a detailed plan for processing the query. There are usually many possible plans for executing the same query, and the optimizer needs to choose a good, if not the best, plan. By best we mean the one with the least cost. (We are only interested in cost-based optimizer since it is used by most commercial RDBMSs.) To this end, the optimizer has to evaluate the cost of each query plan. In most cases, computing the precise cost of a query plan is impossible without actually running the query according to the plan. Therefore, an optimizer will need to estimate the cost based on statistical information about the underlying data distributions of the relevant tables.

A good cost estimation for a query plan depends on selectivity estimation (or *filter*

factor estimation) in a crucial way. Suppose P is a partial query plan. Logically, P itself can be considered as a subquery, or simply a query. For any P, we need to do selectivity estimation, that is, we need to estimate the fraction of tuples that satisfy P. The performance of a query optimizer, and thus the whole query processing, depends essentially on how accurate the selectivity estimation is [52, 76].

Example 2.1 We consider a small example. Table 2.1 shows a relation in a database.

PID	*Price* ($)	*Sales*
p1	2	120
p2	3	100
p3	12	20
p4	8	80

Table 2.1: *PART*: a table (relation) in a database.

A simple SQL [1] query on this relation is:

`SELECT *`

`FROM` *PART*

`WHERE` $50 \leq Sales \leq 100$.

The selectivity of the above query is 50%, since two out of the four rows in the relation have *Sales* value between 50 and 100. □

In a RDBMS, selectivity estimation is usually done through two phases: an *offline phase* and an *online phase* (*run time*). During the offline phase, the data in the database are processed and some statistical/summary information is collected and stored as the database *catalogs*. (The space allowed for a catalog for one attribute of a relation is usually in the order of several hundred bytes.) During online phase, the selectivity for a given query is estimated based on the relevant catalog information.

[1] SQL is simply an implementation [17] of the relational calculus proposed in [19]. A few extensions have been done since then [85, 17].

The accuracy of the selectivity estimation depends on what kind of information is available in the database catalogs. The more information we have, the more accurate the estimation will be. However, database catalogs have to be small due to storage limitation. Besides, a large catalog would slow down the optimizer and thus downgrade the performance of the whole database system.

Now the problem becomes how to store statistical information about any table in small catalogs so that accurate selectivity estimation can be achieved. A lot of work has been done in this area. Basically, a new catalog organization would result in a new method for selectivity estimation.

In this chapter, we briefly review previous work and highlight their advantages and disadvantages. We then introduce our new approaches, one of them is based on a powerful mathematical tool called *wavelet decomposition*. We compare the different approaches and explain why our approaches are preferable in many cases. (The detailed comparison is done in Chapter 4 and Chapter 8.)

2.1 Previous Work

In this section we briefly review previous work on selectivity estimation.

2.1.1 System R

IBM's System R is among the earliest RDBMSs and its optimizer has set a tone for most modern optimizers [76]. The optimizers of several major database products still follow the general framework set by System R, that is, for any given query, the optimizer will do the following:

- Define a query plan space and an algorithm to search and enumerate all plans in the space.
- Define a cost model by which the cost of a query plan can be estimated.

- When a user issues a query, the optimizer will search through its query plan space and estimate the cost of each plan. Then the best (least-cost) plan is chosen and given to a *query execution engine* to run and the output is returned to the user.

However, the cost model of System R is not good enough because its most important component, selectivity estimation, is too simple. The selectivity estimation is done based on very simplified and unrealistic assumptions about the underlying data distribution. For example, the query optimizer assumes that each column of a table has uniform distribution and the columns are independent of each other. As a result, the selectivity estimation may be very inaccurate, and the optimizer could choose very inefficient query plans.

Unfortunately, many modern optimizers do the similar thing, that is, making unrealistic assumptions about the underlying data distributions. We will come back to this issue later.

2.1.2 Bucket Histograms

Histograms is the most widely used form to store statistical information of the data distributions in a database. Many different histograms have been proposed in the literature and some have been deployed in commercial RDBMSs. However, almost all previous histograms have one thing in common, that is, they all use buckets to partition the data, although in different ways.

A bucket-based histogram approximates the data in an attribute of a relation by grouping attribute values into *bucket* and estimating the attribute values and their frequencies based on the summary statistics maintained in each bucket. More specifically, a histogram of an attribute is an approximation of the frequency distribution of its values obtained as follows: the $(attribute\ value, frequency)$ pairs of the distribution

are partitioned into buckets; the frequency of each value in a bucket is approximated by the average of the frequencies of all values in the bucket; and the set of values in a bucket is usually approximated in some compact fashion as well.

The simplest histogram is the so-called *equi-width histogram*. It partitions the domain of an attribute into consecutive equal length subintervals. Each subinterval is characterized by just one parameter: the average frequency of the values in the subinterval.

The popular *equi-depth histogram* [62] partitions the interval between the minimum and maximum attribute value of a column into consecutive subintervals so that the total frequency of the attribute values for each subinterval is the same.

It is interesting to note that these two simple histograms (or their variants) are still used extensively in today's major commercial RDBMSs like Oracle, Informix, Microsoft SQL Server, Sybase, and Ingres.

Poosala et al. [67, 63] propose a taxonomy to capture all previously proposed histograms; new histograms types can be derived by combining effective aspects of different histogram methods. Among the histograms discussed in [67, 63], MaxDiff(V, A) histogram gives the best overall performance. MaxDiff(V, A) histogram uses attribute value V as "sort parameter" and *area* A as "source parameter" to choose the bucket boundaries. In a MaxDiff(V, A) histogram with β buckets, there is a bucket boundary between two adjacent attribute values if the difference between their areas is one of the $\beta - 1$ largest such differences. It is shown in [67, 63] that MaxDiff(V, A) histogram achieves better approximation than previous well-known methods like equi-depth and compressed histograms.

Poosala et al. [67] also mention other histogram techniques. For example, histograms based on minimizing the variance of the source parameter such as area have a performance similar to that of MaxDiff(V,A), but are computationally more ex-

pensive to construct. They also mention a spline-based technique, but it does not perform as well as MaxDiff(V,A).

All the histograms discussed so far follow the same general guideline, that is, they partition the underlying data in some particular fashion. The effectiveness and accuracy of various partition methods highly depend on the data distributions and query types. In general, using a specific partition method to capture many unpredictable data distributions is a hard job, especially for multidimensional data. It is desirable to study general methods to construct histograms whose usability is more robust.

2.1.3 Non-Histogram Techniques

Random sampling has attracted extensive study in database community in the last 15 years and there has been considerable amount of work done in sampling-based estimations [32, 33, 50, 49, 27]. All sampling-based techniques are based on the fact that a large data set can be represented well by a small random sample of the data elements. For selectivity estimation, sampling is a natural tool: at query optimization time, a random sample is collected and the selectivity of any given query is computed based on the sample. The advantages of sampling-based methods are obvious: no pre-computed and stored statistical information is needed, and it has probabilistic guarantees on the accuracy. However, obtaining a good sample at query optimization time is expensive, and sampling results cannot be preserved for later uses.

Another estimation technique is *probabilistic counting* and it is mainly used for estimating the size of projections. We refer the readers to [24, 78, 26] for more details.

Parametric estimation is a standard statistic technique that can also be used to estimate selectivity [18, 82]. However, this technique assumes that the data distribution resemble some well studied mathematical distribution that is usually characterized by

several parameters. The assumption is not true in most real database applications.

2.1.4 Multidimensional Data

When a query involves multiple attributes in a relation, the selectivity depends on these attributes' *joint data distribution*, that is, the frequencies of all combination of attribute values. The main challenge for histograms for multidimensional (multi-attribute) data is to capture the correlations among different attributes.

To simplify the selectivity estimation for multi-attribute queries, most commercial DBMSs make the *attribute value independent assumption*. Under such an assumption, a system only maintains histogram for each individual attribute, and the joint probabilities are derived by multiplying the individual probabilities. Real-life data rarely satisfy the attribute value independent assumption. Functional dependencies and various types of correlations among attributes are very common. Making the attribute value independent assumption in these cases results in very inaccurate estimation of the joint data distribution and poor selectivity estimation.

Muralikrishna and DeWitt [58] use an interesting spatial index partitioning technique to construct equi-depth histograms for multidimensional data. One drawback with this approach is that it considers each dimension only once during the partition. Poosala and Ioannidis [66] propose two effective alternatives. The first approach partitions the joint data distribution into mutually disjointed buckets and approximates the frequency and the value sets in each bucket in a uniform manner. Among this new class of histograms, the multidimensional MaxDiff(V,A) histograms computed using the MHIST-2 algorithm are most accurate and perform better in practice than previous methods [66]. The second approach uses the powerful *singular value decomposition* (SVD) technique from linear algebra, which is limited to handling two dimensions. Its accuracy depends largely on that of the underlying one-dimensional

histograms.

2.1.5 Selectivity Estimation in the Presence of Alphanumeric Correlations

Almost all previous work in selectivity estimation dealt with the estimation of numeric selectivity, i.e., the query contains only numeric variables. The general problem of estimating alphanumeric selectivity is much more difficult and has attracted attention only recently, due to the work of Krishnan, Vitter and Iyer [45, 46], and the focus has been on the special case when only one column is involved.

In practice, a query usually involves multiple columns. A survey of IBM's DB2 found that most SQL SELECT queries involve more than one attributes and queries containing two attributes are especially popular. Therefore, it is necessary to develop techniques for estimating selectivity when more than one columns is involved (we call this *multi-column estimation*). Accurate two predicate selectivity estimation is beneficial, yet no work has been done.

2.2 Our Approaches

A histogram can be considered as a compressed representation of the original data. In this sense, our approaches can also be categorized as histogram methods. However, our methods neither use buckets nor they try to partition the original data. Instead, our methods try to extract and summarize information from the original data. Although the traditional histograms also summarize the original data, they are *local*, while our methods are *hierarchical and global*.

In Chapter 4 we present our new type of histograms based upon a *multiresolution wavelet decomposition*. The multiresolution wavelet decomposition is a process of successive summarizing from local to global. Experiments show that our wavelet-

based histograms offer substantial improvements in accuracy over random sampling and other previous approaches.

Our wavelet-based techniques can be extended very naturally to multiple dimensions and thus provide an efficient way to capture the joint distribution of multiple attributes for selectivity estimation.

In Chapter 8, we study selectivity estimation when there are two correlated alphanumeric columns. We develop efficient algorithms to build storage structures that can capture the joint distribution very well and in the mean time fit in a database catalog.

Chapter 3

Wavelets and Multiresolution Analysis

In this chapter, we give a tutorial on wavelets and multiresolution analysis. The chapter is organized as follows: Section 3.1 is a brief introduction. In Section 3.2 we give a quick review of mathematical preliminaries that are necessary to fully understand this tutorial. We assume the readers are familiar with college calculus and basic linear algebra, and we only simply list some useful definitions and theorems without giving any proof. We do however give simple examples to elaborate on the concepts. Section 3.3 introduces the key concepts of wavelet and multiresolution analysis. In Section 3.4 we derive the formulas of wavelet decomposition and reconstruction for Haar wavelets. Section 3.5 contains a summary of Haar wavelets. Readers can skip Section 3.2–3.4, but they should at least read Section 3.5 to understand the material in Chapter 4–7.

3.1 Introduction

Wavelet analysis is a relatively new and powerful mathematical tool, with applications in a wide range of areas such as differential equations, numerical analysis, signal processing, image analysis, data compression, etc. Our work in part of the thesis is the first attempt to introduce this tool to the database community.

There are many books and survey papers on wavelets. Yet, most of the books do not suit the needs of computer science graduate students very well. Either they are too mathematical, with several chapters on preliminaries, or they are too non-mathematical, without even explaining, for example, how the Haar wavelet decomposition formulas are derived.

We want something in between. To use wavelets, we do not need to know Lebesgue measure, Fubini's theorem, and many other advanced mathematical concepts and results. To define multiresolution analysis, we do not need to use the Riesz basis because an orthonormal basis will serve the purpose equally well. On the other hand, to fully understand (say) the Haar wavelet decomposition formulas, it is better to know how those formulas are derived.

In this tutorial we do just that. We start with some necessary mathematical concepts, and give rigorous definition of wavelets and multiresolution analysis. We then derive the general wavelet decomposition and reconstruction formulas, and from there we obtain the explicit formulas for Haar wavelets that are used as an example in the later parts of this thesis.

3.2 Mathematical Preliminaries

Throughout this chapter, we shall use C, R, Z, and N to denote the sets of complex, real, integer and natural numbers, respectively. We use F to denote either R or C. Note that F is a field.

Let E be a set equipped with a binary operation called *addition*. The elements of E are called *vectors*. As usual, we use $+$ to denote the addition operation. Let $\cdot$ be a *scalar multiplication* that takes any scalar $\alpha \in F$ and any vector $x \in E$ and maps them to some vector $\alpha \cdot x \in E$. We usually omit $\cdot$ and simply write αx.

Definition 3.1 Let E be a set of vectors and F a field, as described above. We call E a *vector space* over F if the following conditions are satisfied for all $x, y, z \in E$ and $\alpha, \beta \in F$:

1. Commutativity: $x + y = y + x$.

2. Associativity: $(x + y) + z = x + (y + z)$, $(\alpha\beta)x = \alpha(\beta x)$.

3. Distributivity: $\alpha(x+y) = \alpha x + \alpha y$, $(\alpha+\beta)x = \alpha x + \beta x$.

4. Additive identity: There exists $0 \in E$ such that $x + 0 = x$, for all $x \in E$.

5. Additive inverse: For any $x \in E$, there exists a $-x \in E$ such that $x + (-x) = 0$.

6. Multiplicative identity: For all $x \in E$, $1 \cdot x = x$, where 1 is the multiplicative identity in F.

Definition 3.2 A subset $M \subset E$ is a *subspace* of the vector space E over F if

1. For all $x, y \in M, x + y \in M$.

2. For all $x \in M, \alpha \in F, \alpha x \in M$.

Example 3.1 Let $E = C^n$, where the elements of E are complex vectors of n components. C^n is a vector space over F with the vector addition and scalar multiplication defined in the standard way. For example, two vectors $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ add together and the sum is $x + y = (x_1 + y_1, \ldots, x_n + y_n)$.

R^n is a subspace of C^n. □

From now on, we always use E to denote a vector space over the field F.

Definition 3.3 Given $S \subset E$, the *span* of S is the subspace of E consisting of all linear combinations of vectors in S. That is,

$$span(S) = \left\{ \sum_{i=1}^{n} \alpha_i x_i \mid \alpha_i \in F, x_i \in E, n \in N \right\}.$$

Example 3.2 Let $e_i \in R^n$ where the ith component of e_i is 1 and all other components are 0. Let $S = \{e_1, e_2\}$. Then

$$span(S) = \{(x_1, x_2, 0, \ldots, 0) \mid x_1, x_2 \in R\}.$$

□

Definition 3.4 Vectors $x_1, x_2, \ldots, x_n$ are *linearly independent* if $\sum_{i=1}^{n} \alpha_i x_i = 0$ is true only if $\alpha_i = 0$, for all i. Otherwise these vectors are *linearly dependent*. An infinite set of vectors S are linearly independent if every finite subset of vectors of S are linearly independent.

Example 3.3 The vectors $e_1, e_2, \ldots, e_n \in R^n$ are linearly independent. However, the vectors e_1, e_2, and $e_1 + e_2$ are linearly dependent. □

Definition 3.5 A subset $\{x_1, x_2, \ldots, x_n\} \subset E$ is called a *basis* for E, if

$$E = span(\{x_1, x_2, \ldots, x_n\}),$$

and $x_1, x_2, \ldots, x_n$ are linearly independent, and we say E has *dimension* n. If E contains an infinite linearly independent set of vectors, we say E is *infinite-dimensional*.

Example 3.4 The subset $\{e_1, e_2, \ldots, e_n\}$ forms a basis for R^n, and thus R^n has dimension n. However, $R^\infty = \{(x_1, x_2, \ldots)\}$ is an infinite-dimensional vector space. □

Definition 3.6 An *inner product* on a vector space E over F is a complex-valued function $\langle \cdot, \cdot \rangle$, defined on $E \times E$, that has the following properties for all $x, y, z \in E$ and $\alpha, \beta \in F$:

1. $\langle x, y \rangle = \overline{\langle y, x \rangle}$, where $\overline{w}$ is the complex conjugate of w for any w.
2. $\langle \alpha x + \beta y, z \rangle = \alpha \langle x, z \rangle + \beta \langle y, z \rangle$.
3. $\langle x, x \rangle \geq 0$, and $\langle x, x \rangle = 0$ if and only if $x = 0$.

Example 3.5 Consider the vector space R^n. For vectors $x, y \in R^n$, we can define their inner product as follows:

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

□

A vector space equipped with an inner product is called an *inner product space* or *linear space*.

Definition 3.7 A *norm* on E is defined as a real-valued function $\|\cdot\|$ that maps any vector to a real number such that for any $x, y \in E$ and $\alpha \in F$, we have

1. $\|x + y\| \leq \|x\| + \|y\|$,
2. $\|\alpha x\| = |\alpha|\|x\|$, and
3. $\|x\| = 0$ if and only if $x = 0$.

Theorem 3.1 *Let $\langle \cdot, \cdot \rangle$ be an inner product defined on E. A norm on E can be defined as follows: for any $x \in E$, $\|x\| = \sqrt{\langle x, x \rangle}$. The distance between two vectors x and y is thus $\|x - y\|$.*

Example 3.6 With the inner product defined as in Example 3.5, we obtain the L_2-norm on the space R^n:

$$\|x\| = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}.$$

$\|x\|$ is also called the length of vector x. □

From now on we assume that E is equipped with an inner product and the norm is defined as in Theorem 3.1.

Definition 3.8 Two vectors x and y are *orthogonal* if $\langle x, y \rangle = 0$, and we denote this fact by $x \perp y$. We also say x is orthogonal to y.

A vector x is said to be orthogonal to a set of vectors S if x is orthogonal to every vector in S, and we denote this by $x \perp S$.

Two sets of vectors S_1, S_2 are called orthogonal if every vector in S_1 is orthogonal to every vector in S_2, and we denote this by $S_1 \perp S_2$.

A set of vectors $\{x_1, x_2, \ldots\}$ is called orthogonal if $x_i \perp x_j$ when $i \neq j$.

Example 3.7 In R^n, e_i and e_j are orthogonal when $i \neq j$. □

Definition 3.9 Let $\{x_1, x_2, \ldots\}$ be an orthogonal set (of vectors). If all vectors in the set are normalized, that is, $\|x_i\| = 1$ for all i, then we call the set an *orthonormal system.*

Example 3.8 In R^n, $\{e_1, e_2, \ldots, e_n\}$ is an orthonormal system. □

It is easy to verify that vectors in an orthonormal system are linearly independent.

Definition 3.10 An orthonormal system in a vector space E is an *orthonormal basis* if it spans E.

Example 3.9 In R^n, $\{e_1, e_2, \ldots, e_n\}$ is an orthonormal basis. □

Definition 3.11 A sequence of vectors $\{x_k\} \subset E$ *converges* to $x \in E$ if $\|x_k - x\| \to 0$ as $k \to \infty$. The value x is called the limit of the sequence $\{x_k\}$, or simply a limit.

Example 3.10 The sequence $\{(1 - \frac{1}{k}, \frac{1}{k^2}, \ldots, \frac{1}{k^n})\}$ converges to $(1, 0, \ldots, 0)$. □

Definition 3.12 A sequence of vectors $\{x_k\}$ is a *Cauchy sequence*, if $\|x_k - x_l\| \to 0$, when $k, l \to \infty$.

Example 3.11 Let us consider the simple space R^n where $n = 1$. The sequence $\{\frac{1}{k}\}$ is a Cauchy sequence because $\|\frac{1}{k} - \frac{1}{l}\| \to 0$, when $k, l \to \infty$. □

Definition 3.13 If every Cauchy sequence in E converges to a vector in E, then E is called *complete.*

Example 3.12 The space R, considered as a vector space over itself, is complete because from elementary calculus we know that every Cauchy sequence in R converges to a real number. □

Definition 3.14 A complete inner product space is called a *Hilbert space*.

Example 3.13 Both C^n and R^n are Hilbert spaces. □

Definition 3.15 A Hilbert space is *separable* if it contains a countable orthonormal basis.

Definition 3.16 Let S be a subset of E. If any convergent sequence of vectors in S converges to a limit in S, then S is called a *closed set*.

Example 3.14 In R, the interval $[0, 1]$ is a closed set, but $(0, 1)$ is not. □

Theorem 3.2 *A closed subspace of a separable Hilbert space is separable.*

Definition 3.17 Let S be a subspace of a Hilbert space E. The *orthogonal complement* of S in E, denoted $S^{\perp}$, is defined as

$$S^{\perp} = \{x \in E \mid x \perp S\}.$$

Example 3.15 Consider the Hilbert space R^n. Let $S = span(\{e_1, \ldots, e_k\})$ for some $1 < k < n$. Then it is easy to see that $S^{\perp} = span(\{e_{k+1}, \ldots, e_n\})$. □

Theorem 3.3 *Let S be a closed subspace of a Hilbert space E. Any vector $y \in E$ can be written in the form*

$$y = v + w,$$

where $v \in S$ and $w \in S^{\perp}$ and the representation is unique. We say E is the direct sum of the subspace S and its orthogonal complement and write

$$E = S \oplus S^{\perp}.$$

Example 3.16 Continuing Example 3.15, we certainly can write

$$R^n = S \oplus S^{\perp}.$$

□

Example 3.17 Now let's turn to a Hilbert space that is the major focus of this chapter. We will see that one of the major tasks of wavelet analysis is to find a orthonormal basis for that space.

Let $L_2(R)$ be the set of functions defined on R that are square integrable, that is,

$$L_2(R) = \left\{ f : R \to C \;\middle|\; \int_{x \in R} |f(x)|^2 dx < \infty \right\},$$

where the inner product on $L_2(R)$ is given by

$$\langle f, g \rangle = \int_{x \in R} f(x)\overline{g(x)}dx,$$

and the norm is

$$\|f\| = \sqrt{\langle f, f \rangle} = \left(\int_{x \in R} |f(x)|^2 dx \right)^{1/2}.$$

It is well-known that $L_2(R)$ is a Hilbert space. (For a proof, please see any book on functional analysis, for example [71].) □

3.3 Wavelet and Multiresolution Analysis

Before we introduce the major definitions of this chapter, let's define two more concepts: *translation and dilation.* These are two function operators. For any function f, a translation by h produces a new function $(\tau_h f)(t) = f(t - h)$, and dilation by r produces another function $(\rho_r f)(t) = f(rt)$.

Definition 3.18 A *wavelet* is a function $\psi(t) \in L_2(R)$ such that the family of functions

$$\psi_{j,k}(t) = 2^{-j/2}\psi(2^{-j}t - k), \qquad j, k \in Z,$$

is an orthonormal basis in $L_2(R)$.

From Definition 3.18, we can see that an orthonormal basis for $L_2(R)$ can be obtained from a wavelet by translation and dilation. Loosely speaking, the central theme of wavelet analysis can be summarized as follows: Find a wavelet (and thus an orthonormal basis) for $L_2(R)$ and use it to represent any function in $L_2(R)$. Based on that (unique) representation, study the function itself.

The problem is: How can we find a wavelet? The next concept will help.

Definition 3.19 A *multiresolution analysis* is a sequence $\{V_j\}_{j\in Z}$ of closed subspaces of $L_2(R)$ with the following properties:

1. $\ldots \subset V_{-1} \subset V_0 \subset V_1 \subset \ldots$.
2. $span\left(\bigcup_{j\in Z} V_j\right) = L_2(R)$, where $span(S)$ (the *span* of S) is the set of all the linear combinations of the elements in set S.
3. $\bigcap_{j\in Z} V_j = \{0\}$.
4. $f(x) \in V_j$ if and only if $f(2^{-j}x) \in V_0$.
5. $f \in V_0$ if and only if $f(x-m) \in V_0$ for all $m \in Z$.
6. There exists a function $\phi \in V_0$, called a *scaling function*, such that the system $\{\phi(t-m)\}_{m\in Z}$ is an orthonormal basis in V_0.

The importance of a multiresolution analysis can be seen from the following theorem, which states that from a multiresolution analysis, we can find an orthonormal basis for $L_2(R)$.

Theorem 3.4 *Suppose we have a multiresolution analysis $\{V_j\}_{j\in Z}$. There exists an orthonormal basis for $L_2(R)$:*

$$\psi_{j,k}(t) = 2^{-j/2}\psi(2^{-j}t - k), \quad j,k \in Z,$$

such that $\{\psi_{j,k}\}_{k\in Z}$ is an orthonormal basis for W_j, where W_j is the orthogonal complement of V_j in V_{j+1}.

The standard proof of the above theorem includes an outline on how to construct the orthonormal basis $\psi_{j,k}$. We briefly reproduce it here.

First, we define the orthogonal complement of V_j in V_{j+1} as W_j, that is,

$$V_{j+1} = V_j \oplus W_j. \tag{3.1}$$

Repeating the process and using property (2) of Definition 3.19, we obtain

$$L_2(R) = \bigoplus_{j \in Z} W_j. \tag{3.2}$$

Based on the scaling property of V_j, we can deduce a similar scaling property for W_j, that is,

$$f(t) \in W_j \Longleftrightarrow f(2^{-j}t) \in W_0. \tag{3.3}$$

If we can find a $\psi(t) \in W_0$, such that $\{\psi(t-k)\}_{k \in Z}$ is an orthonormal basis for W_0, then by the scaling property, we obtain an orthonormal basis $\{\psi_{j,k}\}_{k \in Z}$ for W_j. From properties (2–3) of Definition 3.19 and (3.2), we conclude that $\{\psi_{j,k}\}_{j,k \in Z}$ is an orthonormal basis for $L_2(R)$, and thus we have constructed a wavelet.

From the above discussion, we see that the key is to find a *mother wavelet* function ψ. For a general discussion on how to construct such a mother wavelet, we refer the readers to [43, 80].

The following example illustrates the construction of the Haar wavelet, which is the simplest and earliest wavelet.

Example 3.18 Let V_j be the set of all functions in $L_2(R)$ that are constant on intervals of the form $[2^{-j}k, 2^{-j}(k+1)]$, where $k \in Z$. The set $\{V_j\}_{j \in Z}$ is a multiresolution analysis with the following scaling function:

$$\phi(t) = \begin{cases} 1 & \text{if } x \in [0,1), \\ 0 & \text{otherwise.} \end{cases}$$

From this scaling function, we can construct a wavelet ψ:

$$\psi(t) = \phi(2t) - \phi(2t-1) = \begin{cases} 1 & 0 \le t < 1/2, \\ -1 & 1/2 \le t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

From this "mother wavelet", we obtain an orthonormal basis $\{\psi_{j,k}(t)\}_{j,k \in Z}$ in $L_2(R)$. The function ψ is called the Haar wavelet. In next section, we discuss the Haar wavelet in more detail. □

3.4 Wavelet Decomposition and Reconstruction

Suppose we have a multiresolution analysis as defined in Definition 3.19, and based on it we have constructed a wavelet ψ. For any function $f \in L_2(R)$, we can obtain its (orthogonal) projection onto V_j:

$$P_j f = \sum_{k \in Z} \langle f, \phi_{j,k} \rangle \phi_{j,k}. \tag{3.4}$$

Similarly, we have the projection of f onto W_j, given by

$$Q_j f = \sum_{k \in Z} \langle f, \psi_{j,k} \rangle \psi_{j,k}. \tag{3.5}$$

Since $V_{j+1} = V_j \oplus W_j$, it follows that

$$P_{j+1} f = P_j f + Q_j f. \tag{3.6}$$

The following theorem shows that function f can be approximated by the projections $P_j f$.

Theorem 3.5 *For all $x \in R$,*

$$\lim_{j \to \infty} P_j f(x) = f(x). \tag{3.7}$$

Thus, any function $f(x)$ can be approximated as closely as we want by the projection $P_j f(x)$ for large enough j.

Strictly speaking, the convergence in the above theorem is true only for almost every $x \in R$. However, the set of points for which the convergence does not hold has *Lebesgue measure* zero, and we omit any formal discussion on "Lebesgue measure" and related concepts.

We define the sequence

$$c^j = \{c_{j,k} = \langle f, \phi_{j,k}, \rangle \mid k \in Z\}, \quad j \in Z. \tag{3.8}$$

Our goal is to compute the sequence c^j for arbitrary j and thus be able to approximate f to any desired accuracy.

From property (6) of Definition 3.19, we know that $\phi \in V_0$. Property (4) implies that $\phi(\frac{\cdot}{2}) \in V_{-1}$. Since V_j are subspaces, we have $\frac{1}{2}\phi(\frac{\cdot}{2}) \in V_{-1}$. So we can write

$$\frac{1}{2}\phi\left(\frac{1}{2}x\right) = \sum_{n \in Z} \alpha_n \phi(x+n), \tag{3.9}$$

where

$$\alpha_n = \left\langle \frac{1}{2}\phi\left(\frac{1}{2}x\right), \phi(x+n) \right\rangle = \frac{1}{2}\int_R \phi\left(\frac{1}{2}x\right)\overline{\phi(x+n)}dx. \tag{3.10}$$

(3.10) is valid for the scales V_{-1} and V_0. In the general case, we have

$$\begin{aligned} \phi_{j-1,k}(x) &= 2^{\frac{1}{2}(j-1)}\phi(2^{j-1}x - k) \\ &= 2^{\frac{1}{2}(j+1)}\frac{1}{2}\phi\left(\frac{1}{2}(2^j x - 2k)\right) \\ &= 2^{\frac{1}{2}(j+1)}\sum_{n \in Z} \alpha_n \phi(2^j x - 2k + n). \end{aligned}$$

Thus we obtain

$$\phi_{j-1,k}(x) = \sqrt{2}\sum_{n \in Z} \alpha_n \phi_{j,2k-n}(x), \quad \text{for } j,k \in Z. \tag{3.11}$$

Similarly, $\frac{1}{2}\psi(\frac{\cdot}{2}) \in W_{-1} \subset V_0$, and we have

$$\frac{1}{2}\psi\left(\frac{1}{2}x\right) = \sum_{n \in Z} \beta_n \phi(x+n), \tag{3.12}$$

and

$$\psi_{j-1,k}(x) = \sqrt{2} \sum_{n \in Z} \beta_n \phi_{j,2k-n}(x), \quad j,k \in Z. \tag{3.13}$$

Now we can compute the sequence c^j as follows:

$$\begin{aligned} c_{j-1,k} &= \langle f, \phi_{j-1,k} \rangle \\ &= \langle f, \sqrt{2} \sum_{n \in Z} \alpha_n \phi_{j,2k-n} \rangle \\ &= \sqrt{2} \sum_{n \in Z} \overline{\alpha_n} c_{j,2k-n}. \end{aligned} \tag{3.14}$$

When j is fixed, the right-hand side of (3.14) is the following: Compute the convolution of the two sequences $\{\sqrt{2}\overline{\alpha_n}\}$ and $\{c_{j,n}\}$, and retain the entries in the even places. Thus, if we have N significant terms in the sequence c^j at level j, we can compute a "blurred" version of c^j that contains only $N/2$ terms at level $(j-1)$.

We can compute c^{j-1} from c^j according to (3.14). The process is a resolution from V_j to V_{j-1}, which is one part of the resolution from level j to level $(j-1)$. Since we have $V_j = V_{j-1} \oplus W_{j-1}$, The other part of the resolution is from V_j to W_{j-1}, which we derive below.

From (3.5), we have

$$Q_j f(x) = \sum_{n \in Z} d_{j-1,k} \psi_{j-1,k}(x), \tag{3.15}$$

where $d_{j-1,k} = \langle f, \psi_{j-1,k} \rangle$. From (3.12), we obtain

$$\begin{aligned} d_{j-1,k} &= \langle f, \psi_{j-1,k} \rangle \\ &= \langle f, \sqrt{2} \sum_{n \in Z} \beta_n \phi_{j,2k-n} \rangle \\ &= \sqrt{2} \sum_{n \in Z} \overline{\beta_n} c_{j,2k-n}. \end{aligned} \tag{3.16}$$

The following example illustrates the Haar wavelet decomposition.

Example 3.19 Consider the Haar wavelet in Example 3.18. It is easy to verify the following:

$$\frac{1}{2}\phi\left(\frac{1}{2}x\right) = \frac{1}{2}\phi(x) + \frac{1}{2}\phi(x-1), \tag{3.17}$$

and

$$\frac{1}{2}\psi\left(\frac{1}{2}x\right) = \frac{1}{2}\phi(x) - \frac{1}{2}\phi(x-1). \tag{3.18}$$

According to (3.9) and (3.12), we obtain the two sequences $\{\alpha_n\}$ and $\{\beta_n\}$:

$$\alpha_n = \begin{cases} \frac{1}{2} & n = -1, 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\beta_n = \begin{cases} -\frac{1}{2} & n = -1, \\ \frac{1}{2} & n = 0, \\ 0 & \text{otherwise.} \end{cases}$$

From (3.14) and (3.16), we obtain

$$c_{j-1,k} = \frac{\sqrt{2}(c_{j,2k} + c_{j,2k+1})}{2}. \tag{3.19}$$

$$d_{j-1,k} = \frac{\sqrt{2}(c_{j,2k} - c_{j,2k+1})}{2}. \tag{3.20}$$

Haar wavelet decomposition is thus a very simple process: Each term $c_{j-1,k}$ is a moving average of two adjacent terms of the sequence c^j times the constant factor $\sqrt{2}$, and each $d_{j-1,k}$ (called detail) is one half of the difference of the same two adjacent terms times the same factor. □

The wavelet decomposition is a process of computing $d^{j-1}, d^{j-2}, \ldots, d^{j-l}$, and c^{j-l} from the sequence c^j, for some l ($l < j$). Now we consider the reverse transformation, that is, the reconstruction of c^j from $d^{j-1}, d^{j-2}, \ldots, d^{j-l}$, and c^{j-l}.

From (3.6), we have

$$P_j f = P_{j-1} f + Q_{j-1} f.$$

Plugging in the general decompositions (3.4) and (3.5) on both sides of the above equation, and applying (3.11) and (3.13), we have

$$\begin{aligned}
\sum_{l \in Z} c_{j,l} \phi_{j,l}(x) &= \sum_{k \in Z} c_{j-1,k} \phi_{j-1,k}(x) + \sum_{k \in Z} d_{j-1,k} \psi_{j-1,k}(x) \\
&= \sum_{k \in Z} c_{j-1,k} \left(\sum_{n \in Z} \sqrt{2} \alpha_n \phi_{j,2k-n}(x) \right) + \sum_{k \in Z} d_{j-1,k} \left(\sum_{n \in Z} \sqrt{2} \beta_n \phi_{j,2k-n}(x) \right) \\
&= \sum_{l \in Z} \left(\sum_{k \in Z} (\sqrt{2} c_{j-1,k} \alpha_{2k-l} + \sqrt{2} d_{j-1,k} \beta_{2k-l}) \right) \phi_{j,l}(x).
\end{aligned}$$

Hence,

$$c_{j,l} = \sqrt{2} \sum_{k \in Z} (c_{j-1,k} \alpha_{2k-l} + d_{j-1,k} \beta_{2k-l}). \tag{3.21}$$

Example 3.20 Continuing from Example 3.19 and applying (3.21) , we obtain the following reconstruction formulas for Haar wavelets:

$$\begin{aligned}
c_{j,2k} &= \frac{c_{j-1,k} + d_{j-1,k}}{\sqrt{2}}; \\
c_{j,2k+1} &= \frac{c_{j-1,k} - d_{j-1,k}}{\sqrt{2}}.
\end{aligned}$$

□

3.5 Haar Wavelets

We summarize the basic facts and formulas about Haar wavelets. Suppose S is an array of N values:

$$S = [S(0), S(1), \ldots, S(N-1)].$$

To apply the Haar wavelet decomposition to S, we assume N is a power of 2. (Otherwise we can pad S with dummy zero entries.) We first normalize S to obtain a sequence S_j at level j:

$$S_j = [S_{j,0}, S_{j,1}, \ldots, S_{j,N-1}],$$

where $j = \log N$ and $S_{j,k} = S(k)/\sqrt{2^j}$.

The decomposition of array S can be done using the following recursive formulas:

$$\begin{aligned} S_{i,k} &= \frac{\sqrt{2}(S_{i+1,2k} + S_{i+1,2k+1})}{2} \\ \widehat{S}_{i,k} &= \frac{\sqrt{2}(S_{i+1,2k} - S_{i+1,2k+1})}{2} \end{aligned}$$

for $0 \leq i \leq j-1$ and $0 \leq k \leq 2^i - 1$. The decomposition results is the sequence

$$[S_{0,0},\ \widehat{S}_{1,0},\ \widehat{S}_{1,1}, \ldots, \widehat{S}_{j-1,2^{j-1}-1}].$$

The entries in the sequence are called *wavelet coefficients*.

The reconstruction is a reverse process to the decomposition, and can be done using the following backward recursive formulas:

$$\begin{aligned} S_{i,2k} &= \frac{\sqrt{2}(S_{i-1,k} + \widehat{S}_{i-1,k})}{2}; \\ S_{i,2k+1} &= \frac{\sqrt{2}(S_{i-1,k} - \widehat{S}_{i-1,k})}{2}, \end{aligned}$$

for $1 \leq i \leq j$ and $0 \leq k \leq 2^{i-1} - 1$.

The following pseudocode procedure accomplishes the decomposition, transforming a set of values into wavelet coefficients in place.

Procedure *Decomposition*(s : **array**$[0 \ldots 2^j - 1]$ **of reals**)
  $s = s/\sqrt{2^j}$; //normalize input values
  $g = 2^j$;

while $g \geq 2$ **do**
$DecompositionStep(s[0 \ldots g-1])$;
$g = g/2$;

Procedure $DecompositionStep(s : \textbf{array}[0 \ldots 2^j - 1]$ **of reals**$)$
for $i = 0$ **to** $2^j/2 - 1$ **do**
$s'[i] = s[2i] + s[2i+1]/\sqrt{2}$;
$s'[i + 2^j/2] = s[2i] - s[2i+1]/\sqrt{2}$;
$s = s'$;

The following pseudocode procedure reconstruct the original data from the wavelet coefficients.

Procedure $Reconstruction(s : \textbf{array}[0 \ldots 2^j - 1]$ **of reals**$)$
$s = s/\sqrt{2^j}$;
$g = 2$;
while $g \leq 2^j$ **do**
$ReconstructionStep(s[0 \ldots g-1])$;
$g = 2g$;
$s = s'\sqrt{2^j}$; //undo normalization

Procedure $ReconstructionStep(s : \textbf{array}[0 \ldots 2^j - 1]$ **of reals**$)$
for $i = 0$ **to** $2^j/2 - 1$ **do**
$s'[2i] = s[i] + s[2^j/2 + i]/\sqrt{2}$;
$s'[2i+1] = s[i] - s[2^j/2 + i]/\sqrt{2}$;
$s = s'$;

The wavelet decomposition (reconstruction) is very efficient computationally, requiring only $O(N)$ CPU time for an array of N values.

The above decomposition and reconstruction processes include the normalization operation. The input array has been normalized in the very beginning. In the following, we illustrate the process of Haar wavelet decomposition using a concrete example. To simplify the presentation, we ignore the normalization factors.

Example 3.21 Suppose we have a one-dimensional array of $N = 8$ data items:

$$S = [2,\ 2,\ 0,\ 2,\ 3,\ 5,\ 4,\ 4].$$

We perform a wavelet transform on it. We first average the array values, pairwise, to get the new lower-resolution version with values

$$[2,\ 1,\ 4,\ 4].$$

That is, the first two values in the original array (2 and 2) average to 2, and the second two values 0 and 2 average to 1, and so on. Clearly, some information is lost in this averaging process. To recover the original array from the four averaged values, we need to store some *detail coefficients*, which capture the missing information. Haar wavelets store one half of the pairwise differences of the original values as detail coefficients. In the above example, the four detail coefficients are $(2-2)/2 = 0$, $(0-2)/2 = -1$, $(3-5)/2 = -1$, and $(4-4)/2 = 0$. It is easy to see that the original values can be recovered from the averages and differences.

We have succeeded in decomposing the original array into a lower-resolution version of half the number of entries and a corresponding set of detail coefficients. By repeating this process recursively on the averages, we get the full decomposition:

Resolution	Averages	Detail Coefficients
8	$[2,\ 2,\ 0,\ 2,\ 3,\ 5,\ 4,\ 4]$	
4	$[2,\ 1,\ 4,\ 4]$	$[0,\ -1,\ -1,\ 0]$
2	$[1\frac{1}{2},\ 4]$	$[\frac{1}{2},\ 0]$
1	$[2\frac{3}{4}]$	$[-1\frac{1}{4}]$

We have obtained the *wavelet transform* (also called *wavelet decomposition*) of the original eight-value array, which consists of the single coefficient representing the overall average of the original values, followed by the detail coefficients in the order of increasing resolution:

$$\widehat{S} = [2\tfrac{3}{4},\ -1\tfrac{1}{4},\ \tfrac{1}{2},\ 0,\ 0,\ -1,\ -1,\ 0]. \tag{3.22}$$

No information has been gained or lost by this process. The original array has eight values, and so does the transform. Given the transform, we can reconstruct the exact values by recursively adding and subtracting the detail coefficients from the next-lower resolution. □

Chapter 4

Wavelet-Based Histograms for Selectivity Estimation

In this chapter, we describe how to build histograms using a multiresolution wavelet decomposition.

This chapter is organized as follows: In Section 4.1 we introduce necessary notations to simplify our discussions. We present the details of constructing a wavelet-based histogram in Section 4.2 and show how to use it in the online phase for selectivity estimation in Section 4.3. In Section 4.4 we extend our results to multidimensional case. We report our experimental results in Section 4.5. Section 4.6 contains some concluding remarks.

4.1 Notations

The set of predicates we are going to consider is the set of selection queries, in particular, *range predicates* of the form $a \leq X \leq b$, where X is a non-negative attribute of the domain of a relation R, and a and b are constants. The set of *equal predicates* is the subset of the range predicates that have $a = b$. The set of *one-side range predicates* is the special case of range predicates in which $a = -\infty$ or $b = \infty$.

We adopt the notations in [67] to describe the data distributions and various histograms. The *domain* $D = \{0, 1, 2, \ldots, N-1\}$ of an attribute X is the set of all possible values of X. The value set $V \subseteq D$ consists of the n distinct values of X that are actually present in relation R. Let $v_1 < v_2 < \cdots < v_n$ be the n values of V. The *spread* s_i of v_i is defined as $s_i = v_{i+1} - v_i$. (We take $s_0 = v_1$ and $s_n = 1$.) The *frequency* f_i of v_i is the number of tuples in which X has value v_i. The *cumulative*

frequency c_i of v_i is the number of tuples $t \in R$ with $t.X \leq v_i$; that is, $c_i = \sum_{j=1}^{i} f_j$. The data distribution of X is the set of pairs $\mathcal{T} = \{(v_1, f_1), (v_2, f_2), \ldots, (v_n, f_n)\}$. The *cumulative data distribution* of X is the set of pairs $\mathcal{T}^{\mathcal{C}} = \{(v_1, c_1), (v_2, c_2), \ldots, (v_n, c_n)\}$. The *extended cumulative data distribution* of X, denoted by $\mathcal{T}^{\mathcal{C}^+}$, is the cumulative data distribution of $\mathcal{T}^{\mathcal{C}}$ extended over the entire domain D by assigning zero frequency to every value in $D - V$.

4.2 Wavelet-Based Histograms

In this section we describe how to construct a wavelet-base histogram. We first give a high level description of the construction algorithm. We then describe the algorithm steps in details.

4.2.1 Building A Histogram: A High Level Description

Building a wavelet-based histogram is a three-step procedure, and we outline them as follows:

Preprocessing In a preprocessing step, we form the *extended cumulative data distribution* $\mathcal{T}^{\mathcal{C}^+}$ of the attribute X, from the original data or from a random sample of the original data.

Wavelet Decomposition We compute the wavelet decomposition of $\mathcal{T}^{\mathcal{C}^+}$, obtaining a set of N wavelet coefficients.

Pruning We keep only the m most significant wavelet coefficients, for some m that corresponds to the desired storage usage. The choice of which m coefficients we keep depends upon the particular thresholding method we use.

After applying the above algorithm, we obtain m wavelets coefficients. The values of these coefficients, together with their positions (indices), are stored and serve as a

histogram for reconstructing the approximate data distribution in the on-line phase (query phase). To compute the estimate for the number of tuples whose X value is in the range $a \leq X \leq b$, we reconstruct the approximate values for b and $a-1$ in the extended cumulative distribution function and then subtract them.

Further benefits can be obtained by quantizing the wavelet coefficients and entropy-encoding the quantized coefficients. In this thesis, we restrict ourselves to choosing m complete coefficients, so as to facilitate direct comparisons with previous work.

Next we discuss each step in greater details.

4.2.2 Preprocessing

In the preprocessing step, we compute $\mathcal{T}$. The extended cumulative data distribution $\mathcal{T}^{c+}$ can be easily computed from $\mathcal{T}$. Exact computation of $\mathcal{T}$ can be done by maintaining a counter for each distinct attribute value in V. When the cardinality of V is relatively small so that $\mathcal{T}$ fits in internal memory, which is often the case for the low-dimensional data we consider, we can keep a (hash) table in memory, and $\mathcal{T}$ can be obtained in one complete scan through the relation.

When the cardinality of V is too large, multiple passes through the relation will be required to obtain $\mathcal{T}$, resulting in possibly excessive I/O cost. We can instead use an I/O-efficient external merge sort to compute $\mathcal{T}$ and minimize the I/O cost [91]. The merge sort process here is different from the traditional one: During the merging process, records with the same attribute value can be combined by summing the frequencies. After a small number of multi-way passes in the merge sort, the lengths of the runs will stop increasing; the length of each run is bounded by the cardinality of V, whose size, although too large to fit in memory, is typically small in comparison with the relation size.

If for any reason it's necessary to further reduce the I/O and CPU costs of the

precomputation of $\mathcal{T}$, a well-known approach is to use random sampling [62, 58]. The idea is to sample s tuples from the relation randomly and compute $\mathcal{T}$ for the sample. The sample data distribution is then used as an estimate of the real data distribution. To obtain the random sample in a single linear pass, the method of choice is the skip-based method [90] when the number of tuples T is known beforehand or the reservoir sampling variant [89] when T is unknown. A running up-to-date sample can be kept using a backing sample approach [28]. We do not consider in this thesis the issues dealing with sample size and the errors caused by sampling. Our experiments confirm that wavelet-based histograms that use random sampling as a preprocessing step give estimates that are almost as good as those from wavelet-based histograms that are built on the full data. On the other hand, as we shall see in Section 4.5, the wavelet-based histograms (whether they use random sampling in their preprocessing or not) perform significantly better at estimation than do naive techniques based on random sampling alone.

In this chapter, we do not consider the I/O complexities of the wavelet decomposition and of pruning, since both of them are performed on the cumulative data distribution $\mathcal{T}^{\mathcal{C}^+}$ and the size of $\mathcal{T}^{\mathcal{C}^+}$ is generally not large and can fit in internal memory for the low-dimensional data we considered in this chapter. For high-dimensional data, which is generally larger, the I/O efficiencies of the wavelet decomposition and of thresholding are considered in Chapter 6 and Chapter 7.

4.2.3 Wavelet Decomposition

The goal of the wavelet decomposition step is to represent the extended cumulative data distribution $\mathcal{T}^{\mathcal{C}^+}$ at hierarchical levels of detail.

First we need to choose wavelet basis functions. Haar wavelets are conceptually the simplest wavelet basis functions, and for purposes of exposition in this thesis,

we focus our discussion on Haar wavelets. They are fastest to compute and easiest to implement. We also implement a decomposition based upon linear wavelets that gives better estimation.

As illustrated in Chapter 3, the wavelet coefficients at each level of the recursion are normalized; the coefficients at the lower resolutions are weighted more heavily than the coefficients at the higher resolutions. One advantage of the normalized wavelet transform is that in many cases a large number of the detail coefficients turn out to be very small in magnitude. Truncating these small coefficients from the representation (i.e., replacing each one by 0) introduces only small errors in the reconstructed signal. We can approximate the original signal effectively by keeping only the most significant coefficients, determined by some pruning method, as discussed in the next subsection.

A better higher-order approximation for purposes of range query selectivity, for example, can be obtained by using linear wavelets as a basis rather than Haar wavelets. Linear wavelets share the important properties of Haar wavelets that we exploit for efficient processing. It is natural in conventional histograms to interpolate the values of items within a bucket in a uniform manner. Such an approximation corresponds to a linear function between the endpoints of the bucket. The approximation induced when we use linear wavelets is a piecewise linear function, which implies exactly this sort of linear interpolation. It therefore makes sense intuitively that the use of linear wavelets, in which we optimize directly for the best set of interpolating segments, will perform better than standard histogram techniques.

4.2.4 Pruning and Error Measures

Given the storage limitation for the histogram, we can only "keep" a certain number of the N wavelet coefficients. Let m denote the number of wavelet coefficients that

we have room to keep; the remaining wavelet coefficients will be implicitly set to 0. Typically we have $m \ll N$. The goal of pruning is to determine which are the "best" m coefficients to keep, so as to minimize the error of approximation.

We can measure the error of approximation in several ways. Let v_i be the actual size of a query q_i and let $\widehat{v}_i$ be the estimated size of the query. We use the following five different error measures:

	Notation	Definition
absolute error	e_i^{abs}	$\lvert v_i - \widehat{v}_i \rvert$
relative error	e_i^{rel}	$\dfrac{\lvert v_i - \widehat{v}_i \rvert}{\max\{1, v_i\}}$
modified relative error	$e_i^{\text{m\_rel}}$	$\dfrac{\lvert v_i - \widehat{v}_i \rvert}{\max\{1, \min\{v_i, \widehat{v}_i\}\}}$
combined error	e_i^{comb}	$\min\{\alpha \times e_i^{\text{abs}}, \beta \times e_i^{\text{rel}}\}$
modified combined error	e_i^{comb}	$\min\{\alpha \times e_i^{\text{abs}}, \beta \times e_i^{\text{m\_rel}}\}$

The parameters α and β are positive constants.

Our definition of relative error is slightly different from the traditional one, which is not defined when $v_i = 0$. The modified relative error treats over-approximation and under-approximation in a uniform way. For example, suppose the exact size of a query is $v_i = 10$. The estimated sizes $\widehat{v}_i = 5$ or $\widehat{v}_i = 20$ each have the same modified relative error, namely, $5/5 = 10/10 = 1$. In contrast, in terms of relative error, the estimation $\widehat{v}_i = 5$ has a relative error of $5/10 = 0.5$, and the estimation $\widehat{v}_i = 20$ has a relative error of $10/10 = 1$. The estimation $\widehat{v}_i = 0$ has a relative error of only $10/10 = 1$, while the modified relative error is $10/1 = 10$. The combined error reflects the importance of having either a good relative error or a good absolute error for each approximation. For example, for very small v_i it may be good enough if the absolute error is small even if the relative error is large, and for large v_i the absolute error may not be as meaningful as the relative error.

Once we choose which of the above measures to represent the errors of the individual queries, we need to choose a norm by which to measure the error of a collection of queries. Let $e = (e_1, e_2, \ldots, e_Q)$ be the vector of errors over a sequence of Q queries. We assume that one of the above five error measures is used for each of the individual query errors e_i. For example, for absolute error, we can write $e_i = e_i^{\text{abs}}$. We define the overall error for the Q queries by one of the following error measures:

	Notation	Definition
1-norm average error	$\|e\|_1$	$\frac{1}{Q} \sum_{1 \le i \le Q} e_i$
2-norm average error	$\|e\|_2$	$\sqrt{\frac{1}{Q} \sum_{1 \le i \le Q} {e_i}^2}$
infinity-norm error	$\|e\|_\infty$	$\max_{1 \le i \le Q} \{e_i\}$

These error measures are special cases of the p-norm average error, for $p > 0$:

$$\|e\|_p = \left(\frac{1}{Q} \sum_{1 \le i \le Q} {e_i}^p \right)^{1/p}.$$

The first step in pruning is weighting the coefficients in a certain way (which corresponds to using a particular basis, such as an orthonormal basis, for example). In particular, for the Haar basis, normalization is done by dividing the wavelet coefficients $\widehat{S}(2^j), \ldots, \widehat{S}(2^{j+1} - 1)$ by $\sqrt{2^j}$, for $0 \le j \le \log N - 1$. Given any particular weighting and error measure, we propose the following different pruning methods:

1. Choose the m largest (in absolute value) wavelet coefficients.
2. Choose m wavelet coefficients in a greedy way. For example, we might choose the m largest (in absolute value) wavelet coefficients and then repeatedly do the following two steps m times:
   (a) Choose the wavelet coefficient whose inclusion leads to the largest reduction in error.

(b) Throw away the wavelet coefficient whose deletion leads to the smallest increase in error.

Another approach is to do the above two steps repeatedly until a cycle is reached or improvement is small.

Several other variants of the greedy method are possible:

3. Start with the $m/2$ largest (in absolute value) wavelet coefficients and choose the next $m/2$ coefficients greedily.
4. Start with the $2m$ largest (in absolute value) wavelet coefficients and throw away m of them greedily.

The straightforward method of performing each iteration of the greedy method requires $O(N^2)$ time, and thus the total time is $O(mN^2)$. By maintaining a special dynamic programming tree structure, we can speed up the preprocessing significantly.

Theorem 4.1 *We can greedily choose m coefficients for any of the standard error measurements in $O(N(\log N)\log m)$ time and $O(N)$ space. If Method 4 is used, it can be done in $O(N\log^2 m)$ time and $O(N)$ space.*

Proof Sketch: For simplicity, we consider the case when Method 4 is used. The other proofs are similar.

We build an "error tree" of the wavelet transform. The leaves of the tree correspond to the original signal values, and the internal nodes correspond to the wavelet coefficients. Figure 4.1 is the error tree for $N = 8$; each node is labeled with the wavelet coefficient or signal value that it corresponds to. The wavelet coefficient associated with an internal node in the error tree contributes to the signal values at the leaves in its subtree. For each of the $2m$ nodes that correspond to the $2m$ largest wavelet coefficients, we store the error change introduced by deleting this coefficient.

At the ith ($1 \leq i \leq m$) step of the greedy thresholding, we throw away the

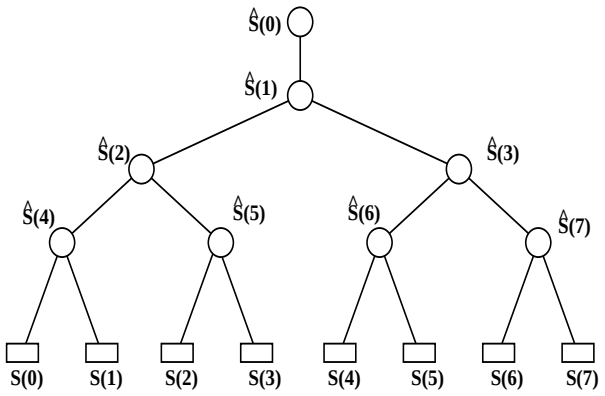


Figure 4.1: Error tree for $N = 8$

wavelet coefficient whose deletion causes the smallest increase in error. Suppose this coefficient corresponds to node n_i in the error tree. We need to update the error information of all the "relevant" nodes after the deletion. The relevant nodes fall into two classes: (a) the wavelet coefficients in the subtree rooted at n_i and (b) the wavelet coefficients on the path from n_i up to the root of the error tree.

Suppose the subtree rooted at n_i has k' leaves and m' class (a) wavelet coefficients. The maximum number of class (b) wavelet coefficients is at most $\log \frac{N}{k'}$. The important point is that the time to update a wavelet coefficient is proportional to the number of leaves in its subtree that change value. By a convexity argument, the worst-case locations for the m' class (a) wavelet coefficients are in the top $\log m'$ levels of n_i's subtree. The resulting time to update the m' class (a) wavelet coefficients is $O(k' \log m')$. The time to update the class (b) wavelet coefficients is $O(k' \log \frac{N}{k'})$.

By a convexity argument, over the m deletions, the worst case is for the m deleted wavelet coefficients to be in the top $\log m$ levels of the error tree. In this case, the m terms of $k' \log m'$ and the m terms of $k' \log \frac{N}{k'}$ sum to $O(N \log^2 m)$. □

It is well known if the wavelet basis functions are orthonormal then Method 1 is provably optimal for the 2-norm average error measure. However, for non-orthogonal wavelets like our linear wavelets and for norms other than the 2-norm, no efficient

technique is known for how to choose the m best wavelet coefficients, and various approximations have been studied [22].

Our experiments show that Method 2 does best overall in terms of accuracy for wavelet-based histograms. Method 1 is easier to compute but does not perform quite as well.

4.3 Online Reconstruction

In the online phase, a range query $a \leq X \leq b$ is presented. We reconstruct the approximate cumulative frequencies of $a-1$ and b, denoted by ${c'}_{a-1}$ and ${c'}_b$, using the m wavelet coefficients. The size of the query is estimated to be $c'_b - c'_{a-1}$. The time for reconstruction is crucial in the on-line phase:

Theorem 4.2 *For a given range query $a \leq X \leq b$, the cumulative frequencies of $a-1$ and b can be reconstructed from the m wavelet coefficients using an $O(m)$-space data structure in time $O(\log m + \#$ of relevant coefficients$) = O(\min\{m, \log N\})$.*

Proof Sketch: The method for reconstructing the frequency for a given domain element from the wavelet coefficients consists of identifying the $O(\log N)$ coefficients that are involved in the reconstruction. Each wavelet coefficient contributes to the reconstruction of the frequencies in a contiguous set of the domain. In a Haar wavelet, for example, each coefficient contributes a negative additive term to the reconstructed frequency for each domain value within a certain interval in the domain, and it contributes the opposite (positive) term for each value within an adjacent interval in the domain.

In particular, as demonstrated by the error tree of Figure 4.1 for the case $N = 8$, the wavelet coefficient $\widehat{S}(0)$ contributes as a positive additive term to each leaf value $S(0)$, $S(1)$, ..., $S(N-1)$ in its subtree. The wavelet coefficient $\widehat{S}(1)$ contributes as a positive term to the leaves $S(0)$, ..., $S(\frac{N}{2}-1)$ in its left subtree and as

a negative term to the leaves $S(\frac{N}{2}), \ldots, S(N-1)$ in its right subtree. The wavelet coefficient $\widehat{S}(2)$ contributes as a positive term to $S(0), \ldots, S(\frac{N}{4}-1)$ and as a negative term to $S(\frac{N}{4}), \ldots, S(\frac{N}{2}-1)$. The wavelet coefficient $\widehat{S}(3)$ contributes as a positive term to $S(\frac{N}{2}), \ldots, S(\frac{3N}{4}-1)$ and as a negative term to $S(\frac{3N}{4}), \ldots, S(N-1)$.

For higher-order wavelets (like linear wavelets), the contribution of a given wavelet coefficient can be represented by a constant number of adjacent intervals in the domain; unlike in the Haar case, the contribution of a given wavelet coefficient varies from point to point within each interval, but its contribution within an interval is specified by a polynomial function (which is linear in the case of linear wavelets).

The $O(m)$ intervals can be stored in linear space in an interval tree data structure [68]. Given a domain element, the wavelet coefficients corresponding to the element's frequency can be found in $O(\log m)$ time by a stabbing query on the intervals stored in the interval tree. The reconstructed frequency is then the sum of the contributions of each of those wavelet coefficients. □

Often it is useful to represent the histogram as an explicit piecewise smooth function rather than as m wavelet coefficients. For Haar wavelets the resulting function is a step function with at most $3m$ steps in the worst case, and for linear wavelets the function is piecewise linear with at most $5m$ changes in slope in the worst case. In real-life data, we can expect that the number of steps or segments is very close to m (in many cases exactly m). This property has been confirmed by an extensive set of experiments. Previous methods for expressing the histogram as a piecewise smooth function required $O(N)$ time, although some researchers suspected that $O(m \log N)$-time algorithms were possible [83]. We have developed an efficient and practical technique using priority queues that offers a substantial speedup:

Theorem 4.3 *The wavelet-based histogram can be transformed into a standard piecewise smooth representation in time* $O(m \times \min\{\log m, \log\log N\})$ *and using* $O(m)$

space.

Proof Sketch: By the reasoning behind the proof of Theorem 4.2, the reconstructed frequency is a polynomial function (which is constant for Haar wavelets and linear for linear wavelets) as long as the interval boundaries associated with the wavelet coefficients are not crossed. There are at most three interval boundaries per Haar wavelet coefficient and at most five interval boundaries per linear wavelet coefficient. We refer to the domain values where such boundaries occur for a given wavelet coefficient as the coefficient's *event points*.

We group the coefficients into the $\log N$ levels corresponding to the multiresolution wavelet decomposition. Let Δ be the maximum number of wavelet coefficients that can overlap another wavelet coefficient from the same level. We say that two coefficients overlap if there is a domain value whose frequency they both contribute to. For Haar wavelets we have $\Delta = 0$, and for any fixed order wavelet we have $\Delta = O(1)$. We construct the histogram by inserting into a priority queue the first event point for the first $\Delta + 1$ wavelet coefficients at each level. The polynomial function describing the reconstructed frequency does not change until the domain value corresponding to an event point is reached. We can find the next event point by performing a *delete_min* operation on the priority queue, at which point we insert into the priority queue the next event point for the coefficient involved in the *delete_min*. The polynomial function represented by the histogram is updated at each event point.

The desired time bound follows because each *delete_min* and *insert* operation takes time logarithmic in the size of the priority queue, which consists of at most $\min\{m, (\Delta + 1) \log N\})$ values at any time. □

4.4 Multi-Attribute Histograms

In this section, we extend the techniques in previous section to deal with the multi-dimensional case.

We extend the definitions in Section 4.1 to the multidimensional case in which there are multiple attributes. Suppose the number of dimensions is d and the attribute set is $\{X_1, X_2, \ldots, X_d\}$. Let $D_k = \{0, 1, \ldots, N_k - 1\}$ be the domain of attribute X_k. The value set V_k of attribute X_k is the set of n_k values of X_k that are present in relation R. Let $v_{k,1} < v_{k,2} < \cdots < v_{k,n_k}$ be the individual n_k values of V_k. The data distribution of X_k is the set of pairs $\mathcal{T}_k = \{(v_{k,1}, f_{k,1}), (v_{k,2}, f_{k,2}), \ldots, (v_{k,n_k} f_{k,n_k})\}$. The *joint frequency* $f(i_1, \ldots, i_d)$ of the value combination $(v_{1,i_1}, \ldots, v_{d,i_d})$ is the number of tuples in R that contain $v_{i_k,k}$ in attribute X_k, for all $1 \leq k \leq d$. The *joint data distribution* $\mathcal{T}_{1,\ldots,d}$ is the entire set of (*value combination, joint frequency*) pairs. The joint frequency matrix $\mathcal{F}_{1,\ldots,d}$ is a $n_1 \times \cdots \times n_d$ matrix whose $[i_1, \ldots, i_d]$ entry is $f(i_1, \ldots, i_d)$. We can define the *cumulative joint distribution* $\mathcal{T}^{\mathcal{C}}_{1,\ldots,d}$ and *extended cumulative joint distribution* $\mathcal{T}^{\mathcal{C}^+}_{1,\ldots,d}$ by analogy with the one-dimensional case. The extended cumulative joint frequency $\mathcal{F}^{\mathcal{C}^+}_{1,\ldots,d}$ for the d attributes X_1, X_2, ..., X_d is a $N_1 \times N_2 \times \cdots \times N_d$ matrix p defined by

$$p[x_1, x_2, \ldots, x_d] = \sum_{i_1=0}^{x_1} \sum_{i_2=0}^{x_2} \cdots \sum_{i_d=0}^{x_d} f(i_1, i_2, \ldots, i_d).$$

A very nice feature of our wavelet-based histograms is that they extend naturally to multiple attributes by means of multidimensional wavelet decomposition and reconstruction. The procedure of building the multidimensional wavelet-based histogram is similar to that of the one-dimensional case except that we approximate the *extended cumulative joint distribution* $\mathcal{T}^{\mathcal{C}^+}_{1,\ldots,d}$ instead of $\mathcal{T}^{\mathcal{C}^+}$.

In the preprocessing step, we obtain the joint frequency matrix $\mathcal{F}_{1,\ldots,d}$ and use it to compute the extended cumulative joint frequency matrix $\mathcal{F}^{\mathcal{C}^+}_{1,\ldots,d}$. We then use

the multidimensional wavelet transform to decompose $\mathcal{F}^{\mathcal{C}^+}_{1,\ldots,d}$. Finally, thresholding is performed to obtain the wavelet-based histogram.

In the query phase, in order to approximate the selectivity of a range query of the form $(a_1 \leq X_1 \leq b_1) \wedge \cdots \wedge (a_d \leq X_d \leq b_d)$, we use the wavelet coefficients to reconstruct the 2^d cumulated counts $p[x_1, x_2, \ldots, x_d]$, for $x_j \in \{a_j - 1, b_j\}$, $1 \leq j \leq d$. The following theorem adopted from [37] can be used to compute an estimate S' for the result size of the range query:

Theorem 4.4 ([37]) *For each $1 \leq j \leq d$, let*

$$s(j) = \begin{cases} 1 & \text{if } x_j = b_j; \\ -1 & \text{if } x_j = a_j - 1. \end{cases}$$

Then the approximate selectivity for the d-dimensional range query specified above is

$$S' = \sum_{\substack{x_j \in \{a_j - 1, b_j\} \\ 1 \leq j \leq d}} \prod_{i=1}^{d} s(i) \times p[x_1, x_2, \ldots, x_d].$$

By convention, we define $p[x_1, x_2, \ldots, x_d] = 0$ if $x_j = -1$ for any $1 \leq j \leq d$.

4.5 Experiments

In this section we report on some experiments that compare the performance of our wavelet-based technique with those of Poosala et al [67, 63, 66] and random sampling. Our synthetic data sets are those from previous studies on histogram formation and from the TPC-D benchmark [84]. For simplicity and ease of replication, we use method 1 for thresholding in all our wavelet experiments.

4.5.1 Experimental Comparison of One-Dimensional Methods

In this section, we compare the effectiveness of wavelet-based histograms with MaxDiff(V,A) histograms and random sampling. Poosala et al [67] characterized the types of histograms in previous studies and proposed new types of histograms. They concluded in their experiments that the MaxDiff(V,A) histograms perform best overall.

Random sampling can be used for selectivity estimation [32, 33, 50, 49]. The simplest way of using random sampling to estimate selectivity is, during the off-line phase, to take a random sample of a certain size (depending on the catalog size limitation) from the relation. When a query is presented in the on-line phase, the query is evaluated against the sample, and the selectivity is estimated in the obvious way: If the result size of the query using a sample of size t is s, the selectivity is estimated as sT/t, where T is the size of the relation.

Our one-dimensional experiments use the many synthetic data distributions described in detail in [67]. We use $T = 100{,}000$ to $500{,}000$ tuples, and the number n of distinct values of the attribute is between 200 and 500.

We use eight different query sets in our experiments:

A: $\{X \leq b \mid b \in D\}$.

B: $\{X \leq b \mid b \in V\}$.

C: $\{a \leq X \leq b \mid a, b \in D,\ a < b\}$.

D: $\{a \leq X \leq b \mid a, b \in V,\ a < b\}$.

E: $\{a \leq X \leq b \mid a \in D,\ b = a + \Delta\}$, where Δ is a positive integer constant.

F: $\{a \leq X \leq b \mid a \in V,\ b = a + \Delta\}$, where Δ is a positive integer constant.

G: $\{X = b \mid b \in D\}$.

H: $\{X = b \mid b \in V\}$.

Different methods need to store different types of information. For random sampling, we only need to store one number per sample value. The MaxDiff(V,A) histogram stores three numbers per bucket: the number of distinct attribute values in the bucket, the largest attribute value in the bucket, and the average frequency of the elements in the bucket. Our wavelet-based histograms store two numbers per coefficient: the index of the wavelet coefficient and the value of the coefficient.

In our experiments, all methods are allowed the same amount of storage. The default storage space we use in the experiments is 42 four-byte numbers (to be in line with Poosala et al's experiments [67], which we replicate); the limited storage space corresponds to the practice in database management systems to devote only a very small amount of auxiliary space to each relation for selectivity estimation [76]. The 42 numbers correspond to using 14 buckets for the MaxDiff(V,A) histogram, keeping $m = 21$ wavelet coefficients for wavelet-based histograms, and maintaining a random sample of size 42.

The relative effectiveness of the various methods is fairly constant over a wide variety of value set and frequency set distributions. We present the results from one experiment that illustrates the typical behavior of the methods. In this experiment, the spreads of the value set follow the *cusp_max* distribution with Zipf parameter $z = 1.0$, the frequency set follows a Zipf distribution with parameter $z = 0.5$, and frequencies are randomly assigned to the elements of the value set.[1] The value set size is $n = 500$, the domain size is $N = 4096$, and the relation size is $T = 10^5$. Tables 4.1–4.5 give the errors of the methods for query sets A, C, E, G, and H. Figure 4.2 shows how well the methods approximate the cumulative distribution of the underlying data.

[1]The *cusp_max* and *cusp_min* distributions are two-sided Zipf distributions. Zipf distributions are described in more detail in [63]. Zipf parameter $z = 0$ corresponds to a perfectly uniform distribution, and as z increases, the distribution becomes exponentially skewed, with a very large number of small values and a very small number of large values. The distribution for $z = 2$ is already very highly skewed.

Wavelet-based histograms using linear bases perform the best over almost all query sets, data distributions, and error measures. The random sampling method does the worst in most cases. Wavelet-based histograms using Haar bases produce larger errors than MaxDiff(V,A) histograms in some cases and smaller errors in other cases. The reason for Haar's lesser performance arises from the limitation of the step function approximation. For example, in the case that both frequency set and value set are uniformly distributed, the cumulative frequency is a linear function of the attribute value; the Haar wavelet histogram produces a sawtooth approximation, as shown in Figure 4.2b. The Haar estimation can be improved by linearly interpolating across each step of the step function so that the reconstructed frequency is piecewise linear, but doing that type of interpolation after the fact amounts to a histogram similar to the one produced by linear wavelets (see Figure 4.2a), but without the explicit error optimization done for linear wavelets when choosing the m coefficients.

We also studied the effect of storage space for different methods. Figure 4.3 plots the results of two sets of our experiments for queries from query set A. In the experiments, the value set size is $n = 500$, the domain size is $N = 4096$, and the relation size is $T = 10^5$. For data set 1, the value set follows *cusp_max* distribution with parameter $z = 1.0$, the frequency set follows a Zipf distribution with parameter $z = 1.0$, and frequencies are assigned to value set in a random way. For data set 2, the value set follows *zipf_dec* distribution with parameter $z = 1.0$, the frequency set follows uniform distribution.

In addition to the above experiments we also tried a modified MaxDiff(V,A) method so that only two numbers are kept for each bucket instead of three (in particular, not storing the number of distinct values in each bucket), thus allowing 21 buckets per histogram instead of only 14. The accuracy of the estimation was improved. The advantage of the added buckets was somewhat counteracted by less

Error Norm	*Linear Wavelets*	*Haar Wavelets*	*MaxDiff(V,A)*	*Random Sampling*
$\|e^{\text{rel}}\|_1$	0.6%	4.5%	8%	20%
$\|e^{\text{abs}}\|_1/T$	0.16%	0.8%	3%	8%
$\|e^{\text{abs}}\|_2/T$	0.26%	0.64%	3.2%	10%
$\|e^{\text{abs}}\|_\infty/T$	1.5%	5.6%	11%	13%
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 100$	0.6	4.4	8	20
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 1000$	5	30	80	200
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 1000$	5.1	70.4	12.8	19
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 1000$	19	224	192	243

Table 4.1: Errors of various methods for query set A.

Error Norm	*Linear Wavelets*	*Haar Wavelets*	*MaxDiff(V,A)*	*Random Sampling*
$\|e^{\text{abs}}\|_1/T$	0.2%	1.1%	5%	3.5%
$\|e^{\text{abs}}\|_2/T$	0.035%	0.18%	0.71%	0.6%
$\|e^{\text{abs}}\|_\infty/T$	2.4%	10%	20%	16%

Table 4.2: Errors of various methods for query set C.

Error Norm	*Linear Wavelets*	*Haar Wavelets*	*MaxDiff(V,A)*	*Random Sampling*
$\|e^{\text{abs}}\|_1/T$	0.1%	0.42%	0.15%	0.35%
$\|e^{\text{abs}}\|_2/T$	0.19%	0.96%	0.26%	0.64%
$\|e^{\text{abs}}\|_\infty/T$	1.5%	6%	3%	4.6%

Table 4.3: Errors of various methods for query set E with $\Delta = 10$.

Error Norm	*Linear Wavelets*	*Haar Wavelets*	*MaxDiff(V,A)*	*Random Sampling*
$\|e^{\text{abs}}\|_1/T$	0.03%	0.04%	0.04%	0.04%
$\|e^{\text{abs}}\|_2/T$	0.077%	0.32%	0.096%	0.24%
$\|e^{\text{abs}}\|_\infty/T$	1.6%	7%	2%	4.6%

Table 4.4: Errors of various methods for query set G.

accurate modeling within each bucket. The qualitative results, however, remain the same: The wavelet-based methods are significantly more accurate. Further improvements in the wavelet techniques are certainly possible by quantization and entropy encoding, but they are beyond the scope of this thesis.

Error Norm	*Linear Wavelets*	*Haar Wavelets*	*MaxDiff(V,A)*	*Random Sampling*
$\|e^{\text{abs}}\|_1/T$	0.03%	0.42%	0.2%	0.4%
$\|e^{\text{abs}}\|_2/T$	7.7%	16.1%	25.6%	38%
$\|e^{\text{abs}}\|_\infty/T$	0.2%	7%	2%	5 %

Table 4.5: Errors of various methods for query set H.

Data Range	*Linear Wavelets*	*Haar Wavelets*	*MHIST-2*
255	0.3%	1.5%	7%
511	0.3%	1.6%	8%
1023	0.3%	1.6%	6%
2047	0.3%	1.6%	6%

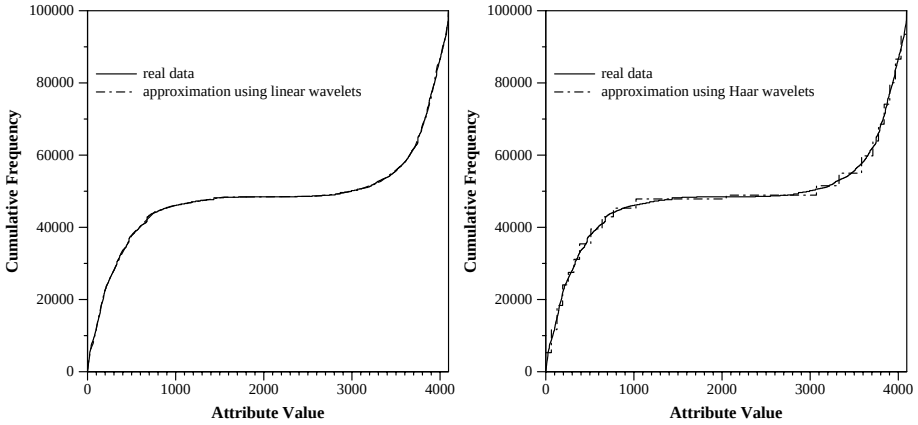
Table 4.6: $\|e^{\text{abs}}\|_1/T$ errors of various two-dimensional histograms for TPC-D data.

4.5.2 Experimental Comparison of Multidimensional Methods

In this section, we evaluate the performance of histograms on two-dimensional (two-attribute) data. We compare our wavelet-based histograms with the MaxDiff(V, A) histograms computed using the MHIST-2 algorithm [66] (which we refer to as MHIST-2 histograms).

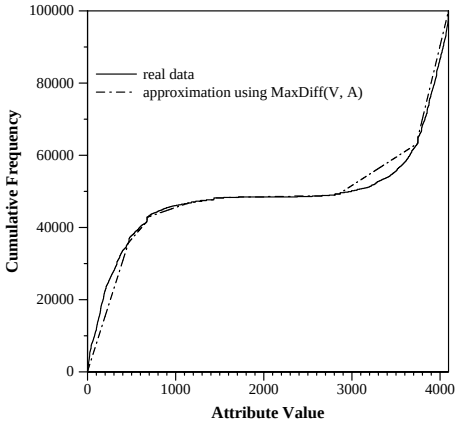
In our experiments we use the synthetic data described in [66], which is indicative of various real-life data [13], and the TPC-D benchmark data [84]. Our query sets are obtained by extending the query sets A–H defined in Section 4.5.1 to the multidimensional cases.

The main concern of the multidimensional methods is the effectiveness of the histograms in capturing data dependencies. In the synthetic data we used, the degree of the data dependency is controlled by the z value used in generating the Zipf distributed frequency set. A higher z value corresponds to fewer very high frequencies, implying stronger dependencies between the attributes. One question raised here is what is the reasonable range for that z value. As in [66], we fix the relation size T to be 10^6 in our experiments. If we assume our joint value set size is $n_1 \times n_2$, then

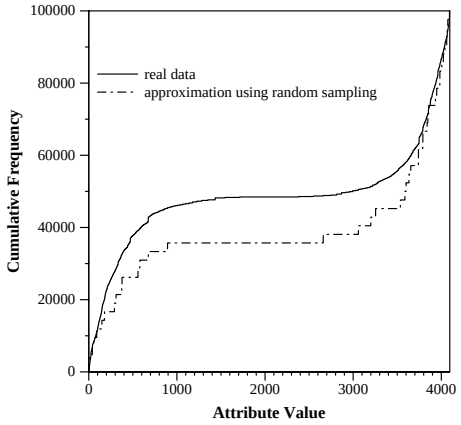


(a) Linear Wavelets

(b) Haar Wavelets



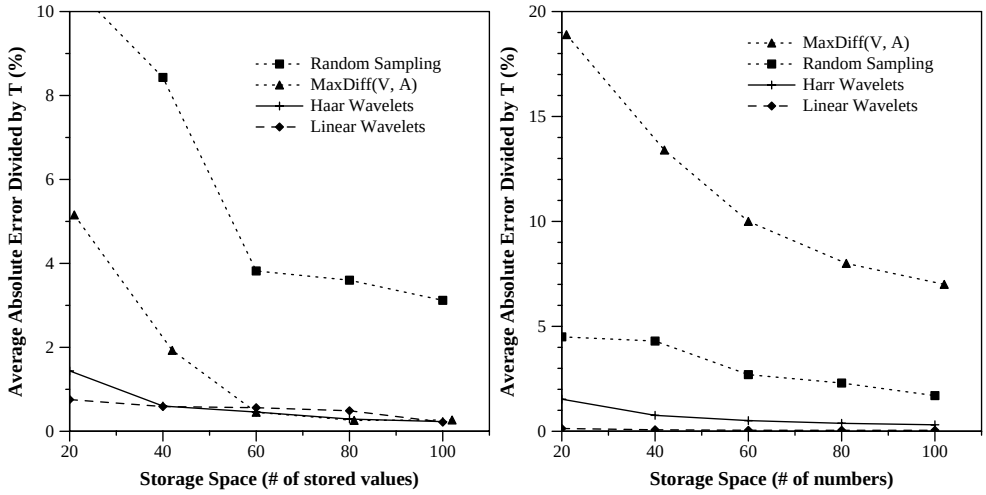
(c) MaxDiff(V, A)



(d) Random Sampling

Figure 4.2: Approximation of the cumulative data distribution using various methods.

in order to get frequencies that are at least 1, the z value cannot be greater than a certain value. For example, for $n_1 = n_2 = 50$, the upper bound on z is about 1.67. Any larger z value will yield frequency values smaller than 1. In our experiments, we choose various z in the range $0 \leq z \leq 1.5$. The value $z = 1.5$ already corresponds to a highly skewed frequency set; its top three frequencies are 388747, 137443, and 74814, and the 2500th frequency is 3. In [66], larger z values are considered; most of the Zipf frequencies are actually very close to 0, so they are instead boosted up to 1, with



(a) Data Set 1 (b) Data Set 2

Figure 4.3: Effect of storage space for various one-dimensional histograms using query set A.

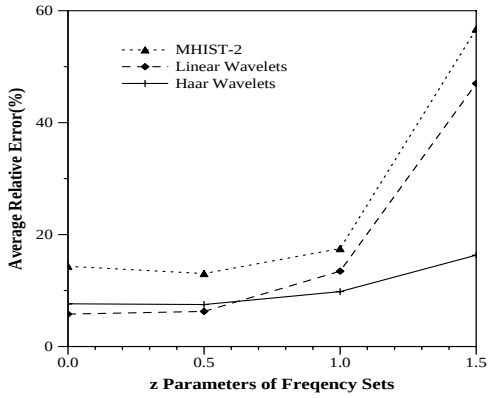
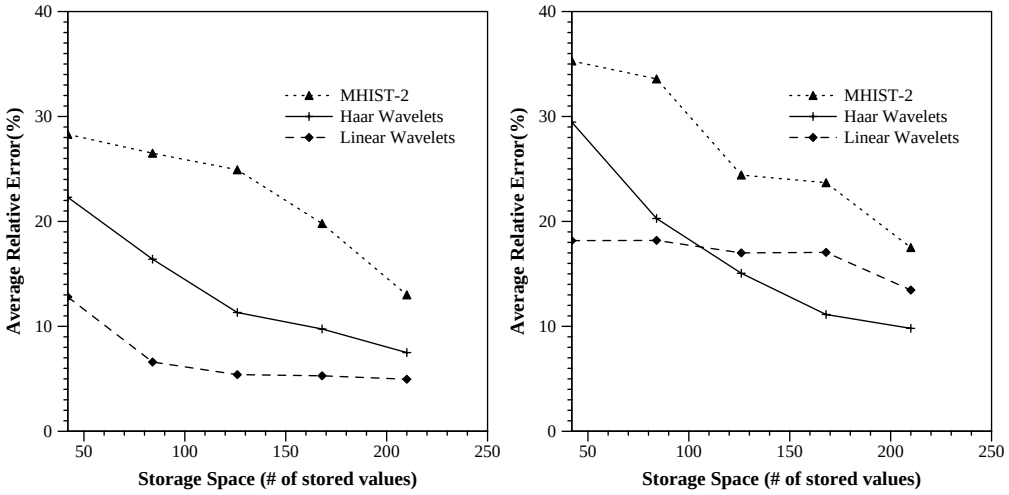


Figure 4.4: Effect of frequency skew, as reflected by the Zipf z parameter for the frequency set distribution.

the large frequencies correspondingly lowered, thus yielding semi-Zipf distributed frequency sets [64]. The relative effectiveness of different histograms is fairly constant over a wide variety of data distributions and query sets that we studied. Figure 4.4 depicts the effect of the Zipf skew parameter z on the accuracy of different types of histograms for one typical set of experiments. In these experiments, we use $N_1 = N_2 = 256$ and $n_1 = n_2 = 50$; the value set in each dimension follows *cusp_max*



(a) Zipf parameter z=0.5 (b) Zipf parameter z=1.0

Figure 4.5: Effect of storage space on two-dimensional histograms.

distribution with $z_s = 1.0$. The storage space is 210 four-bytes numbers (again, to be in line with the default storage space in [66]). It corresponds to using 30 buckets for MHIST-2 histogram (seven numbers per bucket) and keeping 70 wavelet coefficients for wavelet-based based histogram (three numbers per coefficient). The queries used are those from query set A.

In other experiments we study the effect of the amount of allocated storage space upon the accuracy of various histograms. As we mentioned above, the amount of storage devoted to a catalog is quite limited in any practical DBMS. Even without strict restrictions on the catalog size, a big catalog means that more buckets or coefficients need to be accessed in the on-line phase, which slows down performance. Figure 4.5 plots the effectiveness of the allocated storage space on the performance of various histograms. In the experiments, we use the same value set and query set as for Figure 4.4. The frequency skew is $z = 0.5$ for (a) and $z = 1.0$ for (b).

We conducted experiments using TPC-D data [84]. We report the results for a typical experiment here. In this experiment, we use column *L_SHIPDATA* and

column *L_RECEIPDATA* in the *LINEITEM* table, defined as follows:

$$L\_SHIPDATA = O\_ORDERDATA + random(121)$$

$$L\_RECEIPTDATA = L\_SHIPDATA + random(30),$$

where *O_ORDERDATA* is uniformly distributed between values *STARTDATA* and *ENDDATA* − 151, and $random(k)$ returns a random value between 1 and k. We fix the table size to be $T = 10^6$ and vary the size n of the value set V by means of changing *data range*, the difference between *ENDDATA* and *STARTDATA*. Table 4.6 shows the $\|e^{\text{abs}}\|_1/T$ errors of the different histogram methods for Set A queries.

4.6 Conclusions

In this chapter we have proposed a method to build efficient and effective histograms using wavelet decomposition. Our histograms give improved performance for selectivity estimation compared with random sampling and previous histogram-based approaches.

In Chapter 6, a new method based on a logarithm transform is proposed to further reduce the relative errors in wavelet-based approximation. Experiments show that by applying the new method, we can achieve much better accuracy for the data considered in Section 4.5.

High-dimensional data can often be much larger than the low-dimensional data considered in this chapter, and I/O communication can be a bottleneck. I/O efficient techniques for computing the wavelet transform and thresholding for high-dimensional data are discussed in Chapter 6–7 in the context of wavelet-based approximation techniques for OLAP applications.

Chapter 5

Wavelet-Based Approximation Techniques for OLAP Applications

Decision support systems (DSSs) are rapidly becoming a key to gaining competitive advantage for business. DSSs allow businesses to get data that is locked away in operational databases and turn the data into useful information. Many corporations have built or are building new unified decision-support databases called *data warehouses* on which users can carry out their analysis.

While operational databases maintain state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and grow over time. Analysts use the data warehouse to extract the business information that enable better decision making. This interactive decision-support process is called OLAP (On-line Analytical Processing) to distinguished it from conventional OLTP (On-line Transaction Processing) applications.

Computing multiple related group-bys and aggregates is one of the core OLAP operations. To faciliate computations, data warehouses often represent data in the form of several multidimensional databases (MDDB), also known as data cubes [30].

Consider a very simple exmaple, in which we have three dimensions *age, income*, and *education_level* and the "measure" *population*. The raw data are stored as a three-dimensional array S. For example, one "cell" of the array S may correspond to ($age = 30$, $income = \$45K$, $education\_level = 5$) with a *population* value of 5000, where education level 5 corresponds to high school graduate. Thus, an MDDB can be viewed as a d-dimensional array S, indexed by the values of the d dimensions (or *functional attributes*), whose cells contain the values of the *measure attribute* for the

corresponding combination of the functional attributes. In this thesis we shall call this kind of multidimensional array S the *raw data cube* to distinguish it from the *extended data cube* and the *partial-sum data cube*, which will be defined later.

Gray et al. [30] propose that the domain of each dimension be augmented with an additional value for each aggregation operation, denoted by *all*, to store aggregated values of the measure attribute among all the cells along that dimension, and this results in the *extended data cube* (or just *data cube*). In the above example, if we consider the aggregation operation *SUM*, any range-sum query of form (*age, income, education_level*), in which each attribute is either a singleton value or *all*, can be answered by accessing a single cell in the extended data cube. For example, the total population of 30-year-olds having an income of \$45K is a query specified by (30, \$45K, *all*), which can be answered by one cell access.

The extended data cube contains all the *subcubes* that correspond to elements of the power set of the set of dimensions, i.e., $\{age, income, education\_level\}$. To compute the data cube, we need to compute the total *population* grouped by all subsets of the three dimnesions. That is, we need to compute the total *population* grouped by *age*, by *income*, by *education_level*, by *age* and *income*, by *age* and *education_level*, by *income* and *education_level*, and the overall *population*.

An important class of queries on the data cube are the so-called *range-sum* queries, which are defined by applying the *SUM* operation over a selected contiguous range in the domains of some of the attributes [37]. For instance, in the data cube described above, an example of a range-sum query is to determine the total population for people with age from 25 to 45 and with income from \$50K to \$70K. Range-sum queries are important because several other classes of queries on the data cube, such as singleton queries and slice queries, are special cases.

5.1 Previous Work

The rapid acceptance of the importance of the cube operator has led to a variant of algorithms for fast computation of the data cube, and all previous work has concentrated on how to efficiently pre-compute the *exact* data cube and how to use the pre-computed cube to answer typical OLAP aggregation queries efficiently.

Harinarayan et al. [34] look into the problem of determining which subcubes to precompute when it is too expensive to precompute and store the entire extended data cube. They propose to use a lattice framework to express dependencies among subcubes and give a simple greedy algorithm to select subcubes based on the lattice framework. They also prove a strong performance guarantee for the greedy algorithm.

Agarwal et al. [2] present fast algorithms for computation of the entire data cube. They show how the structure of the data cube computation can be viewed in terms of a hierarchy of group-by operations. Their algorithms extend sort-based and hash-based grouping methods with several optimizations, like combining common operations across multiple group-bys, caching, and using precomputed groups-bys for computing other group-bys.

Zhao et al. [100] present an array-based algorithm to compute the data cube. It is shown that given appropriate compression techniques, the MOLAP (Multidimensional OLAP) algorithm is significant fast than a leading ROLAP (Relational OLAP) algorithm in [2].

Ross et al. [70] propose a novel divide-and-conquer algorithm for the fast compuation of data cubes over large sparse relations.

The most common method to reduce the response time of OLAP queries is to precompute some selected subcubes and then to build indices in these subcubes. In [31], Gupta et al. give algorithms that automate the selection of subcubes and indecies.

Ho et al. [37] present an efficient algorithm to speed up range-sum queries on a single data cube. The main idea is to preprocess the raw data cube S and precompute all the multidimensional partial sums, which can be represented in what we call the *partial-sum data cube* P. Any range-sum query can be answered by accessing and computing $2^{d'}$ entries from the partial-sum data cube, where d' is the number of dimensions for which ranges have been specified in the query. For example, in one dimension, the answer to a range-sum query

$$l_1 \leq D_1 \leq h_1 \tag{5.1}$$

(for which $d' = 1$) can be answered either as $\sum_{l_1 \leq i \leq h_1} S[i]$ in terms of the extended data cube or more efficiently as

$$P[h_1] - P[l_1 - 1]$$

in terms of the partial-sum data cube. In two dimensions, the range-sum query

$$l_1 \leq D_1 \leq h_1 \quad \text{AND} \quad l_2 \leq D_2 \leq h_2$$

(for which $d' = 2$) can be answered either as

$$\sum_{i_1=l_1}^{h_1} \sum_{i_2=l_2}^{h_2} S[i_1, i_2]$$

in terms of the extended data cube or more efficiently as

$$P[h_1, h_2] - P[l_1 - 1, h_2] - P[h_1, l_2 - 1] + P[l_1 - 1, l_2 - 1]$$

in terms of the partial-sum data cube. If query (5.1), for which $d' = 1$, is given in a two-dimensional setting, the answer can be computed either as $\sum_{l_1 \leq i_1 \leq h_1} S[i_1, all]$ in terms of the extended data cube or more efficiently as

$$P[h_1, |D_2| - 1] - P[l_1 - 1, |D_2| - 1]$$

in terms of the partial-sum data cube.

The problem with this partial-sum approach is that the partial sums are typically more dense in terms of storage representation than the original data. The resulting storage required can be proportional to the size of the raw data cube, which is very large. The appropriate $2^{d'}$ partial sum values for a given range-sum query might be stored in different disk blocks in the external memory and accessing them may require up to $2^{d'}$ disk I/Os, which can be very expensive in terms of I/O for high dimensions.

In reality, a data warehouse usually has more than one table (in a ROLAP system) or multidimensional array (in a MOLAP system), and each of these tables or arrays contains a very large number of tuples or entries. A user can query on any element in an data cube computed from any of those relations/arrays in an OLAP query. Computing all the data cubes and storing and retrieving them on disk becomes infeasible when the number of underlying relations/cubes is large since no enough disk storage is available. On the other hand, even with a huge amount of storage space, some popular aggregation queries (e.g., range-sum queries) may need to access a lot of precomputed cube cells and it will take a long time to execute.

The size of the pre-computed data cube and the complexity of the aggregation queries can cause queries to take very long to complete using any of the previous approaches. This delay is unacceptable in most DSS environments, as it severally limits productivities.

5.2 Our Approach

Our approach is different from previous work in that we target a dramatic reduction in storage and efficiently answering typical OLAP queries *approximately*.

There are a number of scenarios in which a user may prefer an approximate

answer in a few seconds over an exact answer that requires tens of minutes or more to compute. An example is a drill-down query sequence in data mining [36, 96]. Another consideration is that sometimes the base data are remote and unavailable, so that an exact answer is not an option until the data again become available [23].

We present I/O-efficient techniques based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the underlying multi-dimensional data. In the on-line phase, each aggregation query can generally be answered using the compact representation in one I/O or a small number of I/Os, depending upon the desired accuracy.

Our approach is preferable to the traditional approaches in two important respects. First of all, the traditional approaches require a huge amount of storage space for both the precomputation and the storage of the precomputed data cube. As we all know, the size of the precomputed data cube is much larger than that of the underlying raw data, especially when S is high-dimensional (e.g., more than six dimensions) [61]. In some applications there may be 100 dimensions! Even in moderately sized scenarios, there are usually many tables (in a ROLAP system) or multidimensional arrays (in a MOLAP system), and most of them are already very large in size by themselves. Since a query may be issued against any table (array), we have to compute and store one data cube for each of them. This fact would easily make the task infeasible even for moderately sized applications. Our approach in this chapter does not have the storage space problem. The size of each compact data cube is very small.

Secondly, even when a huge amount of storage space is available and all data cubes can be stored comfortably, it may take too long to answer a range-sum query, since all cells covered by the range need to be accessed. The problem persists even if the partial-sum technique of Ho et al. is used. However, our approach does not have

this problem at all. A query can be answered by retrieving the compact data cube, which in typical cases takes just one or a small number of I/Os.

In Chapter 6, we present our I/O-efficient techniques for approximately answering OLAP queries when the underlying multidimensional data are *dense*. In Chapter 7, we extend our techniques to deal with the more difficult case when the the underlying data are *sparse*.

Chapter 6

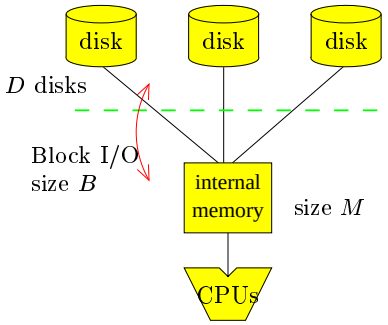
Approximation of the Dense Data Cube via Wavelets

In this chapter, we present an I/O-efficient technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the data cube for dense mutidimensional data.

In Section 6.1 we describe our model of I/O performance. In Section 6.2 we discuss the construction of the approximate partial-sum data cube and the analysis of I/O performance. The online phase, which requires only a constant number of I/Os in total to answer any range-sum queries, is explained and analyzed in Section 6.3. Our experimental results in Section 6.4 demonstrate a competitive advantage of our wavelet technique over histograms and random sampling.

6.1 I/O Model

We use the conventional *parallel disk model*, popularized in [93, 92]:



Parameters (in units of items):

N = problem data size;
M = size of internal memory;
B = size of disk block;
I = number of independent disks.

Data are transferred in large units of *blocks* of size B so as to amortize the latency of moving the read-write head and waiting for the disk to spin into position. For

brevity in this thesis, we restrict our attention to the case $I = 1$ of only one disk, but our results can be extended to the case $I > 1$ of parallel disks; the I/O results are improved by a factor of I.

6.2 The Compact Partial-Sum Data Cube Construction Algorithm

We adopt the notations in [37] to formulate the problem. Let $D = \{D_1, D_2, \ldots, D_d\}$ denote the set of dimensions, where each dimension corresponds to a functional attribute. We represent the underlying data by a d-dimensional array S of size $|D_1| \times |D_2| \times \cdots \times |D_d|$, where $|D_i|$ is the size of dimension D_i. We assume that each dimension D_i has an index domain $\{0, 1, \ldots, |D_i| - 1\}$. Each array element is a *cell*. A cell contains $S[i_1, i_2, \ldots, i_d]$, the value of the *measure attribute* for the corresponding combination $(i_1, i_2, \ldots, i_d)$ of the functional attributes. We let $N = \prod_{1 \le i \le d} |D_i|$ denote the total size (i.e., number of cells) of array S, and we define N_z to be the number of populated (nonzero) entries in S. We also refer to N_z as the *size of the raw data*. The *density* of array S is defined as

$$density(S) = \frac{N_z}{N}. \tag{6.1}$$

The sparse (ROLAP) representation of S is

$$\{(i_1, i_2, \ldots, i_d, S[i_1, i_2, \ldots, i_d]) \mid S[i_1, i_2, \ldots, i_d] \neq 0\} \tag{6.2}$$

and is used extensively in practice for sparse data.

Let us consider a general *range-sum* query:

$$Sum(l_1 \colon h_1, \ldots, l_d \colon h_d) = \sum_{i_1 = l_1}^{h_1} \cdots \sum_{i_d = l_d}^{h_d} S[i_1, \ldots, i_d].$$

An interesting subset of the general range-sum queries are d'-*dimensional range-sum queries* in which $d' \ll d$. Ranges are explicitly specified for only d' dimensions, and the ranges for the other $d-d'$ dimensions are implicitly set to be the entire domain $all_i = \{0, \ldots, |D_i| - 1\}$. The d' dimensions with explicit ranges can be any subset of the d dimensions and can vary from query to query. For simplicity in notation, for any given query, let us write the d' dimensions first so that the d' explicit ranges are on dimensions $D_1, D_2, \ldots, D_{d'}$ and the last $d - d'$ dimensions have implicitly defined ranges over the entire domain (i.e., $l_j = 0$ and $h_j = |D_j| - 1$, for $d' + 1 \leq j \leq d$). Using the above notation, these queries have the form $Sum(l_1{:}\, h_1, \ldots, l_{d'}{:}\, h_{d'}, all_{d'+1}, \ldots, all_d)$. For brevity, we simply write $Sum(l_1{:}\, h_1, \ldots, l_{d'}{:}\, h_{d'})$.

The popular *data cube* operator [30] can be viewed as computing the special case of all range-sums with singleton ranges, $Sum(l_1{:}\, h_1, \ldots, l_{d'}{:}\, h_{d'})$, in which $0 \leq l_i = h_i < |D_i|$, for $1 \leq i \leq d'$.

The partial-sum data cube P is a d-dimensional array of size $|D_1| \times |D_2| \times \cdots \times |D_d|$ (which is the same as the size of S). Its cells are defined as

$$\begin{aligned} P[x_1, \ldots, x_n] &= Sum(0 : x_1, \ldots, 0 : x_d) \\ &= \sum_{0 \leq i_1 \leq x_1} \cdots \sum_{0 \leq i_d \leq x_d} S[i_1, \ldots, i_d]. \end{aligned}$$

At a high level, our approximate partial-sum data cube construction algorithm works as follows:

1. In a preprocessing step, we form the partial-sum data cube P from the (raw) data cube S. (In our method, we further process P by replacing each cell value by its natural logarithm.)
2. We compute the wavelet decomposition of P, obtaining a set of N coefficients, where N is the size of array S.
3. We keep only the C most significant wavelet coefficients, for some C that cor-

responds to the desired storage usage and accuracy. The choice of which C coefficients to keep depends upon the particular thresholding method we use.

In the online phase, a query is answered by using the C wavelet coefficients to reconstruct approximations of the necessary values in P. Details of the online phase are given in Section 6.3.

Steps 1, 2, and 3, respectively, are discussed in the following three subsections. We show in particular that the total I/O complexity of Steps 1 and 2 is often linear in $O(N/B)$, which is optimal, and is never worse than $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, which is the number of I/Os required for sorting [3, 93, 91]. If we have $(M/B)^c \geq N/B$ for small c, which is typically the case, then the I/O time is $O(N/B)$. If not, it is still possible to get an $O(N/B)$-I/O algorithm by proper use of "chunking." Step 3 uses only $O(N/B)$ I/Os. The thresholding in Step 3 is interesting in that it takes advantage of the logarithm transform of Step 1 to achieve extremely good relative errors in its approximation of the partial-sum data cube P.

6.2.1 Computing the Partial-Sum Data Cube P

Computing the partial-sum data cube P from the (raw) data cube S in a naive manner can be expensive in terms of I/O when the size of S is very large and cannot fit in internal memory (which is likely in many OLAP applications). Thus, it is important to design I/O-efficient algorithms.

To compute the partial-sum data cube P, we need to do a series of partial-sum operations along each of the d dimensions of S. The standard programming language technique of storing a multidimensional array in a certain dimension order (such as row-major order or column-major order in the case of two-dimensional arrays) may not be efficient in terms of I/O when used with virtual memory. For example, in a row-major representation of a two-dimensional array, the partial sums along each row

can be computed efficiently, but accessing the array column-wise will cause almost one page fault per element accessed.

To achieve optimal I/O efficiency even when virtual memory is used, we can "chunk" the array, as suggested by [75, 73] and implemented in [88, 100] for other applications. Chunking is a way to divide a d-dimensional array into d-dimensional chunks in which each chunk is stored as one block on disk. Figure 6.1 shows a chunked three-dimensional array.

The size of a d-dimensional chunk is $c_1 \times \cdots \times c_d = B$ where c_i is the chunk size along dimension D_i and B is the size of a disk block. To implement the chunked array, we could use an approach similar to those of [88, 100], in which the disk layout is explicitly managed. For example, the high-level yet efficient TPIE system is used in [88] to do the chunking for matrix multiplication. In this chapter, for simplicity, we avoid the need for a separate disk management system like TPIE and use the virtual memory system to our advantage. Instead of forming the chunks and storing them as disk blocks, we form *logical chunks*. We store the multidimensional array as a one-dimensional array, with the order of the array cells specified as in Figure 6.1. All the cells in the same chunk are put into the same segment of the array. The solid lines are the boundaries between different chunks while the dashed lines separate different cells in the same chunk.

The I/O cost per chunk access is just one I/O, since the virtual memory system will handle the paging. The mapping between the indices of a cell in the logical multidimensional array and the index in the one-dimensional array is very easy to compute as long as the sizes of the dimensions and the size of a chunk along each dimension are known.

Theorem 6.1 *Consider multidimensional data cube S of size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i and $|D_1| \le |D_2| \le \cdots \le |D_d|$. The total amount of*

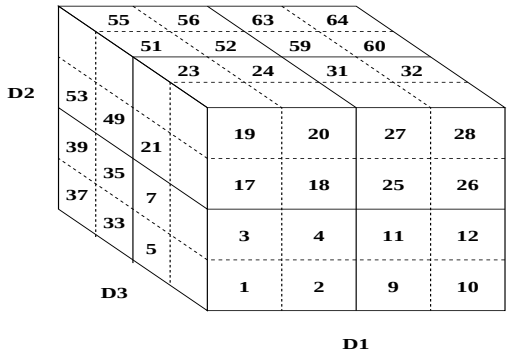


Figure 6.1: A chunked three-dimensional array with each chunk consisting of $2 \times 2 \times 2$ array cells. The number in each cell indicates its position in the one-dimensional array representation. In this example, cells numbered 1–8 form one chunk, cells 9–16 form the second chunk, and so on. Cells 1–32 form a "hyperplane of chunks" along dimensions D_1 and D_2; cells 33–64 form another hyperplane of chunks.

internal memory required to compute the the partial-sum data cube P in one linear scan of S is $O(\prod_{1 \leq i \leq d-1} |D_i|)$.

Proof Sketch: To minimize the memory needed, we use chunks with $c_d = 1$. For $k = 0, 1, \ldots, |D_d| - 1$, we do the following:

1. Read into internal memory the $(d-1)$-dimensional hyperplane for which the value of D_d is k.
2. Compute the partial sums along each of those $d - 1$ dimensions.
3. If $k \neq 0$, add the previous hyperplane values to those of the current hyperplane in a cell-by-cell manner along dimension D_d. Then write the resulting hyperplane back to disk. If $k = |D_d| - 1$, also write the current hyperplane back to disk.

□

We define the *dimension order* $(D_{i_1}, \ldots, D_{i_d})$ of the data cube chunks in d dimensions to be the ordering of the chunks that changes most rapidly along the rightmost dimension D_{i_d}, next most rapidly along dimension $D_{i_{d-1}}$, and so on. Different dimension orders correspond to different orders of accessing the data cube chunks. In the

example in Figure 6.1, the chunks are ordered with respect to one another according to the dimension order (D_3, D_2, D_1).[1]

Theorem 6.2 *In the case of general internal memory size M, we consider a data cube S in chunked form of size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i. The I/O complexity of computing the partial-sum data cube P is $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where B is the disk block size.*

Proof Sketch: We partition the d dimensions into g groups so that for the ith group $G_i = \{D_{i_1}, D_{i_2}, \ldots, D_{i_{k_i}}\}$, we have

$$\prod_{1 \le j \le k_i} |D_{i_j}| \prod_{\substack{1 \le l \le d \\ D_l \notin G_i}} c_l \le M. \tag{6.3}$$

If (6.3) cannot be met, that is, if the size of dimension D_{i_1} satisfies $|D_{i_1}| \prod_{l \ne i_1} c_l > M$, we put D_{i_1} into a singleton group containing only itself.

To compute the partial sums, at the ith pass of our algorithm, we read the chunks computed in the previous pass in dimension order $(*, \ldots, *, D_{i_1}, D_{i_2}, \ldots, D_{i_{k_i}})$, where $*$ denotes the dimensions that are not in G_i. If the ith group G_i satisfies (6.3), we successively read in a k_i-dimensional hyperplane of chunks along dimensions D_{i_1}, D_{i_2}, $\ldots$, $D_{i_{k_i}}$ (as defined in Figure 6.1). The size of each such k_i-dimensional hyperplane of chunks is given by the left-hand size of (6.3), and thus it fits in internal memory. For each hyperplane of chunks, after it is brought into internal memory, we compute the partial sums along each of those dimensions $D_{i_1}, \ldots, D_{i_{k_i}}$ and write the resulting hyperplane of chunks back to disk, chunk by chunk.

For a singleton group G_i, each hyperplane of chunks along dimension D_{i_1} (which because of the one-dimensionality is a "line" of chunks) may not fit in internal mem-

[1] Our definition of dimension order corresponds to the C programming language array declaration syntax and is different from that of [100].

ory. However, because of its single dimension, we can read each line of chunks along dimension D_{i_1} and compute its partial sums in one pass using $O(N/B)$ I/Os.

When we are done with the gth pass, we have the partial-sum data cube P. By algebraic manipulation we can show that $g = O(\log_{M/B} \frac{N}{B})$. The I/O cost of each of the g passes is $O(\frac{N}{B})$. The desired time bound follows. □

All choices of the chunk sizes c_i yield the upper bound result of Theorem 6.2. But some choices can do much better than others. Let us consider the example $B = M/2$, $N = \frac{1}{2}M^3$, $d = (\log M) + 1$, $|D_i| = 2$ (for $1 \leq i \leq d-2$), and $|D_{d-1}| = |D_d| = M$. The smart choice $c_i = 2$ (for $1 \leq i \leq d-2$) and $c_{d-1} = c_d = 1$ yields $g = 3$, and thus the algorithm runs in $O(N/B)$ I/Os, even though Theorem 6.1 does not apply. However, the alternate choice $c_i = 1$ (for $1 \leq i \leq d-1$) and $c_d = M/2 = B$ yields $g = d = (\log M) + 1$, and thus the algorithm runs in $O(\frac{N}{B} \log M)$ I/Os.

6.2.2 Wavelet Decomposition of a Partial-Sum Data Cube

The raw data cube S is often sparse in terms of the number of nonzero elements, but the partial-sum data cube P tends to be dense. For that reason we confine ourselves in this chapter to wavelet decompositions of dense data cubes, especially since the I/O processing is very efficient in terms of the dense representation.

The procedure of one-dimenisonal wavelet decomposition and reconstruction is illustrated in Section 3.5.

The one-dimensional wavelet decomposition and reconstruction procedure can be extended naturally to the multidimensional case. One way to do a multidimensional wavelet decomposition is by a series of one-dimensional decompositions. For example, in the two-dimensional case, we first apply the one-dimensional wavelet transform to each row of the data. Next, we treat these transformed rows as if they were themselves the original data, and we apply the one-dimensional transform to each column.

In terms of the I/O involved, the procedure for doing the multidimensional wavelet decomposition on the partial-sum cube P is similar to that of computing S from P, since they both perform certain kinds of operations along each dimension of a d-dimensional array:

Theorem 6.3 *Consider a multidimensional partial-sum data cube P having size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i and $|D_1| \le |D_2| \le \cdots \le |D_d|$. The total amount of internal memory required to compute the wavelet decomposition of P in one linear scan of P is $O(\prod_{1 \le i \le d-1} |D_i|)$.*

Theorem 6.4 *In the case of general internal memory size M, we consider a multidimensional partial-sum data cube P in chunked form of size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i. The I/O complexity of computing the multidimensional wavelet decomposition of P is $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where B is the disk block size.*

The proofs of the above theorems are similar to those of Theorem 6.1 and Theorem 6.2, respectively, except that we perform a one-dimensional wavelet decomposition instead of a one-dimensional partial-sum operation along each dimension.

6.2.3 Thresholding

Our motivation in this chapter is a compact, yet accurate representation of the partial-sum data cube. Given the storage limitation for the compact partial-sum data cube, we can only "keep" a certain number of the N wavelet coefficients. Let C denote the number of wavelet coefficients that we have room to keep; the remaining wavelet coefficients are implicitly set to 0. Typically we have $C \ll N$. The goal of thresholding is to determine which are the "best" C coefficients to keep, so as to minimize the error of approximation.

We can measure the error of approximation in several ways as descibed in Section 4.2.4.

The first step in thresholding is normalizing the coefficients in a certain way (which corresponds to using a particular basis, such as an orthonormal basis, for example). It is well-known that thresholding by choosing the C largest (in absolute value) wavelet coefficients after normalization is provably optimal in minimizing the L_2-norm of the absolute errors, among all possible choices of C nonzero coefficients, assuming that the wavelet basis functions are orthonormal [80]. With proper normalization, which we use, the Haar basis is orthonormal. As a result, normalization and thresholding perform well in practice on other norms of absolute error, such as the L_p-norms, for $p > 0$. There are no known computationally efficient methods for minimizing these other norms, although some approximation techniques have been studied [22].

In many circumstances, we want to minimize the *relative* error of our approximation. We use the following method to convert a method for achieving good absolute error into a method for achieving good relative error: We take the natural logarithm of each element in P before we do the wavelet decomposition, and we apply the inverse (i.e., exponentiation) after reconstruction in the online phase. The intuition for this preprocessing is based on the following fact: If we denote the function we want to approximate by $f(x)$, where x varies over some domain, let us consider its logarithm

$$g(x) = \ln f(x).$$

If we approximate the function g using wavelets with normalization and thresholding, then we can expect that the resulting approximation $\widehat{g}$ has small absolute error. In particular we have

$$\widehat{g}(x) = g(x) + \Delta(x),$$

where $|\Delta(x)|$ is typically small. The approximation $\widehat{f}(x)$ of $f(x)$ can be obtained by

$$\widehat{f}(x) = e^{\widehat{g}(x)} = f(x) \times e^{\Delta(x)}.$$

The relative error $|(f(x) - \widehat{f}(x))/f(x)|$ is thus given by

$$\left|\frac{f(x) - f(x)e^{\Delta(x)}}{f(x)}\right| = \left|1 - e^{\Delta(x)}\right| \approx |\Delta(x)| + O(\Delta(x)^2),$$

which is small when $|\Delta(x)|$ is small. The last approximation follows from the Taylor expansion of $e^{\Delta(x)}$ for small $|\Delta(x)|$.

In the wavelet decomposition step, in order to avoid values of $-\infty$ or large negative values, we further modify the partial-sum data cube by adding a small constant c to each cell before doing the multidimensional wavelet decomposition. In our experiments, we use $c = 1$. The value c is then subtracted in the online phase after the reconstruction is done.

This logarithm transformation has one remarkable property that we discuss further in Section 6.4: Not only does it dramatically lower the relative error of the approximation in our experiments, it also lowers the absolute error, no matter which norm we use to measure the error. Such a phenomenon does not occur when the logarithm transform is used with traditional histogram methods, such as MaxDiff histogram [66], for example; the MaxDiff relative error shows some improvement (compared with when the logarithm transform is not used), but its absolute error increases substantially.

Using the standard thresholding method, we need to pick the C largest (in absolute value) wavelet coefficients, which can be done in $O(N/B)$ I/Os by using a recursive distribution (or bucketing) method [3]. We can possibly combine the thresholding with the last pass of the wavelet decomposition procedure to further reduce the actual I/O cost.

The C wavelet coefficients together with their C indices (in the one-dimensional order of cells), form the compact partial-sum data cube. The table storage is thus $2C$ numbers in size. Further compression may be possible by quantization and entropy

encoding, but it is beyond the scope of this thesis. As a result, our experimental conclusions in Section 6.4 are conservative.

6.3 Answering Range Queries in the Online Phase

Each range-sum query can be expressed as sums and differences of a certain set of cell values from the multidimensional partial-sum data cube P [37]. The set of cells are the ones on the corners of the query hyperplane:

Theorem 6.5 ([37]) *The answer for the d-dimensional range-sum query*

$$l_1 \leq D_1 \leq h_1 \quad \text{AND} \quad \ldots \quad \text{AND} \quad l_d \leq D_d \leq h_d$$

is

$$v = \sum_{\substack{i_k \in \{l_k - 1, h_k\} \\ \text{for each } 1 \leq k \leq d}} \Big(\prod_{1 \leq k \leq d} s(i_k) \Big) \times P[i_1, i_2, \ldots, i_d], \tag{6.4}$$

where

$$s(k) = \begin{cases} 1 & \text{if } i_k = h_k; \\ -1 & \text{if } i_k = l_k - 1. \end{cases}$$

By convention, we define $P[i_1, i_2, \ldots, i_d] = 0$ if $i_j = -1$ for any $1 \leq j \leq d$.

We make use of Theorem 6.5 to compute our approximation $\widehat{v}$ of the query value v by computing an approximate reconstruction of each needed cell value $P[i_1, i_2, \ldots, i_d]$ in (6.4). Each reconstruction is based on the inverse wavelet transform of the C wavelet coefficients; the other $N - C$ coefficients are implicitly set to 0.

The time for reconstruction is crucial for the query performance.

Theorem 6.6 *In a d-dimensional partial-sum data cube with dimension sizes $|D_1|$, $\ldots$, $|D_d|$, the partial-sum value for a given cell can be reconstructed from the C wavelet coefficients using $O(dC)$ space in time $O(\min\{C, \sum_{1 \leq i \leq d} \log |D_i|\})$.*

The proof of this theorem is an extension of the proof for the one-dimensional case in Chapter 4.

If the C coefficients correspond to one disk block, only one I/O is needed to approximate any or all of the cells of P. The CPU time is $O(C)$ for each cell. By Theorem 6.5 each range-sum query may require the approximate reconstruction of 2^d cells. However, in typical range-sum queries, where only a few of the dimensions are specified, fewer cells need to be reconstructed and our technique is especially efficient:

Corollary 6.1 *If only d' of the d dimensions are involved in the query (and the remaining $d - d'$ dimensions are implicitly over their entire domains), we need to reconstruct only $2^{d'}$ cell values in Theorem 6.5 to answer the range query.*

Proof Sketch: For each of the $d - d'$ dimensions D_j that are completely spanned by the range-sum query, we have $l_j = 0$ and thus $P[i_1, i_2, \ldots, i_d] = 0$ if $i_j = l_j - 1$. Hence, there are only $2^{d'}$ nonzero values of $P[i_1, i_2, \ldots, i_d]$ in the summation in (6.4). □

6.4 Experimental Results

In this section we report on some experiments that compare the effectiveness and accuracy of our wavelet-based approximation technique with histogram-based techniques and random sampling.

6.4.1 Methods Used for Comparison

MaxDiff and Modified MaxDiff Histograms

Histograms approximate the data in one or more attributes of a relation by grouping attribute values into "buckets" and approximating the true attribute values and their frequencies based on summary statistics maintained in each bucket [8]. By replacing

the frequencies with the measure attribute values, we can use histograms to approximate a data cube. Since the data cube is a multidimensional array, we concentrate on multidimensional histograms in our discussion.

Muralikrishna and DeWitt [58] use an interesting spatial index partitioning technique for constructing equidepth histograms for multidimensional data. One drawback with this approach is that it considers each dimension only once during the partition. Poosala and Ioannidis [66] propose effective alternatives. Among the new classes of histograms they proposed, the multidimensional MaxDiff(V,A) histograms computed using the MHIST-2 algorithm are most accurate and perform better in practice than previous methods [66]. We compare our wavelet-based compact data cube with this class of histograms, which we shall simply refer to as MaxDiff histograms.

In our experiments, we form MaxDiff histograms to approximate the raw data cube. The range-sum queries are answered in the online phase using the approximate data cube represented by the histogram.

As we discussed in Section 6.2.3, combining a logarithm transformation with the approximation technique can be effective in reducing the relative error of the approximation. Our experiments also test a modified MaxDiff histogram that uses the logarithm transform.

The storage required for each of the b buckets in the histogram is three numbers: one to store the index of the front corner of the bucket (w.r.t. the linear order of the cells), another to store the index of the far corner of the bucket, and a third to store the (average) value associated with the bucket. (Poosala and Ioannidis [66] use $d + 1$ numbers to represent each bucket, but we instead use just three numbers per bucket by taking advantage of the one-dimensional order of the cells, as shown in Figure 6.1.)

Random Sampling

Several random sampling techniques, in which a large set of data is represented by a smaller random sample of the data, have been developed for database optimization [32, 33, 50, 49]. We can approximate the raw data cube by taking a random sample of a certain size from the nonzero cells of the raw data cube. When a range-sum query is presented in the online phase, the query is evaluated against the sample, and the approximate answer is extrapolated in the obvious way: If the answer to the query using a sample of size t is s, the approximate answer that will be reported is sT/t, where T is the total number of the nonzero cells in the raw data cube.

To store the samples as a compact data cube, we need to keep the indices of the sampled cells in the linear order, together with the value of the cell. Thus, storing a random sample of size t requires $2t$ numbers.

6.4.2 Data Description

In Chapter 4, a series of experimental results on selectivity estimation in low dimensions is presented that compares the accuracy of the wavelet-based approximation technique with that of traditional histograms and random sampling. The data used in Chapter 4 are TPC-D [84] benchmark data and some synthetic data sets generated using Zipf distribution. Those same data sets are used in our experiments for approximating low-dimensional data. We apply our new thresholding method on the wavelet-based histogram technique to show the significant improvement in accuracy.

In many real OLAP applications, the data have high dimension and the correlations among the functional attributes and the measure are intricate and do not match artificial data models. To make our experimental results meaningful, we performed our experiments using real-world data as well as synthetic data. For brevity in this chapter, we report our high-dimensional results on only the real-world data.

Our real-world data are obtained from the U.S. Census Bureau databases using their Data Extraction System (DES) [13]. Our data source is the Current Population Survey (CPS) and our extracted file is the March Questionnaire Supplement–Person Data File. This file contains 372 attributes from which we chose 11. Our measure attribute is *income* and the 10 functional attributes are *age, marital status, sex, education_attainment, race, origin, family_type, detailed_household_summary, age group,* and *class_of_worker*. In the original data file, all the attributes are already preprocessed and have a relatively small dimension size; that is, the domain of each dimension D_i is $\{0, 1, \ldots, |D_i| - 1\}$, for some small integer $|D_i|$. Although the dimension sizes are generally small, the high dimensionality results in a raw data cube with more than 16 million cells. The density of the raw data cube is about 0.001; there are 15,985 nonzero elements. In this setting we can imagine that several data cubes are approximated, and each data cube must be approximated using very little space.

6.4.3 Experimental Comparisons of Approximation

We compare the effectiveness of our compact partial-sum data cube via wavelets with those of MaxDiff histogram and random sampling. Different techniques need to store different types of information. We saw earlier in Sections 6.2.3 and 6.4.1 that our wavelet technique needs to store $2C$ numbers to represent C coefficients, MaxDiff needs $3b$ numbers to represent b buckets, and random sampling needs $2t$ numbers to store a sample of size t.

In our experiments, all methods are allowed the same amount of storage. We varied the allowed storage from 400 four-byte numbers to 2000 four-byte numbers. This small storage space corresponds to the practice in OLAP applications of handling several data cubes using a limited amount of storage space. The approximate data cubes can be accessed in the online phase in a single I/O.

For example, a space usage of 800 numbers corresponds to keeping $C = 400$ wavelet coefficients for our wavelet-based approximation, using $b = 267$ buckets for the MaxDiff histogram, and maintaining a random sample of size $t = 400$.

We measure the errors of the various approximation techniques in our experiments using five types of query predicates. In each dimension, the range is independently and uniformly generated according to the specified type:

A: $\{D_i \leq h_i \mid h_i \in \{0, 1, \ldots, |D_i| - 1\}\}$.

B: $\{l_i \leq D_i \leq h_i \mid l_i, h_i \in \{0, 1, \ldots, |D_i| - 1\}, l_i < h_i\}$.

C: $\{l_i \leq D_i \leq h_i \mid l_i, h_i \in \{0, 1, \ldots, |D_i| - 1\}, h_i = l_i + \Delta\}$, where Δ is a positive integer constant.

D: $\{D_i = b \mid b \in \{0, 1, \ldots, |D_i| - 1\}\}$.

E: $\{D_i = |D_i| - 1\}$.

We applied different combinations of these predicates on the set of 10 attributes. Our method is more accurate in approximating most types of online queries than the histogram and random sampling techniques, while it is comparable to the histogram method and better than random sampling for other types of queries. We present the results from one experiment of Type A queries. Table 6.1 gives the detailed comparisons using our wavelet technique, the MaxDiff histograms, and random sampling for a storage size of 800 four-byte numbers. Figure 6.2 plots the effect of different storage sizes. For random sampling, the errors are the averages over several different runs.

From Table 6.1 and Figure 6.2, we can see that our compact wavelet-based partial-sum data cube is more accurate in approximating online queries than the histogram and sampling techniques. As an example, let us consider a range-sum query on income whose exact answer is 10K (dollars). Our compact data cube with storage size 800 may give an answer of 12.2K (with relative error 22%). The average relative

Error Norm	*Wavelets*	*MaxDiff*	*Modified MaxDiff*	*Random Sampling*
$\|e^{\text{abs}}\|_1/S$	0.39%	0.84%	2.87%	1.6%
$\|e^{\text{abs}}\|_2/S$	1.37%	1.92%	84.5%	4.5%
$\|e^{\text{rel}}\|_1$	22%	6400%	580%	90%
$\|e^{\text{m\_rel}}\|_1$	27%	6400%	1180%	5000%
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 100$	12	690	98	70
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 100$	25	2361	155	100
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 100$	1.7	232	22	10
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 100$	3.1	904	54	10

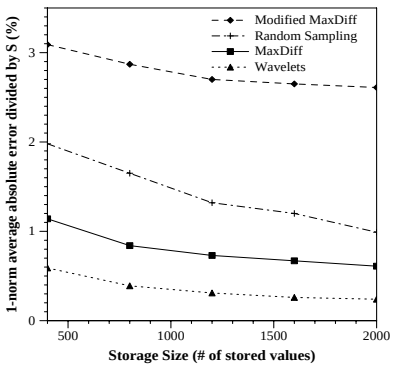
Table 6.1: Errors of various methods with storage size 800 on queries of Type A. The absolute errors are normalized by the largest value $S = 907,589$ in the multidimensional partial-sum data cube.

Error Norm	Data Set A		Data Set B		Data Set C		Data Set D	
	New Hist.	*Old Hist.*	*New Hist.*	*Old Hist.*	*New Hist.*	*Old Hist.*	*New Hist.*	*Old Hist.*
$\|e^{\text{abs}}\|_1/T$	1.2%	0.8%	2.0%	1.2%	1.0%	1.3%	1.0%	1.5%
$\|e^{\text{abs}}\|_2/T$	3.2%	1.1%	2.5%	1.6%	1.4%	1.7%	1.4%	1.8%
$\|e^{\text{rel}}\|_1$	6.9%	246%	4.6%	8.8%	3.0%	8.3%	5.5%	15%
$\|e^{\text{m\_rel}}\|_1$	7.7%	585%	4.9%	9.2%	3.1%	12835%	5.8%	24187%
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 100$	6.8	177	4.7	8.1	3.1	8.4	5.5	15.2
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 100$	10.5	313	6.5	64	4.1	20.3	10.4	18.6
$\|e^{\text{comb}}\|_1$, $\alpha = 1, \beta = 10$	0.7	24	0.5	0.9	0.3	0.8	0.8	1.5
$\|e^{\text{comb}}\|_2$, $\alpha = 1, \beta = 10$	1.1	47	0.7	8.8	0.4	2.0	1.0	1.8

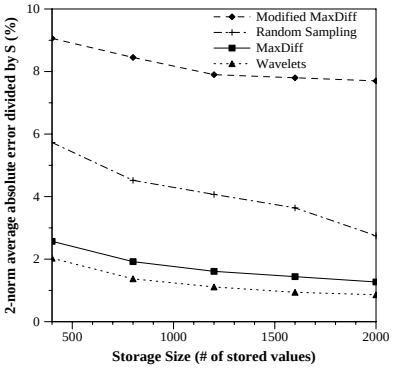
Table 6.2: Errors of the new wavelet-based histogram and that of the old wavelet-based histogram for query Type A using various low-dimensional data sets.

error can typically be reduced to about 13% if we increase the storage space from 800 numbers to 2000 numbers.

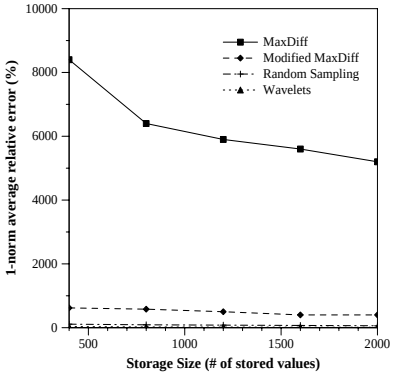
In Chapter 4, the wavelet-based histogram method was proposed to approximate low-dimensional data for selectivity estimation. The wavelet-based histograms performed best overall in comparison with MaxDiff histograms and random sampling. In this chapter we demonstrate the effectiveness of our new method against that of the previous wavelet-based histograms in Chapter 4. We call our new histograms *new wavelet-based histograms* to distinguish them from the *old wavelet-based histograms*



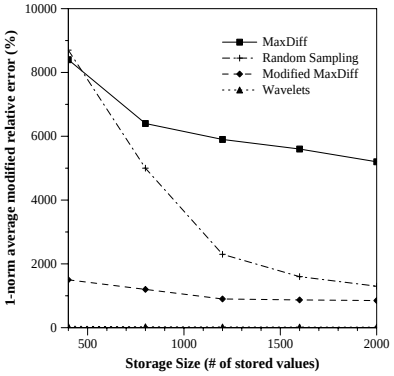
(a) 1-norm average absolute error



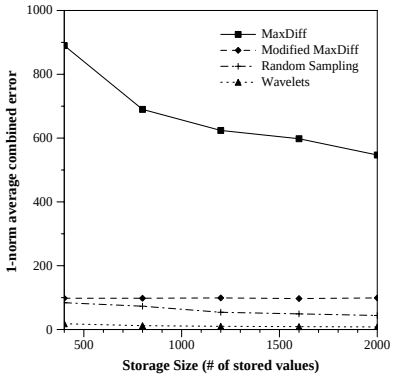
(b) 2-norm average absolute error



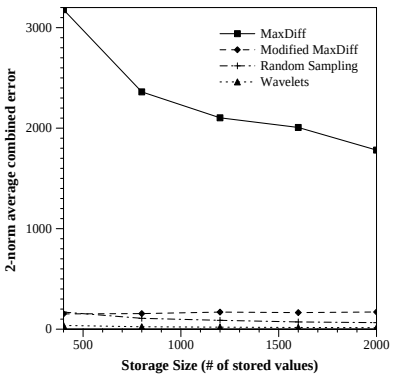
(c) 1-norm average relative error



(d) 1-norm average modified relative error



(e) 1-norm average combined error, $\alpha = 1.0$, $\beta = 100$



(f) 2-norm average combined error, $\alpha = 1.0$, $\beta = 100$

Figure 6.2: Effect of storage space for various methods on range-sum queries of Type A.

in Chapter 4. The relative effectiveness of the two methods is constant over a wide variety of low-dimensional data sets that we used. We present results for Type A queries using the synthetic data described in Chapter 4. Table 6.2 shows various types of errors for four data sets. In the experiments, all data sets are generated using Zipf distribution with different Zipf parameters. Data sets A and B are one-dimensional data. Their value set size is $n = 500$, the domain size is $N = 4096$, and the relation size is $T = 10^5$. We keep 21 wavelets coefficients in the experiments. Data sets C and D are two-dimensional data. Their value set size is $n = 2500$, the domain size is $N = 65536$, and the relation size is $T = 10^6$. We keep 70 wavelets coefficients in the experiments. The new method dramatically reduces the relative error and combined errors of the approximation, while keeping other error measurements roughly the same.

6.5 Conclusions

In this chapter, we have presented an I/O-efficient technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the partial-sum data cube. We build our compact partial-sum data cube on the logarithms of the partial sums of the raw data values of a multidimensional array. We get excellent approximations for online range-sum queries with limited space usage and computational cost. Our new thresholding method of taking the logarithmic transform also provides significant improvement for wavelet-based histograms used in selectivity estimation of low-dimensional data.

In constructing the compact partial-sum data cube, our algorithm needs to first compute the dense partial-sum data cube and then perform wavelet decomposition on it. Our algorithm works well when the raw data cube is dense, since the size of the partial-sum data cube is similar to that of the raw data cube for the dense case. But

if the data are sparse and the amount of (nonzero) entries is very large, it may not be feasible to compute the partial-sum data cube first and do the wavelet decomposition on the partial-sum data cube since the size of the partial-sum data cube will be much large than that of the underlying raw data. We address how to efficiently construct the compact representation for massive sparse data using wavelets in Chapter 7.

Chapter 7

Approximate the Sparse Data Cube Using Wavelets

In this chapter, we extend the techniques in Chapter 6 and present a novel method that provides approximate answers to high-dimensional OLAP aggregation queries in *massive sparse* sets.

This chapter is organized as follows: In Section 7.1 we describe the problem. In Section 7.2 we give a high level description of our method which consists of three components. Section 7.3– 7.4 describe the components in details. We present our experimental results in Section 7.5 and make concluding remarks in Section 7.6.

We use the I/O model defined in Section 6.1 and the notations defined in Section 6.2 through this chapter.

7.1 Introduction

Computing multiple related group-bys and aggregates is one of the core operations in On-Line Analytical Processing (OLAP) applications. A particular characteristic of the data sets—and the primary concern of this chapter—is that they are *massive* and *sparse*.

In this chapter, we extend our wavelet-based techniques presented in Chapter 6 to approximately answer OLAP range-sum queries for *sparse* data cube. Particularly, we resolve the following four important issues:

1. *I/O-efficiency* of the compact data cube construction, especially when the underlying multidimensional array is very sparse. Our earlier wavelet approach

in Chapter 6 requires a dense storage representation during the construction of the compact data cube, which is infeasible for very large sparse data sets. Histogram techniques [66, 67] usually require excessive I/O costs when the data size is large and the dimensionality is high. Our new wavelet approach is fast and space-efficient even for massive sparse data.

2. *Response time* in answering an online query. Generally one or a small number of I/Os suffice, and the CPU time is small.

3. *Accuracy* in answering typical OLAP queries. The performance of the algorithm is generally superior to that of random sampling.

4. *Progressive refinement* of the approximate answers in case more accuracy is desired.

7.2 The Method: A High-Level Outline

In this section we summarize our basic method. We elaborate on the details in the following sections. Our method has three main components:

1. *Decomposition.* We compute the wavelet decomposition of the multidimensional array S, obtaining a set of C' wavelet coefficients, where C' is roughly equal to the number N_z of nonzero coefficients. We assume as in practice that the array is very sparse, that is, $N_z \ll N$. (The dense case is covered previously in Chapter 6.) We use sparse techniques to do the wavelet decomposition directly based upon the sparse (ROLAP) representation of S. In Section 7.3.1, we give the details of the our efficient wavelet decomposition algorithms and analyze their I/O performance.

2. *Thresholding and Ranking.* We keep only C wavelet coefficients, for some $C \ll C'$ that corresponds to the desired storage usage and accuracy. The choice of

which C coefficients to keep depends upon the particular thresholding method we use. We order (rank) the C wavelet coefficients according to their importance in the context of accurately answering typical aggregation queries. The C ordered coefficients compose our compact data cube. The issue of how to choose proper thresholding method and how to define the (relative) importance of a wavelet coefficient is the key for the accuracy of our approximation method and will be addressed in Section 7.3.2.

3. *Reconstruction.* In the online phase, an aggregation query is processed by using the k most significant wavelet coefficients, for some $k \leq C$, to reconstruct an approximate answer. The choice of k depends upon the time the user is willing to spend. More accurate answers can be provided upon request by using more coefficients to refine the previous approximations. The efficiency of the reconstruction step, in terms of both I/O performance and CPU time, is crucial, since it affects the query response time directly. We give our efficient query answering algorithm in Section 7.4.

7.3 Constructing the Compact Data Cube

7.3.1 Building the Compact Data Cube

The goal of this step is to compute the wavelet decomposition of the multidimensional array S, obtaining a set of C' wavelet coefficients. In this section, we present two algorithms to deal with the difficult and important case in which the underlying data are very sparse.

Our algorithms takes the sparse representation (6.2) of array S as input, which we assume is in *dimension order* $\langle D_1, \ldots, D_d \rangle$; that is, the indices of the entries change most rapidly along the rightmost dimension D_d, next most rapidly along

dimension D_{d-1}, and so on. The array entries for which the values in the initial set of dimensions $D_1, \ldots, D_k$ are fixed form a $(d-k)$-dimensional hyperplane, which we denote by $\langle D_{k+1}, \ldots, D_d \rangle$. Without loss of generality, we assume that D_d is the dimension with the smallest domain size, which improves performance in practice.

Algorithm I: Decomposition with Separate Transposition Step

We use a concrete example to illustrate the compact data cube construction process. Suppose S is a three-dimensional array for which $|D_2| \times |D_3| \leq M - 2B$, but $|D_1| \times |D_2| \times |D_3| > M - 2B$. We do the wavelet decomposition in two passes. We partition the three dimensions into two groups: $\{D_1\}$ and $\{D_2, D_3\}$. All tuples with a fixed dimension D_1 value are contiguous in the input S and form a $\langle D_2, D_3 \rangle$ hyperplane. The first pass is done by reading in all $\langle D_2, D_3 \rangle$ hyperplanes, one by one. Each hyperplane is guaranteed to fit in internal memory. An ordinary two-dimensional wavelet decomposition is performed on each hyperplane and the result, still using the sparse representation, is written out using an output double buffer. After all $\langle D_2, D_3 \rangle$ hyperplanes have been processed, we obtain an intermediate array S', which is the result of applying the wavelet decomposition to S along dimensions D_2 and D_3. The elements of array S' are still stored in the dimension order $\langle D_1, D_2, D_3 \rangle$. We reorganize them so that they are stored according to the dimension order $\langle D_2, D_3, D_1 \rangle$. We then do the wavelet decomposition of S' along D_1. We scan S' and read in the $\langle D_1 \rangle$ hyperplanes, one by one, and an ordinary multidimensional wavelet decomposition is performed on each of them (in this particular example, a one-dimensional decomposition). The output of this pass constitutes the final result of the algorithm.

In general, we partition the d dimensions into g groups, for some $1 \leq g \leq d$. Let the jth group be $G_j = \{D_{i_{j-1}+1}, D_{i_{j-1}+2}, \ldots, D_{i_j}\}$, where $i_0 = 0$ and $i_g = d$. The

requirement that G_j must satisfy is that either

$$\left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right| \leq M - 2B \tag{7.1}$$

or else G_j is a singleton group (i.e., $i_{j-1} + 1 = i_j$). We can form the groups one by one in a greedy manner: Given groups $G_1, \ldots, G_{j-1}$, we choose i_j to be the largest integer in the range $(i_{j-1}, d]$ such that (7.1) still holds, or else $i_j = i_{j-1} + 1$.

Algorithm I for constructing the compact data cube consists of g passes. The groups are processed in reverse order, one per pass. In the $(g - j + 1)$st pass, each hyperplane in the jth group G_j is read in, one by one, and processed (i.e., the ordinary multidimensional wavelet decomposition is performed), and the results are written out to be used for the next pass.

One problem is that the density of the intermediate results will increase from pass to pass, since performing wavelet decomposition on sparse data usually results in more nonzero coefficients. The number of nonzero coefficients can increase by a factor of $\log |D_i|$ when doing the wavelet decomposition along dimension D_i. We may thus have to process more and more entries from pass to pass, even though a lot of entries are very small in magnitude. The natural solution to this problem is truncation. In each pass, after obtaining the intermediate array S', we truncate S' by cutting off entries with small magnitude and keep roughly only N_z entries. We then use the truncated S' as the input for next pass. This process keeps the sparsity of all the intermediate results unchanged. The truncation operation is reasonable because it is in line with the wavelet decomposition method itself; that is, the most significant wavelet coefficients contribute the most to the reconstruction of the original signal.

However, it is too expensive to do truncation after all intermediate entries are written out in a multipass process. Instead, we do truncation in each pass via an online learning process. We keep online statistics of the distribution of the values of the intermediate wavelet coefficients during each pass and dynamically maintain a

cutoff value. Any entry with its absolute value below the cutoff value will be thrown away on the fly. The cutoff value is adjusted periodically when more coefficients are generated and the statistics change. For example, if too many entries have been cut off, the cutoff value will be decreased. On the other hand, if too few entries have been thrown away, we need to increase the cutoff value. This self-adjusting procedure works well in practice.

After the $(g-j+1)$st pass, for each $1 < j \leq g$, in which G_j is processed, a transposition is performed on the output of the pass in order to regroup the cells according to the dimension order required by the next pass. (There is no need to do a transposition after the last pass, namely, when $j = 1$.) The data can be transposed in $\log_{M/B}(|D_{i_{j-1}+1}| \times |D_{i_{j-1}+2}| \times \cdots \times |D_{i_j}|)$ distribution passes, based upon the values of the indices in G_j, and thus the number of I/Os is

$$O\left(\frac{N_z}{B}\log_{M/B}\left(\left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right|\right)\right). \tag{7.2}$$

The first distribution pass of each transposition can be done during the wavelet decomposition procedure (at the cost of reserving half the internal memory for buffer space), which speeds up performance in practice.

After all g passes are done, we obtain the final wavelet decomposition, which consists of $C' \approx N_z$ coefficients. (In the next section we describe the final thresholding process to reduce the number of coefficients from C' to C.) We denote the value of a coefficient by v. Each coefficient, with its index, is of the form

$$c = (i_1, \ldots, i_d, v). \tag{7.3}$$

The method described above for how the groups are formed is rather conservative and is related to how dense arrays are processed in Chapter 6. The following result follows from (7.2) with little algebra:

Theorem 7.1 *For internal memory of size M and block size B, we consider an array S of size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i, having a total of N_z nonzero entries. The I/O complexity of Algorithm I (using the truncation procedure) is*

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N}{B}\right).$$

In practice we can do better than the conservative bound in Theorem 7.1 by using Algorithm I with a more liberal group partitioning and a smaller number of groups. For example, we might want relax condition (7.1) and partition the dimensions so that the jth group satisfies

$$density(S) \times \left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right| \le \frac{M - 2B}{2}, \tag{7.4}$$

for $1 \le j \le g$. The value of $density(S) = N_z/N$ is typically a very small fraction in practice. The new partition results in a much smaller value of g than the one in Theorem 7.1. Often we get $g = 2$, and thus only two passes (and one intermediate transposition) are needed. However, it may no longer be desirable to do the transposition via the distribution approach of (7.2); instead we can do the transposition by sorting, which uses $O(\frac{N_z}{B} \log_{M/B} \frac{N_z}{B})$ I/Os. (See [92] for a proof in the I/O model that transposition is equivalent to sorting.) If all the processed hyperplanes individually fit into internal memory, the resulting I/O bound for Algorithm I will be

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N_z}{B}\right), \tag{7.5}$$

which is optimal.

The tradeoff for the liberal partitioning strategy is that from time to time, certain hyperplanes may not fit in internal memory and their wavelet decomposition may require multiple passes. But the number of such hyperplanes requiring extra time

is usually small, and the recomputation is localized to the hyperplanes. Overall we can get great savings in Algorithm I by using a smaller g value, as our experiments indicate in Section 7.5. We use (7.4) as a guideline for determining the groups, and we find that $g = 2$ as long as

$$M \geq 2\sqrt{density(S) \times N_z} + 2B.$$

Algorithm II: Decomposition without Separate Transposition Step

One problem with Algorithm I when using a conservative group partitioning is that we need to perform a transposition operation to reorder the array entries between passes, for example, after processing one group and before proceeding to the next.

In this section, we present another decomposition algorithm, called Algorithm II, that uses buffering and knowledge of the domain sizes to avoid an explicit transposition step between passes. The tradeoff is that Algorithm II must use a conservative group partition, and thus may require more passes g than in Algorithm I. The input to the algorithm is the sparse representation of array S. We assume the input is in dimension order $\langle D_1, \ldots, D_d \rangle$. As before, we partition the d dimensions into g groups, for some $1 \leq g \leq d$, in a greedy fashion. Let the jth group be $G_j = \{D_{i_{j-1}+1}, D_{i_{j-1}+2}, \ldots, D_{i_j}\}$, such that either

$$\left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right| \leq \frac{M}{2B+1}, \tag{7.6}$$

or if $j = 1$ then

$$|D_1| \times |D_2| \times \cdots \times \left|D_{i_j}\right| \leq M - 2B, \tag{7.7}$$

or else G_j is a singleton group (i.e., $i_{j-1} + 1 = i_j$).

The algorithm consists of g passes. We process the g groups in reverse order, one per pass. Let us illustrate the process with the following concrete example. Suppose S

is a six-dimensional array and we partition the six dimensions according to the above procedure. Let the partition be $\{D_1, D_2\}$, $\{D_3, D_4\}$, and $\{D_5, D_6\}$.

At the beginning of the first pass, we reserve two types of buffers in internal memory: a *processing buffer* and *output buffers*. We have one processing buffer whose size is $|D_5| \times |D_6|$. We have $|D_5| \times |D_6|$ output double buffers, each of size $2B$. Each output buffer has a unique $b\_id \in \{0, 1, \ldots, |D_5| \times |D_6| - 1\}$.

We then read in all $\langle D_5, D_6 \rangle$ hyperplanes, one by one, into the processing buffer, and perform the ordinary multidimensional wavelet decomposition. The results of the decomposition are then subjected to the cutoff value. For those coefficients whose magnitudes are bigger than the cutoff value, we do not write them to disk. Instead, for a coefficient $c = (i_1, \ldots, i_6, v)$, we write it into the output buffer with $b\_id = i_5 \times |D_6| + i_6$. When half of an output double buffer becomes full, we write its data to disk.

After we are done with all the $\langle D_5, D_6 \rangle$ hyperplanes, we are finished with the first pass.

In the second pass, we read into the processing buffer the blocks created during the previous pass, in the order of increasing b_id value (and for each b_id, in the order the blocks were created). The important observation here is that the resulting order is the dimension order $\langle D_5, D_6, D_1, D_2, D_3, D_4 \rangle$, which is needed for doing the $\langle D_3, D_4 \rangle$ hyperplane decompositions, and thus we avoid the need for a separate transposition step. The transposition is done for free as a result of the buffering mechanism. We can then process similarly as in the previous pass, except that now the number of output buffers becomes $|D_3| \times |D_4|$ and the size of processing buffer becomes $|D_3| \times |D_4|$.

When performing the decomposition for the dimensions in the last pass (when processing G_1), we no longer need the output buffers, and we can write the decomposition results out directly.

In the previous example, all the individual dimension sizes satisfied the condition

$$|D_i| \leq \frac{M}{2B+1}, \tag{7.8}$$

except possibly for D_1. We call D_i a *big dimension* if its size does not satisfy (7.8). All big dimensions form singleton groups. So far we have not described how to process big dimensions. If there is only one big dimension, namely, D_1, then we can perform the wavelet decomposition along D_1 in a linear pass since there is no need for transposing the data via the output buffers.

However, when there are multiple big dimensions, Algorithm II as described above no longer works (except for D_1). Let us suppose that D_i is a big dimension, for some $i \neq 1$. The simplest approach is to use the technique of Algorithm I, in which the wavelet decomposition is computed in $O(N_z/B)$ I/Os, followed by a distribution-based transpose operation, which takes $O(\frac{N_z}{B} \log_{M/B} |D_i|)$ I/Os. The first level of the distribution can be incorporated into the pass that does the wavelet decomposition, yielding an improvement in practice.

Putting everything together, we find that the total wavelet decomposition of array S requires $O(\log_{M/B} \frac{N}{B})$ passes over the data, each pass using $O(N_z/B)$ I/Os, and we get the following result:

Theorem 7.2 *For internal memory of size M and block size B, we consider an array S of size $N = \prod_{1 \leq i \leq d} |D_i|$, where $|D_i|$ is the size of dimension D_i, having a total of N_z nonzero entries. The number of I/Os needed for the wavelet decomposition of S using Algorithm II is*

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N}{B}\right).$$

The I/O bounds in Theorems 7.1 and 7.2 are a tremendous improvement over the bound in Theorem 6.4, which is larger by a multiplicative factor of $1/\mathit{density}(S) =$

$O(N/N_z)$. In practice, the approach of Algorithm I is generally preferable, since it can accommodate a more liberal group partitioning strategy, which often results in a much smaller g value, typically $g = 2$, and the optimal I/O bound of (7.5).

7.3.2 Thresholding and Ranking

Given the storage limitation for the compact data cube, we can only "keep" a certain number of the C' wavelet coefficients. Let C denote the number of wavelet coefficients that we have room to keep; the remaining wavelet coefficients will be implicitly set to 0. Typically we have $C \ll C'$, so that the C coefficients can fit into one or a few disk blocks. The goal of thresholding is to determine which are the "best" C coefficients to keep, so as to minimize the error of approximation.

It is well-known that thresholding by choosing the C largest (in absolute value) wavelet coefficients after normalization is provably optimal in minimizing the 2-norm of the absolute errors for the set of singleton queries:

$$\{Sum(i_1{:}\, i_1, \ldots, i_d{:}\, i_d) \mid 0 \le i_j < |D_j|, \text{ for each } 1 \le j \le d\}.$$

That is, if we want to minimize the average absolute error in approximating all the individual cells in S, the best choice is to keep the C largest (in absolute value) wavelet coefficients [80].

But our goal here is to approximate d'-dimensional range-sum queries, where usually $d' \ll d$. If a coefficient c_i is more likely to contribute to a query than another coefficient c_j, we would like to give c_i a higher weight, even its absolute value is smaller than that of c_j. From Lemma 7.4 in Section 7.4, we can observe the following fact: For a coefficient $c = (i_1, \ldots, i_d, v)$, the bigger the value $\sum_{j=1}^{k}[i_j = 0]$, the more likely c is going to contribute to a d'-dimensional range-sum query.[1] We therefore

[1] We use the notation $[i_j = 0]$ to denote 1 if $i_j = 0$ and 0 if $i_j \neq 0$.

define the weight function w for coefficient c as

$$w(c) = \sum_{j=1}^{k} [i_j = 0].$$

In doing thresholding, we pick the C'' ($C < C'' < C'$) largest wavelet coefficients in absolute value, and among those we pick the C wavelet coefficients with the largest weight with respect to function w. (We break ties using the absolute value.) We rank the C coefficients by ordering them according to their weights in decreasing order to get our compact data cube. Let us denote by R the compact data cube computed for a d-dimensional array S. We can view R as a one-dimensional array of length C, with each entry being a wavelet coefficient value and its indices in the sparse representation of form

$$R[j] = (i_{j_1}, \ldots, i_{j_d}, v_j), \qquad 1 \leq j \leq C. \tag{7.9}$$

The entries are ranked according to their importance in decreasing order.

7.4 Answering Online Queries

In this section, we show how to answer online aggregation queries using the compact data cube constructed in the previous section. Let's consider a range-sum query $Sum(l_1 : h_1, \ldots, l_{d'} : h_{d'})$.

An advantage of the partial-sum approach of Chapter 6 is that we need to reconstruct only $2^{d'}$ values, not 2^d values, of the partial-sum data cube in order to answer this query, which requires processing $\min\{k, 2\prod_{1 \leq i \leq d'} \log |D_i|\}$ wavelet coefficients in the worst case, where k is the specified number of stored coefficients in the compact data cube to use for the approximation. If we abandon the partial-sum data cube and use the compact data cube, one big concern is that we may lose this speed advantage. It turns out that we will not. In fact, our algorithm for answering queries

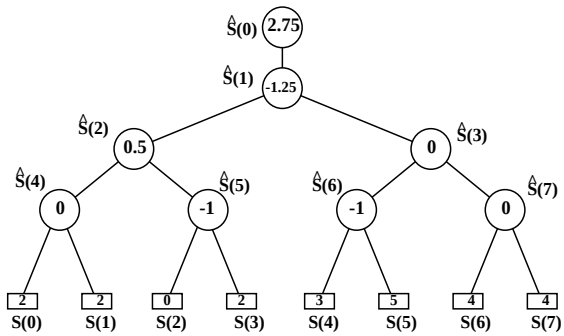


Figure 7.1: Error tree for Example 3.21

is even faster, both theoretically and in practice: In the partial-sum scenario using the logarithm transform in Chapter 6, a wavelet coefficient may be involved in the approximation of several of the $2^{d'}$ values, so the CPU time complexity is $O(2^{d'}d'k)$, whereas in our new approach the CPU time complexity for the standard implementation is only $O(d'k)$ with a very small constant factor of proportionality (roughly 2) in terms of the number of arithmetic operations. (A more complicated approach yields an $O(d'k)$-time algorithm for the former case, but only if the logarithm transform is not used.)

To fully understand the online query processing algorithm, we examine the relationship between wavelet coefficients and a range-sum query by using the *error tree* structure introduced in Section 4.2.4. The error tree is built based upon the wavelet transform procedure. Figure 7.1 is the error tree for Example 3.21 in Section 3.5.

In an error tree, each internal node is associated with a wavelet coefficient value, and each leaf is associated with an original signal value. (For purposes of exposition, the wavelet coefficients are unnormalized, but in the implementation the values are normalized and the algorithm is modified appropriately.) Internal nodes and leaves are labeled separately. Their labels are in the domain $\{0, 1, \ldots, N-1\}$ for a signal of length N. For example, the root is an internal node with label 0 and its node value is 2.75 in Figure 7.1. For convenience, we shall use "node" and "node value" interchangeably.

The construction of the error tree exactly mirrors the wavelet transform procedure. It is a bottom-up process. First, leaves are assigned original signal values from left to right. Then wavelet coefficients are computed, level by level, and assigned to internal nodes.

As the figure shows, the (unnormalized) value of each internal node i is denoted by $\widehat{S}(i)$, and the value of each leaf j is denoted by $S(j)$. We use $left(i)$ and $right(i)$ to denote the left and right child of any node i, and we use $leaves(i)$ to denote the set of leaves in the subtree rooted at i. The average value of the nodes in $leaves(i)$ is denoted by $ave\_leaf\_val(i)$. For any leaf i, we use $path(i)$ to denote the set of internal nodes (or the node values) along the path from i to the root. For any two leaves $l \leq h$, we use $S(l\!:\!h)$ to denote the range-sum between $S(l)$ and $S(h)$, that is,

$$S(l\!:\!h) = \sum_{i=l}^{h} S(i). \tag{7.10}$$

Below are some useful facts that are helpful for understanding our algorithm.

Lemma 7.1 *For any nonroot internal node i, we have*

$$\widehat{S}(i) = \frac{ave\_leaf\_val(left(i)) - ave\_leaf\_val(right(i))}{2}.$$

Lemma 7.2 *The reconstruction of any signal value depends only upon the values of those internal nodes along the path from the corresponding leaf to the root. That is, the reconstruction of any leaf value $S(i)$ depends only upon the nodes in $path(i)$.*

Consider the range sum (7.10). It is an algebraic sum of many internal nodes. For example, for $l = 0$, $h = 1$,

$$S(0) + S(1) = (\widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) + \widehat{S}(4)) + (\widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) - \widehat{S}(4))$$

Note that the two terms of $\widehat{S}(4)$ cancel out each other, so $\widehat{S}(4)$ does not contribute to the final summation. In general, any original signal $S(i)$ can be represented as the algebraic sum of the wavelet coefficients along the path $path(i)$. A nonroot internal node contributes positively to the leaves in its left subtree and negatively to the leaves in its right subtree. For a range sum, the contributors may cancel each other, and we have the following result:

Lemma 7.3 *A nonroot internal node x contributes to the range sum (7.10) only if $x \in path(l) \cup path(h)$. In particular, the contribution of x to (7.10) is*

$$(|left\_leaves(x, l\!:\!h)| - |right\_leaves(x, l\!:\!h)|) \times \widehat{S}(x),$$

where

$$left\_leaves(x, l\!:\!h) = leaves(left(x)) \cap [l, h]; \tag{7.11}$$

$$right\_leaves(x, l\!:\!h) = leaves(right(x)) \cap [l, h]. \tag{7.12}$$

Mathematically, we can write any range sum in terms of all the wavelet coefficients as

$$S(l\!:\!h) = \sum_x (|left\_leaves(x, l\!:\!h)| - |right\_leaves(x, l\!:\!h)|)\widehat{S}(x),$$

where the summation is over all internal nodes x. In our algorithm, however, we do not evaluate all the terms. We quickly determine the nonzero contributors and evaluate their contribution.

Let us relook at Example 3.21 in Section 4.2.3; its error tree is shown in Figure 7.1. The original signal S can be reconstructed from $\widehat{S}$ by the following formulas:

$$S(0) = \widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) \qquad + \widehat{S}(4)$$

$$
\begin{aligned}
S(1) &= \widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) - \widehat{S}(4) \\
S(2) &= \widehat{S}(0) + \widehat{S}(1) - \widehat{S}(2) + \widehat{S}(5) \\
S(3) &= \widehat{S}(0) + \widehat{S}(1) - \widehat{S}(2) - \widehat{S}(5) \\
S(4) &= \widehat{S}(0) - \widehat{S}(1) + \widehat{S}(3) + \widehat{S}(6) \\
S(5) &= \widehat{S}(0) - \widehat{S}(1) + \widehat{S}(3) - \widehat{S}(6) \\
S(6) &= \widehat{S}(0) - \widehat{S}(1) - \widehat{S}(3) + \widehat{S}(7) \\
S(7) &= \widehat{S}(0) - \widehat{S}(1) - \widehat{S}(3) - \widehat{S}(7)
\end{aligned}
$$

For example, $S(2)$ depends only upon $path(2) = \{\widehat{S}(5), \widehat{S}(2), \widehat{S}(1), \widehat{S}(0)\}$. If we want to compute the range sum $S(2{:}\,5)$, we can see that although $\widehat{S}(1)$ contributes to each of $S(2)$, $S(3)$, $S(4)$, and $S(5)$, its total contribution cancels out, and the net effect is that it does not contribute at all. Similarly, $\widehat{S}(5)$ and $\widehat{S}(6)$ are gone, and we have

$$S(2{:}\,5) = 4\widehat{S}(0) - 2\widehat{S}(2) + 2\widehat{S}(3).$$

The formula can also be verified by using Lemma 7.3.

We can extend the above observation to the multidimensional case. For example, for a two-dimensional array with $|D_1| = |D_2| = 8$, we can answer the range-sum query $Sum(4{:}\,7, 0{:}\,7)$ (note that D_2's range is the special range all_2) using the following formula that involves only coefficients $\widehat{S}(0,0)$ and $\widehat{S}(1,0)$:

$$Sum(4{:}\,7, 0{:}\,7) = 8 \times 4 \times (\widehat{S}(0,0) - \widehat{S}(1,0)).$$

Lemma 7.4 *In the reconstruction process, a wavelet coefficient $c = (i_1, \ldots, i_d, v)$ contributes to the range sum $Sum(l_1{:}\,h_1, \ldots, l_{d'}{:}\,h_{d'})$ only if $i_{d'+1} = \cdots = i_d = 0$. Its contribution is*

$$
\begin{aligned}
& v \prod_{j=1}^{d} (|left\_leaves(i_j, l_j{:}\,h_j)| - |right\_leaves(i_j, l_j{:}\,h_j)|) \\
&= v \prod_{j=1}^{d'} (|left\_leaves(i_j, l_j{:}\,h_j)| - |right\_leaves(i_j, l_j{:}\,h_j)|) \times \prod_{j=d'+1}^{d} |D_j|.
\end{aligned}
$$

To answer a query of form $Sum(l_1{:}\, h_1, \ldots, l_{d'}{:}\, h_{d'})$ using k coefficients of the compact data cube R, we use the following function:

Function $AnswerQuery(R, k, l_1, h_1, \ldots, l_{d'}, h_{d'})$
//answer a range-sum query using k coefficients of the compact data cube R
  $answer = 0$;
  for $j = 1, 2, \ldots, k$ **do**
    if $Contribute(R[j], l_1, h_1, \ldots, l_{d'}, h_{d'})$
      then $answer = answer + Compute\_Contribution(R[j], l_1, h_1, \ldots, l_{d'}, h_{d'})$;
  for $i = d' + 1, \ldots, d$ **do**
    $answer = answer \times |D_i|$;
  return $answer$;

Function $Contribute(R[j], l_1, h_1, \ldots, l_{d'}, h_{d'})$ returns *true* if the entry $R[j]$ contributes to the range-sum query, and returns *false* otherwise. Function *Compute_Contribution* computes the actual contribution of $R[i]$ to the specified query.

Based upon the regular structure of the error tree and the preceding lemmas, we devise two algorithms to compute the functions *Contribute* and *Compute_Contribution*. The CPU time complexity of the two algorithms are $O(d)$ (for function *Contribute*) and $O(d')$ (for function *Compute_Contribution*), respectively.

We denote an entry of R by $r = (i_1, i_2, \ldots, i_d, v)$ in the following functions.

Function $Contribute(r, l_1, h_1, \ldots, l_{d'}, h_{d'})$
//check the coefficient r to see if it contributes
//to the given range-sum query
  $contribute$ = **true**;
  //check the coefficient according to Lemma 7.4 first
  $j = d' + 1$;
  while ($contribute$) and ($j \leq d$)
    if ($i_j \neq 0$)
      then $contribute$ = **false**;
    else
      $j = j + 1$;
  $j = 1$;

```
  while (contribute) and (j ≤ d')
    //get the the label of the left most leaf for the subtree
    //rooted at node i_j in an error tree with |D_j| leaves
    L_l = left_most_leaf(i_j, |D_j|);
    //get the the label of the right most leaf for the subtree
    //rooted at node i_j in an error tree of N = |D_j|
    L_r = right_most_leaf(i_j, |D_j|);
    if ([l_j, h_j] ∩ [L_l, L_r] = ∅) or  //the subtree has no overlap with the query range
      ([l_j, h_j] ⊆ [L_l, L_r]) or       //the subtree contains the query range
      (([L_l, L_r] ⊆ [l_j, h_j]) and ((L_l − l_j) = (h_j − L_r)))    //a special case
      then contribute = false;
    if (l_j = h_j)                       //consider the special case of equal query
      if ([l_j, h_j] ⊆ [L_l, L_r])       //query point is in the subtree
        then contribute = true
      else                               //query point is not in the subtree
        contribute = false;
  return contribute;

Function Compute_Contribution(r, l_1, h_1, ..., l_d', h_d')
//compute the contribution of r to the give range-sum query
  contribution = v;
  for j = 1, 2, ..., d' do
    if (i_j = 0)                         //the special case for root node
      then contribution = contribution × N;
    else
      depth = ⌊log i_j⌋;                 //compute the depth of node i_j in the error tree
      if (depth < log |D_{i_j}| − 1)     //node i_j is not a parent of leaves
        then
          //get the the label of the left most leaf for node i_j's left
          //subtree in an error tree with |D_j| leaves
          L_ll = left_most_leaf(left_child(i_j), |D_j|);
          //get the the label of the right most leaf for node i_j's left
          //subtree in an error tree with |D_j| leaves
          L_lr = right_most_leaf(left_child(i_j), |D_j|);
```

//get the the label of the left most leaf for node i_j's right
//subtree in an error tree with $|D_j|$ leaves
$L_{rl} = \mathit{left\_most\_leaf}(\mathit{right\_child}(i_j), |D_j|)$;
//get the the label of the right most leaf for node i_j's right
//subtree in an error tree with $|D_j|$ leaves
$L_{rr} = \mathit{right\_most\_leaf}(\mathit{right\_child}(i_j), |D_j|)$;
else //node i_j is a parent of leaves
$L_{ll} = \mathit{left\_most\_leaf}(i_j, |D_j|)$;
$L_{lr} = L_{ll}$;
$L_{rl} = \mathit{right\_most\_leaf}(i_j, |D_j|)$;
$L_{rr} = L_{rl}$;
//get the number of query points that are in the left subtree of i_j
$N_l = |[L_{ll}, L_{lr}] \cap [l_j, h_j]|$;
//get the number of query points that are in the right subtree of i_j
$N_r = |[L_{rl}, L_{rr}] \cap [l_j, h_j]|$;
//use Lemma 7.3
$\mathit{contribution} = \mathit{contribution} \times (N_l - N_r)$;
return *contribution*;

Function $\mathit{left\_most\_leaf}(i, N)$
//compute the label of the left most leaf for subtree of node i
//in an error tree with N leaves
//We denote the binary representation of i by $(i)_2 = (b_{n-1} \ldots b_2 b_1 b_0)$,
//where $n = \log N$, $b_j \in \{0, 1\}$ for $0 \leq j \leq n-1$.
if $(i = 0)$ or $(i = 1)$ //special cases
then
$L = 0$;
return L;
//p is the position of the left most 1 bit of $(i)_2$
$p = \max\{k \mid b_k = 1, 0 \leq k \leq n-1\}$;
$L = b_{p-1} << (n - p)$; //$<<$ is the bitwise left shift operator
return L;

Function *right_most_leaf*(i, N)
//compute the label of the right most leaf for subtree of node i
//in an error tree with N leaves
//We denote the binary representation of i by $(i)_2 = (b_{n-1} \ldots b_2 b_1 b_0)$,
//where $n = \log N$, $b_j \in \{0, 1\}$ for $0 \leq j \leq n-1$.
  if $(i = 0)$ or $(i = 1)$ //special cases
    then
      $R = N - 1$;
      return R**;**
  //p is the position of the left most 1 bit of $(i)_2$
  $p = \max\{k \mid b_k = 1, 0 \leq k \leq n-1\}$;
  $L = b_{p-1} << (n - p)$; //$<<$ is the bitwise left shift operator
  $R = L + 2^{n-p} - 1$;
  return R**;**

Function *left_child*(i)
//compute the label of the left child of a nonroot node i in an error tree
  return $2i$**;**

Function *right_child*(i)
//compute the label of the right child of a nonroot node i in an error tree
  return $2i + 1$**;**

Theorem 7.3 *For a given aggregation query of form Sum*$(l_1{:}\, h_1, \ldots, l_{d'}{:}\, h_{d'})$*, the approximate query answer can be computed based upon the top k coefficients in the compact data cube in $O(d \times \min\{k, 2\prod_{1 \leq i \leq d'} \log |D_i|\})$ CPU time using a $((d+1)k)$-space data structure.*

Proof Sketch: We only need to process the first k coefficients in R. Each of the coefficients is a $(d+1)$-tuple of the form (7.3), so the space complexity is $(d+1)k$. The CPU time complexity follows easily from that of the two functions. An alternate mechanism is to process only the coefficients needed, which are at most $2\prod_{1 \leq i \leq d'} \log |D_i|$. □

Often the first k coefficients of the compact data cube reside in internal memory. If instead they are on disk, they occupy $\lceil dk/B \rceil$ disk blocks, which is typically one or a small constant, so they can be retrieved with a constant number of I/Os. In terms of CPU time, the quantity $2\prod_{1 \le i \le d'} \log |D_i|$ is almost always larger than k in practice, so the faster and simpler way to evaluate the approximation takes $O(dk)$ CPU time. By comparison, the CPU running time is $2^{d'}$ times faster than the algorithm in Chapter 6!

Our algorithm has the useful feature that it can progressively refine the approximate answer with no added overhead. If a coefficient contributes to a query, its contribution can be computed independently of the other coefficients. Therefore, to refine a query answer, the contribution of a new coefficient can be added to the previous answer in $O(d)$ CPU time, without starting over from scratch.

7.5 Experiments

7.5.1 Data Description

In many OLAP applications, the data have high dimensions and the correlations among the functional attributes and the measure are intricate and do not match artificial data models. To make our experimental results meaningful, we performed our experiments using both real-world data and synthetic data of high dimension. For brevity, we report the accuracy of our approximate query answers for real data only. To analyze the speed of our compact data cube construction algorithm, we report the running time of our algorithm on tunable synthetic datasets.

We obtained our real-world data from the U.S. Census Bureau using their Data Extraction System (DES) [13] and the data are as described in Section 6.4.2.

Our synthetic datasets are generated using our own data generation model. We use different combinations of parameters to generate a wide variety of synthetic

Notation	*Definition*
d	the number of dimensions
$\|D_i\|$	the size of the ith dimension, $1 \leq i \leq d$
n_region	the number of dense regions in multidimensional space
$n1\_region$	the number of type 1 dense regions
$n2\_region$	the number of type 2 dense regions
T	the sum of all the nonzero values
Z	the Zipf parameter for the value distribution dense regions
z_min	the minimum Zipf parameter for type 2 dense regions
z_max	the maximum Zipf parameter for type 2 dense regions
V_min	the minimum volume of a dense region
V_max	the maximum volume of a dense region
$noise\_volume\_level$	% of the number of nonzero entries outside dense regions w.r.t to the total number of non-zero entries
$noise\_weight\_level$	% of the sum of the nonzero values outside dense regions w.r.t. T

Table 7.1: Description of the synthetic data.

datasets for our timing experiments. We model the distribution of the nonzero values in a multidimensional array by the parameters defined in Table 7.1.

The program will generate the sparse representation of a d-dimensional array whose size is determined by the parameters $|D_i|$s, for $1 \leq i \leq d$. The nonzero entries are mainly located in n_region dense regions. The center of each dense region is a randomly picked position in the d-dimensional array. The *volume* of any dense region (which is defined as the number of cells in the region) is a random number between V_min and V_max. Each region has a sum which is the summation of the values for all the cells contained in the region. The Zipf distribution parameter Z, together with $T(1 - noise\_weight\_level)$ and n_region, are used to generate values which are randomly assigned to each region as its sum.

A dense region can be either type 1 or type 2. A type 1 region has a distribution where all dimensions are independent of one other and obey the unbiased binomial distribution. To generate a type 2 region, we first use the Zipf distribution with

parameter z (where z is uniformly chosen from $[z\_min, z\_max]$) to generate a set of values. The values are then assigned to the cells in the region in such a way that the closer a cell is to the center, the bigger the assigned value is.

We also consider the fact that besides the nonzero cells in the dense regions, there might be some isolated nonzero cells outside those dense regions. The number of such cells is defined by the parameters *noise_volume_level* and the sum of these isolated nonzero values are defined by *noise_weight_level*. Their positions are generated in a random way.

7.5.2 Efficiency of the Compact Data Cube Construction Algorithm

We implemented our compact data cube construction algorithms using the *Transparent Parallel I/O Programming Environment* (TPIE) system [88, 87, 86]. TPIE is a collection of templated functions and classes to support high-level and efficient implementations of external memory algorithms. The basic data structure in TPIE is a *stream*, representing a list of objects of an arbitrary type. The system includes I/O efficient implementations of algorithms for scanning, merging, distributing, and sorting streams, which are building block for our algorithms.

We did our experiments on a Digital Alpha workstation running Digital UNIX 4.0, with 512MB of internal memory. Since the sizes of the raw data sets used in our experiments are relatively small (44MB to 1GB), we restricted the amount of internal memory used by our program to be in the range from 1MB to 10MB. For all the runs using Algorithm I, the logical block transfer size used by TPIE streams was 256KB (32 times the physical disk block size 8KB) in order to achieve a high transfer rate. Smaller logical block sizes resulted in slightly longer run times. However, for all the runs using Algorithm II, we used a smaller logical block transfer size of 16KB in order

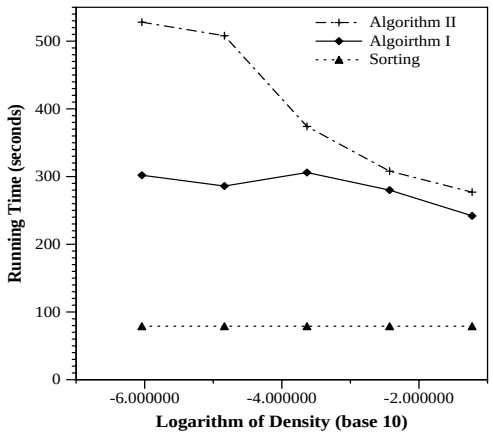


Figure 7.2: Construction time vs. density.

to keep the number g of passes small.

In the first group of experiments, we use synthetic data and measure the elapsed time of the compact data cube construction algorithms as a function of the data density N_z/N. We fix the number of dimensions d, the number of nonzero entries N_z, and the internal memory size M.

Figure 7.2 depicts the results from one set of the experiments, in which we fix $d = 10$ and $N_z = 10^6$. The size of each data item is 44 bytes, and the internal memory size parameter M is set to 190650 (corresponding to 8MB). The size of the sparse representation of the raw data is 44MB. By changing the dimension size parameters $|D_i|$, we obtain multidimensional arrays with different sizes N in the range from $16 \times N_z$ to $2^{20} \times N_z$. This corresponds to the densities in the range from 0.06 to 10^{-6}. We partition the dimensions according to (7.4). For all data sets, we have $g = 2$, although the ones with small density have very big values of N. For example, the data set with density of 10^{-6} has array size $N \approx 2^{40}$.

We ran our compact data cube wavelet decomposition algorithms against five data sets. The results are shown in Figure 7.2. To make the plot clear, we plot the logarithm of density on the x axis. The x coordinate $x = -i$ corresponds to a density of 10^{-i}. As we can see from the figure, the running time of Algorithm I for the five

data sets varies slightly (in the range from 242 seconds to 306 seconds) as the density changes. The differences in running time are mainly caused by the effect of the online cutoff. For some runs, slightly more than N_z coefficients are written out during the first pass, which causes a longer running time, whereas it is the other way around for some other runs. The running time of Algorithm II decreases significantly as the density increases. The reason for the decrease is that N_z is fixed, and a data set with small density corresponds to a big N value. Since Algorithm II cannot apply the more liberal group partitioning of (7.4), the resulting g value is large. On the other hand, Algorithm I takes advantage of (7.4) for partitioning the groups and performs noticeably better than Algorithm II for very sparse data.

Our methods require less time and storage space than do other methods. For example, for the data set with the smallest density $1/10^6$, if we use the partial-sum data cube approach of [37], we will need to process and store a partial-sum cube that contains 10^{12} nonzero cells and takes up 4GB storage space if we use the MOLAP (array) representation, even though the raw data size is only 44MB. If we use the compact partial-sum data cube approach in Chapter 6, the final compact partial-sum data cube is much smaller in size than that of the raw data, but there will be time and space problems during the construction stage because of the need to compute the dense partial-sum data cube during the wavelet decomposition.

In the second group of experiments, we measure the elapsed time in terms of the raw data size. In each set of the experiments in this group, we fix the number of dimensions d, the density N_z/N, and the amount of internal memory M. By changing N_z and N proportionally, we obtain data sets with the same density but different size. Figure 7.3 plots the result of one set of the experiments, in which we use $d = 10$, data item size of 44 bytes, $M = 190650$ (corresponding to 8MB), and a density of 0.001. The value N_z varies from 1 million to 16 million, corresponding to

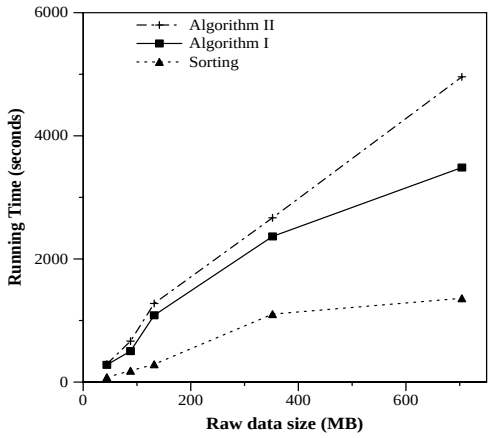


Figure 7.3: Construction time vs. input raw data size.

a raw data size from 44MB to 704MB. From Figure 7.3, we can see that the running time of both algorithms scales almost linearly with respect to the input data size.

In all the experiments, the initial order of the data is in the order needed for the first pass of the data cube construction algorithms. We also graph in Figures 7.2 and 7.3 the time it takes to sort each data set using TPIE, so that the speed of the data cube construction algorithms can be assessed in terms of the time it takes to sort a similarly sized file. Note however that the TPIE sorting routine has an extra speed advantage in that it has been carefully optimized. Algorithm I was fairly easy to program since it makes use of the TPIE scan and sorting operations. However, it can be optimized further by performing the first pass of each transposition step during the actual wavelet decomposition, as suggested in Section 7.3.1. Algorithm II should also be further optimized; our implementation did not make use of double buffering.

7.5.3 Accuracy of the Approximate Answers

In this section, we compare the accuracy of our method with that of the traditional random sampling method in answering typical range-sum queries. The simplest way of using random sampling is, during the offline phase, to take a random sample of a

certain size from the raw data. When a query is presented in the online phase, the query is evaluated against the sample, and an approximate answer is given in the obvious way: If the answer of the query using a sample of size t is s, the approximate answer is $s \times N_z/t$. The new sampling-based summary statistics proposed in [27] cannot be applied here to any advantage since our raw data do not contain duplicate tuples. We chose not to do any comparisons with traditional histogram methods [67, 66], because as we mentioned in Section 7.1, they are too inefficient to construct for high-dimensional data that cannot fit in internal memory.

The relative effectiveness of random sampling and that of our method are fairly constant over a wide variety of synthetic data sets and range-sum query sets. Our compact data cube generally provides more accurate results than that of a random sample of the same size. If the locations of the nonzero entries are uniformly distributed in the multidimensional array, random sampling may perform better. But uniform distributions in real-life data warehouses are rare.

We measure the accuracy of the methods by using both the real data and synthetic data. For the brevity of this chapter, we present the results from a typical set of our real-data experiments. (The other experiments were qualitatively similar.) In the experiments, we specify partial ranges on $d' = 3$ of the $d = 10$ dimensions and average our results over the following d'-dimensional range-sum queries:

$$\{Sum(l_1{:}\, h_1, l_2{:}\, h_2, l_3{:}\, h_3) \mid h_i = l_i + \Delta,\ 0 \le l_i \le h_i < |D_i|,\ \text{for each } 1 \le i \le 3\}$$

where Δ is a nonnegative integer constant.

Figure 7.4 plots the accuracy of our method in comparison with random sampling for different error metrics and various storage space sizes k. We used $\Delta = 10$. The absolute errors are normalized by the largest exact answer $L = 766327$ for the query set. The sampling results are the averages for five different runs. The storage size is measured in terms of the number k of wavelet coefficients (for our method) or the

number of sample points (for sampling) used in answering the queries. The wavelet coefficients and the sample points are each of form (7.3), which is a $(d+1)$-tuple; since $d = 10$ for our data set, each wavelet coefficient (sample point) is represented by a tuple of 11 numbers.

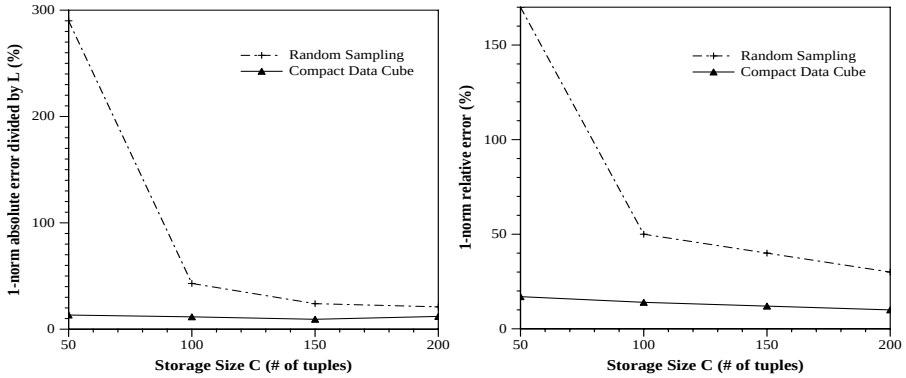
As shown in Figure 7.4, the accuracy of our compact data cube is noticeably better than that of random sampling. For example, when using only $k = 50$ coefficients in our compact data cube method, the average relative error for the query set is only 17%, and the average relative error is about 10 times better than for random sampling.

7.6 Conclusions

In this chapter, we have presented an efficient and effective technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the underlying multidimensional data in OLAP applications.

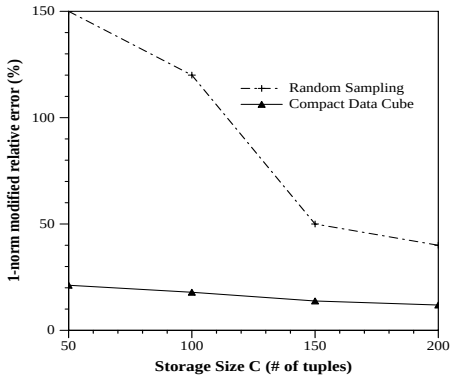
Our compact data cube construction algorithms are very efficient in term of I/O complexity, especially when the raw data are very sparse. In the online phase, by using the hierarchical structure of the wavelet decomposition, we obtain an efficient algorithm for answering typical OLAP aggregation queries. Experiments show our compact data cube provides excellent approximation for online range-sum queries with limited space usage and little computational cost.

As future work, we would like to consider alternative normalization and thresholding methods based upon more sophisticated probability distributions of query patterns. To get further improvements in the space-accuracy tradeoff, we will work on quantizing the wavelet coefficients and entropy encoding of the quantized coefficients. We are developing dynamic efficient algorithms for maintaining the compact data cube, given updates in the underlying raw data. We will also work on applying

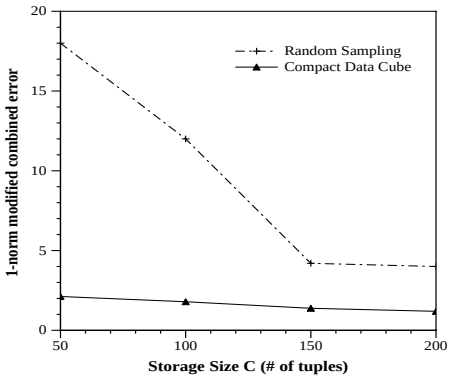


(a) 1-norm absolute error

(b) 1-norm relative error



(c) 1-norm modified relative error



(d) 1-norm modified combined error, $\alpha = 1.0$, $\beta = 10.0$

Figure 7.4: Accuracy of approximate query answers for the compact data cube and for random sampling.

our techniques to other operators (e.g., *projection*) other than *Sum*.

Chapter 8

Selectivity Estimation in the Presence of Alphanumeric Correlations

Almost all previous work on selectivity estimation dealt with the estimation of numeric selectivity, i.e., the predicates contain only numerical variables. The more general problem of estimating alphanumeric selectivity has attracted attention only very recently, due to the work of Krishnan, Vitter and Iyer [45, 46].

We generalize the work of [45, 46]. We consider the problem of selectivity estimation for the case when two alphanumeric columns are involved. As in [45, 46], we use suffix tree as our basic data structure. We introduce a *dynamic graph construction problem* to characterize the possible complex correlation between the two columns. We develop efficient algorithms to collect statistics about the two columns and their correlation and store it in a dynamically constructed graph of reasonable size that we propose to be stored in the database catalog. The graph will be used for selectivity estimation according to various heuristic strategies. We have conducted experiment to validate our algorithms.

This chapter is organized as follows: In Section 8.1 we describe the problem and review previous work. In Section 8.2 we introduce the dynamic graph construction problem and formulate our selectivity estimation problem accordingly. Section 8.3 is a general description of our method. In Section 8.4 we describe the offline phase, i.e., the construction of a small version of a huge graph. In Section 8.5 we discuss many strategies which will be used to do actual estimation. In Section 8.6 we propose a method to build a different kind of catalog. We reduce the problem to the least square problem and show that it can be solved very efficiently. In Section 8.7 we

report our experimental results. Finally, in Section 8.8 we suggest some problems for future research.

8.1 The Problem and Previous Work

8.1.1 The Problem

Suppose we have a relational database which has a table *PART* as shown in Table 8.1. A record (a row) in *PART* stores information about a particular part. A part has many characteristics, e.g., it may have a color and a shape. Each characteristic is represented by an attribute of *PART*. Suppose *PART* has attributes *Color* and *Shape*, among others.

PID	*Price* ($)	*Sales*	*Color*	*Shape*
p1	2	120	dark_green	small_circle
p2	3	100	light_green	large_circle
p3	12	20	dark_red	small_square
p4	8	80	light_blue	medium_rectangle
...	...	...	...	...

Table 8.1: *PART*: a table (relation) in a database.

If we want to find out all parts which have color green and shape circle, we may use the standard SQL operator `LIKE` to issue following query:

```
SELECT *
FROM PART
WHERE Color LIKE green AND Shape LIKE circle
```

If we want to find out all parts whose colors are green-like and whose shapes are circle-like, we may issue the following query:

```
SELECT *
FROM PART
```

`WHERE` *Color* `LIKE` *green* `AND` *Shape* `LIKE` *circle*

where "$*$" represents the wildcard, i.e., $*$ represents any (empty or nonempty) character string.

For a given query, the optimizer will need to estimate how many rows in the relation satisfy the predicate specified by the query. To simplify our discussions for alphanumeric attributes, we restate the selectivity problem as follows.

Let F be an M by N table, i.e., F is a relation with M rows and N columns. Each row represents one tuple of a table, and each column represents one field of a specific type. For example, a column might be of numeric type, which indicates that all data in this column must have a numeric value. In this chapter we are mainly interested in data of alphanumeric type and we assume that all columns are of alphanumeric type. We use $F(i,*)$, $F(*,j)$ and $F(i,j)$ to denote the ith row, jth column, and (i,j) element of F, respectively. Note that any element $F(i,j)$ is a character string.

Suppose we are given a predicate P, which is a set of patterns,

$$P = \{(i_1, p_{i_1}), (i_2, p_{i_2}), \ldots, (i_k, p_{i_k})\},$$

where each i_j $(1 \leq i_j \leq N)$ indicates one column and each p_{i_j} is a pattern string. We define $F(P)$ as the set of rows which contain all the given patterns in the corresponding columns, i.e.,

$$F(P) = \{F(i,*) | F(i,i_j) \text{ contains } p_{i_j} \text{ as a substring, } 1 \leq j \leq k\}.$$

Then the selectivity of predicate P is defined as

$$s(P) = \frac{|F(P)|}{M}.$$

Our goal is to estimate $s(P)$ for any given query predicate P, i.e., we want to estimate the number of rows which contain the given patterns.

We study the special case when $N = 2$, i.e., we only consider two columns.

8.1.2 Suffix Tree and the Krishnan-Vitter-Iyer Method

Krishnan, Vitter and Iyer have developed a method (henceforth referred to as the KVI method) to estimate alphanumeric selectivity for the one-column case. In the following we briefly describe their method. (For more details, see [45, 46].)

KVI method is based on the key data structure of *suffix tree*. Let s be a string of length l. We use s_i to denote the suffix of s starting at position i, $0 \leq i \leq l-1$. (We count the positions starting from position 0 on.) We use $s' \propto s$ to denote that string s' is a substring of string s.

A suffix tree T for string s is a rooted tree that has the following properties:

1. Every edge (u, v) has a substring s' of string s as its label. We call that s' is associated with the edge (u, v). We also call that s' is associated with the node v.
2. For any node v, the first symbols of the substrings associated with v's children are all different.
3. For any node v, let $path(v)$ be the path from the root to v. The string represented by v, denoted by $p(v)$, is the concatenation of the strings associated with the edges along $path(v)$. We also say that $p(v)$ corresponds to node v, and v is the *locus* of $p(v)$. Any suffix of string s is represented by a unique leaf node of T.
4. Since any substring of s is the prefix of a suffix of s, for any substring σ of s, we can find a unique node v such that σ is the prefix of $p(v)$, and σ is not a prefix of the parent of v. We call v the *extended locus* of σ, and we can denote it by $v(\sigma)$.

For a suffix tree T and a string s, if we can find a path from the root to a node v whose corresponding string contains s as a prefix, i.e., we can find an extended locus of s, then we say we match s in T successfully.

The above definition can be generalized to a set of strings in the obvious way. Moreover, for a set of strings S, when constructing its suffix tree T, for each node v that represents a substring $p(v)$, we can assign a count $c(v)$ that is the number of strings in S that contain $p(v)$ as a substring to v.

For more details about suffix trees and their generalization, see [54, 7].

There are two phases for the KVI method. In an *offline* phase, a suffix tree T for F is built. Each node of T corresponds to some substring that occurs in some row in F, and vice versa. For a substring x, we denote its corresponding node in T by $v(x)$. The number of rows in F that contain x as a substring is denoted by $c(x)$, and we shall call it the *exact count* of x.

In the actual construction, not all substrings in F can be represented by nodes in T due to the space limitation. The tree T is continuously pruned so that the overall storage does not exceed an allocated bound. Consequently, substrings with very small true counts don't have corresponding nodes in T. If a substring x does have a corresponding node in T, it may have a count $\widehat{c}(x)$ that is an approximation of the exact count $c(x)$. In subsequent discussions, we will ignore the possible errors introduced by the pruning during the construction and use $c(x)$ to denote either the approximate or the accurate value of $c(x)$.

When T is constructed, further pruning is done. For a given *prune count* p, if $c(x)$ is smaller than p, we (informally) call x a *light substring* and call $v(x)$ a *light node*. All light nodes will be pruned off. The remaining tree is the so-called *catalog* , which is the output of the offline phase. (Strictly speaking, the data structure to be stored in the catalog.) We still use T to denote the final pruned tree.

In the *online phase*, when a predicate "`LIKE` $*p*$" is given, by running the KVI method, an estimated selectivity for the pattern $*p*$ is obtained based on the catalog, that is, an estimation on the number of rows in F that contain p as a substring is

obtained.

Since there is a strict limitation on the size of the catalog, many "infrequent" nodes in T have to be pruned out. So it would happen quite often that a given pattern cannot be matched by any node in the pruned T. The mismatch does not necessarily imply that that pattern never occurs in F, and we need to estimate the selectivity of these "unlikely" substrings using heuristics. In the KVI method, several *matching strategies* have been suggested to deal with these mismatches.

The main advantage of using the suffix trees is that it allows very efficient storage of all substrings. Actually the space complexity is linear in the total length of the represented strings. But still, we cannot store the whole suffix tree since the database catalog size is rather limited. For example, in [45, 46], for a table of 200K rows with each row having an average of 40 characters, a suffix tree of about 450K nodes is maintained in the offline phase, and the storage needed is about 3000KB. The big tree needs to be pruned significantly due to a catalog size of 0.5KB–1KB and the final catalog tree contains only about 100 nodes.

8.2 Problem Formulation

In this section we introduce the *dynamic graph construction problem*. We then formulate the offline phase of our two-column selectivity estimation problem as a dynamic graph construction problem.

8.2.1 The Dynamic Graph Construction Problem

A (directed/undirected) graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of (directed/undirected) edges. A *bipartite graph* is an undirected graph in which V can be partitioned into two sets V_1 and V_2 such that any edge has its one end vertex in V_1 and the other end vertex in V_2. A *weighted graph* is a graph

$G = (V, E)$ is a graph for which each edge has an associated *edge weight* given by a *edge weight function* W_E and each vertex has an associated *vertex weight* given by a *vetex weight function* W_V. We only consider non-negative integer weights, i.e., $W_E : E \to N$ and $W_V : V \to N$. For a weighted graph, let c_V and c_E be two given threshold values. Any vertex with weight greater or equal to c_V is called a *heavy vertex* and any edge with weight greater or equal to c_E is called a *heavy edge* . Other vertices (edges) are called *light vertices* (*edges*). The two values c_V and c_E are called *vertex threshold* and *edge threshold*, respectively. For a graph G and given threshold values c_V and c_E, we delete all light edges. We also delete any vertex that is neither heavy nor incident to any heavy edge. The resulted graph is called the *heavy subgraph with respect to the specified threshold values.* (For precise graph theory terminologies, refer to [20].)

We consider six operations on a graph: insert/delete a vertex v, insert/delete an edge e, and change the weight of a vertex or an edge. When deleting a vertex v, all edges incident to v will also be deleted.

Suppose there is a weighted undirected graph that has a discrete time parameter t, i.e., at any time t, there is a graph $G(t) = (V(t), E(t))$ with weight funcations $W_V(t)$ and $W_E(t)$. At time 0, there is an initial graph $G(0)$ that is usually empty. Suppose at time t, we are given $G(t-1)$ and a set of operations O_t. Then $G(t)$ is defined as the graph obtained by applying all operations in $O(t)$ (possibly in some specified order) to $G(t-1)$.

Now we define the *Dynamic Graph Construction Problem* (DGCP for short). Let c_V and c_E be the given threshold values. Let $G(t)$ be a given initial graph. Let $O(t)$ be a set of operations at time t. At time t, only $\{O(i) \mid i \leq t\}$ are known. The goal is to construct the heavy subgraph of $G(n)$ with respect to c_V and c_E for any nonnegative interger n.

The seemingly most natural method to solve DGCP is deleting all light edges and/or light vertices after the whole $G(n)$ has been constructed. As we will see, this is not preferable and even impossible for very large database applications because $G(n)$ will become huge and we simply do not have enough space to store $G(n)$. Instead, we will keep pruning off light edges and/or light vertices during the construction process. Consequently, the resulted subgraph will be an approximation to the true solution.

For our problem, we have a table with two columns. We will construct two suffix trees, one for each column. At the same time, we maintain all relations between the two columns by creating links between the two suffix trees.

We are now ready to describe our problem formally.

8.2.2 The Problem: Revisited

Let F be an $M \times 2$ table as described in Section 8.1.1. Let $S_j = \{F(i,j) \mid 1 \leq i \leq M\}$ be the set of strings contained in column j, for $j \in \{1,2\}$. For any $0 \leq k \leq M$, let $T_j(k)$ be the suffix tree formed by the strings in the first k rows of column j, and let $V_j(k)$ be the set of nodes of $T_j(k)$, for $1 \leq j \leq 2$.

We define the weighted graph $G(k) = (V(k), E(k))$ where

$$\begin{aligned} V(k) &= V_1(k) \cup V_1(k), \\ E(k) &= \{(v_1, v_2) \mid v_1 \in V_1(k), v_2 \in V_1(k), \text{and} \\ &\qquad \exists i, 1 \leq i \leq k, \text{ s.t. } p(v_1) \propto F(i,1), p(v_2) \propto F(i,2)\}. \end{aligned}$$

The weight functions are defined as

$$\begin{aligned} W_V(k, v) &= \text{the count of } v \text{ in } T_j(k), \text{ for } v \in V_j(k) \\ W_E(k, (v_1, v_2)) &= \text{the number of rows that satisfy } p(v_1) \propto F(i,1), \\ &\qquad p(v_2) \propto F(i,2), \text{ for } 1 \leq i \leq k, (v_1, v_2) \in E(k). \end{aligned}$$

Let $G(0)$ be an empty graph. Let c_V and c_E be two threshold values to be determined later. The goal of the offline phase is to construct the heavy subgraph of $G(M)$ with respect to c_V and c_E.

Each node v in the suffix trees has its own node weight $c(v)$ which is obtained in the usual way, i.e., each suffix tree is constructed based on one column and the node counts are obtained consequently. The tree edges in each suffix tree are not considered as graph edges, although we will need to use them to trace tree paths.

Graph $G(M)$ is usually huge. For example, if we have a table of 200K rows and two columns, and each column contains about l characters, then each suffix tree is of size $O(Ml)$ and G_M may contain $O(M^2l^2)$ edges. Suppose each tree has 300K–450K nodes, then there may be about 9×10^{10}–20×10^{10} edges.

8.3 Our Method: A General Description

Our method has two phases: an offline phase and an online phase. The goal of the offline phase is to construct a catalog of reasonable size that will be used in the online phase to do the selectivity estimation.

As in [45, 46], we use suffix tree as our basic data structure. In the offline phase, we first construct two big suffix trees, one for each column. The correlations between those two columns will be represented by weighted edges between the two trees. Then we prune the trees to reduce their size significantly and obtain two much smaller trees. The correlation information of the two trees is stored in a matrix of reasonable size. The two small trees and the associated matrix constitute the catalog.

As mentioned before, the simplest method (Method I) is to construct the whole graph G_M during one scan of the table. But Method I is not practical due to storage limitation. Even though a large amount of storage is available during the offline phase and we can easily store the two separate suffix trees, we cannot afford to store

all the edges between the two trees.

We may use the following two-pass alternative (Method II). We scan the table twice. During the first scan, we construct two suffix tree T_1 and T_2 independently, where T_i is for column i, $i \in \{1, 2\}$. When the first scan is finished, each node in the trees has a node weight. Based on some threshold value, we prune the two trees and obtain two smaller trees (we still denote them by T_1 and T_2, respectively). During the second scan, we establish the desired relationship. Specifically, for any row, if substring x occurs in column 1, substring y occurs in column 2, and both x and y have corresponding nodes $v(x)$ and $v(y)$ in T_1 and T_2 respectively, then we build a weighted edge between the two nodes. If such an edge already exists, we simply modify its weight.

Because the end nodes of a heavy edge are likely to be heavy too, Method II can simulate Method I very well. The drawback of Method II is that it need to scan the table twice.

Our method is an *one-pass approximate* implementation of Method II. We partition the whole table into several sections and bring each section into the internal memory only once. Once we have one section in the internal memory, we can do some preprocessing and obtain all necessary information. Those information is then used to update the current approximate solution.

More specifically, during the scan, we maintain two suffix trees of reasonable size, together with a matrix that represents the current relationship between the two trees. The two suffix trees and the matrix constitute an approximate solution. They are updated periodically, i.e., they are updated once every section is brought into the internal memory and preprocessed.

Our method can be used to solve the general DGCP. We can build the dynamic graph by periodically pruning out light vertices/edges. The method will work well if

the node weight and edge weight are correlated "smoothly", i.e., the end vertices of heavy edges are likely to be heavy.

In the online phase, we use the output of the offline phase to do selectivity estimation. Since the catalog is very small, not all information about the graph G_M is recorded accurately and we must use some heuristic strategies to do the estimation. We mainly adapt the various strategies of [45, 46] and generalize them to the two-column case.

8.4 The Offline Phase

In this section we describe the details of the offline phase. The input to the offline phase is an M by 2 table F, where M is very large. Suppose l_i is the maximum length of any string that occurs in column i.

We maintain five data structures during the offline phase: two suffix trees T_1 and T_2, each of size m; an $m \times m$ matrix R; and two arrays of pointers L_1 and L_2. Here m is a parameter determined by the storage available. (In our experiments, we choose $m = 1000$.)

We choose another parameter n that is also determined by the storage available. (In our experiments, we choose $n = 1000$). We preprocess F section by section, with each section containing n rows. At any time, the sections already being preprocessed form a *view* of F.

Each node v in T_1 (T_2) has a node count $c(v)$ that is the number of rows in the current view of F that contain $p(v)$ as a substring in the first (second) column. Each matrix entry $R(u, v)$ is an approximation to the number of rows in the current view of F that contains $p(u)$ in column 1 and $p(v)$ in column 2. We call $R(u, v)$ the *common count* between $p(u)$ and $p(v)$. (The node counts are node weights and the common counts are edge weights.)

When we scan a section, we construct two arrays L_1 and L_2, both of size n, where each $L_i(j)$ ($i \in \{1,2\}$ and $1 \leq j \leq n$) is an array of size $h_i = l_i(l_i+1)/2$. Note that h_i is the upper bound of the number of substrings that can occur in any row of column i. Array $L_1(j)$ ($1 \leq j \leq n$) records pointers to all suffix tree nodes in T_1 that correspond to some substring in column 1 of row j of the current section. Array L_2 is defined similarly for column 2.

Suppose we already have the current versions of T_1, T_2, R, L_1, and L_2. When scanning the next section, for each row j, we insert all suffixes of column i ($i \in \{1,2\}$) into tree T_i, and record the corresponding pointers in $L_i(j)$. When the scan of this section is completed, both suffix trees grow (significantly) with the node weights updated. Now we prune out all light nodes and keep only the heaviest m nodes in each tree. Note that the pruning may cut off some old nodes and add some new nodes. Relabeling is done to make sure that all nodes receive a unique label that is between 1 and m. The goal is to maintain two suffix trees T_1 and T_2 so that T_i ($i \in \{1,1\}$) contains the heaviest m nodes with respect to column i of the current view of F.

We then update the matrix R according to the two pointer arrays L_1, L_2 and the updated suffix trees. (As before, we use $R(u,*), R(*,v)$, and $R(u,v)$ to denote the uth row, vth column, and (u,v) element of R, respectively.) Specifically, for each j ($1 \leq j \leq n$), for any pointers $p_1 \in L_1(j)$ and $p_2 \in L_2(j)$, if p_1 points to an old node u in T_1 and p_2 points to an old node v in T_2, we increase $R(u,v)$ by 1; if p_1 points to a new node u in T_1 and/or p_2 points to a new node v in T_2, we delete the old entries $R(u,*)$ (and/or $R(*,v)$) and insert the new entries. When finished with all n rows, we obtain a new matrix R that records the correlation between the nodes in the updated suffix trees. We clear the two arrays L_1, L_2 and proceed to the next section.

After the scan is finished, we obtain two suffix trees T_1 and T_2 of size m and an $m \times m$ correlation matrix R. We further prune these trees and reduce the size of R. Specifically, we prune off all light edges and light nodes according to some threshold values c_E and c_V, We only maintain the corresponding entries for the left edges in R. In our implementation, the value c_V is chosen in such a way so that after the pruning, there are no more than 100 nodes remaining in either suffix tree. (Note that if all incident edges of a node have been pruned off, the node will also be pruned off).

In our implementation we have modified the above method a little. In pruning the two trees, we delete a node only if all its incident edges are light.

Finally, we obtain the following: two suffix trees T_1, T_2 of sizes t_1 and t_2 respectively, and a $t_1 \times t_2$ matrix R, where bothe t_1 and t_2 are no more than 100. (For simplicity, we assume $t_1 = t_2 = 100$.) T_1, T_2 and R constitute the output of the offline phase.

The final catalog has a very interesting and useful property due to the special implementation of the pruning, that is, if substring x has a corresponding node u in T_1 and substring y has a corresponding node v in T_2, then the matrix entry $R(u, v)$ gives a good estimation $R(u, v)/M$ for $s(x, y)$. If we prune off all light edges, it is possible that x, y have corresponding nodes u, v in the trees but the entry $R(u, v)$ does not reflect the common count between x and y.

To further reduce the size of the catalog, we can use adjacent lists to represent the matrix R. In our experiments, however, R is very dense. Moreover, matrix representation allows faster access. So we include matrix R as part of the catalog.

8.5 The Online Phase

In the online phase, a query of form $P = (p_1, p_2)$ is given. We need to estimate the selectivity $s(P)$ based on the information contained in the catalog that consists of two

small suffix trees T_1, T_2 and a matrix R. Note that the catalog is an approximation to the heavy subgraph of the weighted graph G_M.

We will use the following general strategy. First, we search the two patterns p_1 and p_2 in T_1 and T_2, respectively, trying to match them with some nodes in the trees. If we can find two matching nodes, say node u for p_1 and node v for p_2, we return $R(u, v)/M$ as an estimation of $s(P)$.

In most cases we cannot find exact matches for the two given patterns. We will use some *mismatch strategy* to estimate $s(P)$.

In [45, 46] many mismatch strategies are developed for the one-column cases. Our main job here is to adapt those strategies to cope with two-column cases.

As in [45, 46], there are two classes of mismatch strategies, most of them are based on a *greedy parsing* of the given query patterns.

Let T be a suffix tree and σ be a string. Let σ_i be the suffix of string σ starting at position i, $0 \leq i \leq |\sigma|$. We search T, starting from the root, and try to find the longest prefix of σ that matches the substring represented by a node in T. Let this prefix be $\sigma(0)$. Next we start from the root again, try to search for the longest match for the prefix of the remaining substring, and so on. If at some time we cannot find any match right for any prefix of the remaining substring we take the first character of the remaining string as a result and continue.

The parsing results is a partition of σ, $\sigma = \sigma(0)\sigma(1)\cdots\sigma(m)$, where $|\sigma(i)| > 0$ for all $0 \leq i \leq m$, and each $\sigma(i)$ is either the maximal match of σ_j in T ($j = |\sigma(0)| + \cdots + |\sigma(i-1)|$), or else the single character at position j of σ if no non-null prefix of σ_j successfully matches in T.

For a given query pattern $P = (\alpha, \beta)$, we will use $e(X; P)$ to denote the estimate of $s(P)$ using mismatch strategy X. When X is known from the context, we omit X from our notation.

If we can find exact match node u for α in T_1 and nexact match node v for β in T_2, we can immediately obtain $e(P)$ by looking into the matrix entry $R(u, v)$. We use the following heuristics to handle mismatches.

8.5.1 The Independence-Based Strategies

In using independence-based strategies, we parse the given patterns into sub-patterns that can be matched in the two suffix trees. We then assume independence among all sub-patterns and give our overall estimation based on the selectivity of those independent matchable sub-patterns.

There are seven independence-based strategies. We denote them by $I_i, 1 \leq i \leq 7$.

Strategy I_1: Let α and β be greedily parsed as follows:

$$\alpha = \alpha(0) \cdots \alpha(m), \quad \beta = \beta(0) \cdots \beta(n) \tag{8.1}$$

Then $e(P)$ is defined as

$$e(P) = \prod_{i=0}^{m} \prod_{j=0}^{n} e(\alpha(i), \beta(j)),$$

where $e(\alpha_(i), \beta_(j))$ is the appropriate matrix entry in R divided by M (the number of rows in the relation). We assume that if either $\alpha(i)$ or $\beta(j)$ does not have an exact match in its corresponding tree, i.e., there is not a corresponding entry in R, we have $e(\alpha(i), \beta(j)) = 0$.

Basically, this strategy assume independence among different subpatterns.

Strategy I_2: It is the same as I_1 except that if $e(I_1; P) = 0$, we set $e(I_2, P) = c_E/M$ where c_E is the edge threshold value (also called *common prune count*). That is, we assign the common prune count to all "unlikely" patterns.

Strategy I_3: It is similar to I_1, but we modify the estimation of $e(\alpha(i), \beta(j))$: if either $\alpha(i)$ or $\beta(j)$ does not have an exact match in its corresponding tree, we set $e(\alpha(i), \beta(j)) = c_E/M$.

Strategy I_4: Suppose $|\alpha| = l_1$, $|\beta| = l_2$. Let $h_1 = l_1(l_1+1)/2$, $h_2 = l_2(l_2+1)/2$. Consider all suffixes α_i of α and all suffixes β_j of β, where $1 \leq i \leq l_1, 1 \leq j \leq l_2$. We define $e(I_4; P)$ as

$$e(I_4; P) = \sum_{i=0}^{l_1-1} \sum_{j=0}^{l_2-1} w_{i,j} e(I_1; \alpha_i, \beta_j),$$

where $w_{i,j}$ is a weight. In our implementation, we choose

$$w_{i,j} = \frac{l_1 - i}{h_1} \times \frac{l_2 - j}{h_2}.$$

The rationale behind this strategy is that we expect the weighted average selectivity of all suffixes would give a good estimation on the selectivity of the string itself. We weight the estimations of the longer suffixes more than those of the shorter suffixes.

Strategy I_5: In I_4, replace I_1 by I_2.

Strategy I_6: In I_4, we replace the weight function by

$$w_{i,j} = \frac{2^{l_1-i}}{h_1} \times \frac{2^{l_2-j}}{h_2},$$

and we also change h_1 and h_2 correspondingly, i.e.,

$$h_1 = 2^1 + \ldots + 2^{l_1} = 2^{l_1+1} - 2,$$

$$h_2 = 2^1 + \ldots + 2^{l_2} = 2^{l_2+1} - 2.$$

Strategy I_7: In I_6, replace I_1 by I_2.

8.5.2 The Child-Based Strategies

This class of strategies are based on estimations on the children patterns of the given patterns.

Suppose we are given two strings s and t. Let the first segments of the greedy parsings of s and t be $s(0)$ and $t(0)$, respectively. Let the corresponding node of $s(0)$ in T_1 be u, and let the corresponding node of $t(0)$ in T_2 be v. Suppose v has children $v_1, \ldots, v_k$. We define the quantity Δ as follows:

$$\Delta = R(u, v) - \sum_{i=1}^{k} R(u, v_i).$$

Note that Δ is an estimate of the sum of the counts of the pruned children of v with respect to u. We further define an *uncounted count* $uc(s, t)$ of s with respect to t:

$$uc(s,t) = \begin{cases} \Delta & \text{if } \Delta > 0 \\ c_E & \text{otherwise} \end{cases}$$

We then estimate the number of pruned children of v with respect to u as

$$nc(s,t) = \begin{cases} 0 & \text{if } \Delta < 0 \\ \lceil uc(s,t)/c_E \rceil & \text{otherwise} \end{cases}$$

Finally we have the following estimation on the selectivity of (s, t):

$$es(s,t) = \begin{cases} 0 & \text{if } nc(s,t) = 0 \\ \dfrac{uc(s,t)}{nc(s,c) \times M} & \text{otherwise} \end{cases}$$

Now we are ready to describe all the child-based strategies. There are four of them.

Strategy CE_1: We consider all the suffixes of the two given patterns α and β. Suppose the length of α is l_1 and the length of β is l_2. We estimate $s(P)$ according

to the following formula:

$$e(P) = \min_{0 \le i < l_1} \min_{0 \le j < l_2} \{es(\alpha_i, \beta_j)\}.$$

This strategy is based on the fact that if two strings occur in the same row, any suffix of the first string and any suffix of the second string must occur in that row as well. So the selectivity of (α, β) is at most that of (α_i, β_j), where α_i and β_j are suffixes of α and β, respectively.

Strategy CE_2: Consider the greedy parsings (8.1) of α and β. We then have

$$e(P) = \prod_{i=0}^{m} \prod_{j=0}^{n} \frac{es\left(\alpha(i), \beta(j)\right)}{e\left(\alpha(i), \beta(j)\right)}$$

where $e(\alpha(i), \beta(j))$ is the appropriate matrix entry in R divided by M (the number of rows in the relation). That is, we consider the children of all subpatterns as independent of each other.

Strategy CE_3: It is similar to CE_1. We still define

$$e(P) = \min_{0 \le i < l_1} \min_{0 \le j < l_2} \{es'(\alpha_i, \beta_j)\},$$

where

$$es'(\alpha_i, \beta_j) = \begin{cases} e(\alpha_i, \beta_j) & \text{if there is an exact match} \\ es(\alpha_i, \beta_j) & \text{otherwise} \end{cases}$$

Strategy CE_4: Consider the greedy parsing of (8.1). We use the following formula derived from Bayes rule to estimate $s(P)$:

$$e(P) = e\left(\alpha(0), \beta(0)\right) \prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \frac{es\left(\alpha(i), \beta(j)\right)}{e\left(\alpha(i), \beta(j)\right)}$$

8.6 Depth-Based Estimation

8.6.1 The Basic Method

All the previous discussions are based on a catalog consisting of two suffix trees T_1, T_2, both of size 100, and a 100×100 matrix R. The two trees and the matrix are obtained from two larger suffix trees of size m ($m = 1000$ in our implementation) and an $m \times m$ matrix. In this section, we produce a much smaller catalog based on the two larger suffix trees and the $m \times m$ matrix.

Suppose tree T_1 has depth d_1 and tree T_2 has depth d_2. For any $0 \leq i \leq d_1, 0 \leq j \leq d_2$, we consider the average weight of all edges that link a node of depth i in T_1 to a node of depth j in T_2. This average weight can be obtained easily from the m by m matrix. We store it as the (i, j) entry of a $(d_1 + 1) \times (d_2 + 1)$ matrix D. The matrix D is called the *depth statistics table* (DST).

Then we prune the two trees as before, obtaining two small suffix trees of size 100. We do not maintain the node weights anymore. Instead, we only need to maintain the structures of the trees. We still call them T_1 and T_2.

As a result, we obtain a new catalog that consists of T_1, T_2, and the DST D. The size of D depends on the depths of the original larger suffix trees, and it is usually small. For example, in our experiments, we have $d_1 = d_2 = 7$, and we only need to maintain a 7×7 matrix.

In the online phase, for any given patterns $P = (\alpha, \beta)$, we first greedy parse α and β and obtain (8.1). Then we search for those sub-patterns in the corresponding trees and find out their depths. We use $d(T, x)$ to denote the depth of the node corresponding to string x in tree T. If x does not have a corresponding node in T, we consider it as corresponding to a pruned node at depth 1 and set $d(T, x) = 1$.

We define the depth of α as the weighted average of the depths of its sub-patterns:

$$d(\alpha) = \sum_{i=1}^{m} w_i \times d\left(T_1, \alpha(i)\right). \tag{8.2}$$

Similarly, we define the depth of β as

$$d(\beta) = \sum_{j=1}^{n} w_j \times d\left(T_2, \beta(j)\right). \tag{8.3}$$

Once we have determined $d(\alpha)$ and $d(\beta)$, we look into the DST and return $D(d(\alpha), d(\beta))$ as our estimation of $e(P)$. Since $d(\alpha)$ and $d(\beta)$ may not be integers, we need to adjust them and use the following formula:

$$e(\alpha, \beta) = \frac{D\left(\lfloor d(\alpha) \rfloor, \lfloor d(\beta) \rfloor\right) + \dot{d}(\alpha) \times \dot{d}(\beta) \times \dot{D}(\alpha, \beta)}{M}$$

where

$$\dot{d}(\alpha) = d(\alpha) - \lfloor d(\alpha) \rfloor,$$

$$\dot{d}(\beta) = d(\beta) - \lfloor d(\beta) \rfloor, \text{and}$$

$$\dot{D}(\alpha, \beta) = D(\lceil d(\alpha) \rceil, \lceil d(\beta) \rceil) - D\left(\lfloor d(\alpha) \rfloor, \lfloor d(\beta) \rfloor\right).$$

Depending on the choice of the weights, we define three different depth-based strategies.

Strategy DE_1: Choose $w_i = 1/i$.

Strategy DE_2: Choose $w_i = 1/(2i-1)$.

Strategy DE_3: Choose $w_i = 1/2^i$.

In our experiments, we found that DE_1 and DE_2 are better than DE_3. An important observation is that the accuracy of the estimation is very sensitive to the choice of weight functions. So we should try to find good weight functions.

In the next subsection, we formulate the problem of finding a good weight function as a least square problem and we show that the latter problem can be solved by collecting necessary information in the offline phase.

8.6.2 A Least Square Problem

For simplicity, we discuss the case when the table F has only one column. (The generalization to two-column case is straightforward.) Suppose each row is a string of length at most l. Therefore, any row in the table can be greedily parsed into at most l sub-patterns against any suffix tree.

Suppose in the offline phase, we have constructed the complete suffix tree T_0. We can enumerate all substrings contained in T_0 in any particular order. Suppose there are a total of L $(1 \leq i \leq L)$ substrings and the depth of the ith substring s_i is denoted by y_i $(1 \leq i \leq L)$.

Suppose we prune T_0 to obtain a much smaller tree T (of size 100) and the depth of T is d. For any substring s_i, we greedily parse s_i against T and obtain

$$s_i = s_i(0) \ldots s_i(l-1),$$

and the depth of $s_i(j)$ in T is x_{ij}. We have

$$y_i = \sum_{j=0}^{l-1} w_j x_{ij} \quad \text{for } 1 \leq i \leq L. \tag{8.4}$$

Since we can construct both T_0 and T easily during the offline phase, we can consider all y_i's and x_{ij}'s as known numbers. We need to solve equation (8.4) for the to-be-determined weights $w_1, w_2, \ldots, w_{l-1}$.

Let $Y = (y_1, \ldots, y_L)^T \in R^{L \times 1}$, $X = (x_{ij}) \in R^{L \times l}$, and $w = (w_0, \ldots, w_{l-1})^T \in R^{l \times 1}$. We can rewrite equation (8.4) as

$$Y = Xw. \tag{8.5}$$

Our goal is to solve equation (8.5) for w.

Equation (8.5) is very unlikely to be solvable. Fortunately we do not need to solve it. We only need to solve the following least square problem:

$$\min \left\{ \sum_{i=1}^{L} \left(y_i - \sum_{j=0}^{l-1} x_{ij} w_j \right)^2 \right\} \tag{8.6}$$

In other words, we only need to solve equation (8.5) for an approximate solution w.

By multiplying X^T on both sides of (8.5), we obtain

$$X^T X w = X^T Y.$$

Denote $B = X^T X \in R^{l \times l}$, $Z = X^T Y \in R^{l \times 1}$. Since both B and Z are very small matrices, we only need to solve the following small set of equations:

$$Bw = Z. \tag{8.7}$$

To solve (8.7), we need to obtain B and Z. By examing the definition of B and Z, we can see that each entry in both B and Z can easily be accumulated during a second scanning of the table table F given that we have already constructed T_0 and T.

The second scan is needed for determining the weight w. We can conduct several experiments on different tables and try to find a good choice for the weight function w by scanning each table twice. Once we have such a good weight w available, we can use it in other applications when using the depth-based strategies.

8.7 Experiments

We conduct two groups of experiments. In the first group, we test our method using artificial data. In the second group, we test our method against real-life customer data. We also report experimental results for the original KVI method when real-life data are used. (All the experiments in [45, 46] are conducted on synthetic data.)

8.7.1 Experiments Using Artificial Data

We conduct our experiments according to the following guidelines.

Data Generation: Generate large database tables with two alphanumeric columns. Generate query patterns involving the two columns.

Offline Construction: Use the method of Section 8.4 to build catalogs of reasonable size.

Online Estimation: Use all the mismatch strategies of Section 8.5 to estimate the selectivity of query patterns.

Error Analysis: Compare the estimated selectivity with the exact selectivity.

Data Generation

Since no similar studies have been done before and no benchmark is available for our experiments, we generate our own experimental data. We need a database table that contains two correlated columns of alphanumeric type.

There is an emerging industry standard Transaction Processing Council (TPC) benchmark, known as the TPC-D benchmark [84], that involves the `LIKE` predicate. The TPC-D data is used in [45, 46] to conduct all the experiments for the KVI method. Unfortunately, all the alphanumeric type columns of the TPC-D data are generated independently.

We modify the data generation of the TPC-D data by introducing alphanumeric column correlations. Specifically, we generate our table F as follows. First, we generate the first column C_1 using the standard TPC-D method. We then generate the second column C_2 based on the content of the first column.

There are 92 base patterns specified in the benchmark. Each pattern is a color, e.g., "green". For the data generated, Each entry of C_1 is a obtained by choosing two

base colors randomly and concatenating these two colors separating the two colors by an underscore "_". Thus, each entry of C_1 is of the form $p_1\_p_2$ where both p_1 and p_2 are base colors.

Each element of the second column, C_2, is generated based on the content of C_1 in the same row. To generate C_2 elements, we partition the 92 base colors into six groups, g_1, g_2, g_3, g_4, g_5, and g_6. Each group g_i $(1 \leq i \leq 6)$ is partitioned into four equal size subgroups $g_{i,j}$ for $1 \leq j \leq 4$. The partition is shown in Table 8.2.

Group Name	*Group Size*	*Subgroup Size*
g_1	4	1
g_2	8	2
g_3	12	3
g_4	16	4
g_5	20	5
g_6	32	8

Table 8.2: Partition of the 92 base patterns.

Assume that at the ith row, the C_1 entry is $p_1\_p_2$. We generate the corresponding C_2 element $q_1\_q_2$ as follow: First we determine which subgroups contain p_1 and p_2. Suppose p_1 is in subgroup g_{i_1,j_1} and p_2 is in subgroup g_{i_2,j_2}. With probability λ, we randomly choose a color in g_{i_1,j_1} and a color in g_{i_2,j_2} as q_1 and q_2, respectively. With probability $1-\lambda$, we randomly and independently choose two colors from the 92 base colors as q_1 and q_2 respectively. (In our experiments, we use $\lambda = 80\%$.)

The smaller the subgroup g_{i_1,j_1} is, the stronger the correlation between p_1 and q_1 will be. The same is true for p_2 and q_2. Thus, the correlation between the two columns depends on the partition of the base colors as well as λ.

In our experiments, we have generated a table F of 200K rows, i.e., M =200K.

Our query pattern is of the form (p, q), where p and q are base colors. In our experiments, we have generated 476 queries .

Offline Construction

In the off-line phase, we process F section by section, with each section containing 1000 rows. When finishing processing a section, we obtain two trees of size 1000 (nodes) and a 1000×1000 matrix R. But during the processing of one section, the two trees may grows significantly. In the experiments, those intermediate trees typically contain about 7800 nodes, which is still tolerable as far as the storage space is concerned.

After the whole table has been processed, we further prune the two trees and obtain T_1 and T_2, where T_1 contains 96 nodes and T_2 contains 100 nodes. Correspondingly, we reorganize R and obtain a 96×100 matrix R. T_1, T_2 and R constitute the catalog. Our construction implicitly used an edge weight $c_E = 12675$, i.e., a node remains in one of the trees if one of its adjacent edge has weight ≥ 12675. The node weight for T_1 is 12676, and the node weight for T_2 is 12737.

To use the depth-based methods, we need to construct a different catalog that (in our experiment) consists of two suffix trees T_1 and T_2, plus a small 7×7 DST table. The two trees are the same as before. The DST table stores depth-based common count information. The size 7 comes from the fact that the trees before the final pruning both have depth 6. Note that if we build a suffix tree for column C_1 or C_2 alone, the tree has a depth of 10.

Online Estimation

Although the final catalog contains two suffix trees, no query can be matched exactly. It is natural since the two trees can only record very frequent substrings becasue of their small size. So we must use the mismatch strategies to estimate selectivity.

We test all the mismatch strategies. The performance of those strategies is reported in the next subsection.

Error Analysis

We measure the performance of a mismatch strategy by computing its relative error. That is, for a given pattern P, if the exact selectivity is $s(P)$, the estimated value (based on one mismatch strategy) is $e(P)$, we compute the relative error $(e(P) - s(P))/s(P)$. We allow negative relative errors in our discussion, but it can never be less than -100%. The exact value of the selectivity for any given query can be obtained by doing a exhaust search of table F.

We plot the cumulative number of patterns along the y axis against the relative error on the x axis: a point (x, y) on the graph means that y patterns have relative errors are les or equal to $x\%$.

In the experiments both strategy I_1 and strategy I_3 give zero as the value of $e(P)$ for almost all query patterns. It is not surprising because both strategies estimate $s(P)$ as the product of the selectivity of many subpatterns of P. Strategy I_2 results in big errors since it sets the selectivity for any mismatch subpattern to c_E/M. (The threshold value c_E is usually big.) These three strategies are thus not recommended and their results are not shown.

The performance of other strategies for the 476 query patterns is shown in Figure 8.1–Figure 8.3. As we can see from the figures, $I_4, I_5, CE_1, CE_3, DE_1$ and DE_4 are the best strategies and we plot them in Figure 8.4.

8.7.2 Experiments Using Real Data

Our experiments in this group are all based on real customer data and query.

Description of data and query

The original database table T contains N rows ($N \simeq 1$ million) and 36 columns. One of the columns, column 34, is named as SPECL_INSTR_TEXT. This column is

of the varchar type (variable length character strings), with a maximum length 80 (characters). For example, the following sentences occur in this column:

- DONT JUMP FENCE
- SMALL DOG NO PITT/ SCE LOCK ON GATE
- ENT LEFT DOG ON RGT
- DON'T ENTER UNLESS OWNER IS HOME.

For this particular column, users (home meter readers) were interested in asking queries of the following form (the *DOG query*):

```
SELECT *
FROM   T
WHERE  SPELCL_INSTR_TEXT LIKE '%BAD%'
OR     SPELCL_INSTR_TEXT LIKE '%BEWARE%'
OR     SPELCL_INSTR_TEXT LIKE '%DANGEROUS%'
OR     SPELCL_INSTR_TEXT LIKE '%DOBIE%'
OR     SPELCL_INSTR_TEXT LIKE '%DOBY%'
OR     SPELCL_INSTR_TEXT LIKE '%DOG%'
OR     SPELCL_INSTR_TEXT LIKE '%MEAN%'
OR     SPELCL_INSTR_TEXT LIKE '%PIT%'
OR     SPELCL_INSTR_TEXT LIKE '%VICIOUS%'
OR     SPELCL_INSTR_TEXT LIKE '%DG%'
```

We single out this particular column from the table and form a signgle-column table of N rows. We find that about 4/5 of the rows are empty. For convenience, we delete all the empty rows to obtain a table F based on which we will conduct our experiments. Incidently, the size of F is 200K, the exact same as that of the table

used in the experiemnts of [45, 46]. The original experiments reproted in [45, 46] are based on the TPC-D benchmark data that are generated artificially. Our data are chosen from real-life data and have a very complicated distribution. There are ten patterns in the DOG query and we list the exact selectivity for each single pattern in the Table 8.3. We can see that the selectivity values vary from 0.11072 to 0.000185.

Pattern Label	*Pattern*	*Exact Count*	*Exact Selectivity*
1	BAD	4263	0.021315
2	BEWARE	183	0.000915
3	DANGEROUS	37	0.000185
4	DOBIE	1768	0.00884
5	DOBY	182	0.00091
6	DOG	22144	0.11072
7	MEAN	882	0.00441
8	PIT	5125	0.025625
9	VICIOUS	151	0.000755
10	DG	17235	0.086175

Table 8.3: Exact selectivity of the 10 patterns in the DOG query.

Using KVI method on DOG query

We use KVI method to estimate the selectivties of the 10 patterns that appear in the DOG query.

We conduct our experiments using three catlogs of different sizes. Each catalog contains x nodes where $x \in \{100, 500, 3577\}$. For each catalog, we use all mismathch strategies of KVI method to estimate the selectivity for each pattern. The overall performance of a strategy is evaluated according to the standard variance that is described below.

Suppose we are interested in the patterns in P as a whole, where P is a set of n patterns. Let

$$e(P) = (e(p_1), \ldots, e(p_n)) .$$

$$s(P) = (s(p_1), \ldots, s(p_n)) .$$

We use the standard variance (the Euclidean distance) to measure the error of $e(P)$ with respect to $s(P)$:

$$var(e(P), s(P)) = \left[\sum_{i=1}^{n}(e(p_i) - s(p_i))^2\right]^{1/2} .$$

We rank the 11 strategies in [46] according to their overall errors on the 10 patterns. We observe that for all three catalogs, strategies I_1, CE_3 and DE_1 are the best strategies for the DOG query patterns. We plot the relative estimation errors of these three strategies for each pattern in Figure 8.5– Figure 8.7.

In [46], it is suggested that strategies CE_2, CE_3, DE_1 and I_1 are the best strategies for positive single patterns of TPC-D data and strategy CE_2 is particular good. Our experiments with real data conclude with a slightly different best strategy set. Their intersection, $\{I_1, CE_3, DE_1\}$, gives us a candidate set of good strategies.

We also find that different from TPC-D data, to obtain reasonable accurate estimations for the 10 single patterns in DOG query, we must use a much bigger catalog (e.g. 500 nodes).

Using our extended method on DOG query

Although DOG query only involves one column in the table, the multiple patterns in the query make it very interesting to apply our two-column method . By assuming that the two columns in the table are exact the same, we can apply our method directly to estimate the selectivity for the one-column two-pattern query:

$$C \texttt{ LIKE } *p_1* \texttt{ AND } C \texttt{ LIKE } *p_2*$$

One interesting thing to notice is that for this type of queries, the two trees built during the offline stage are exactly the same and the final matrix R is symmetric. So we just need to keep one tree and half of the entries in R in the catalog.

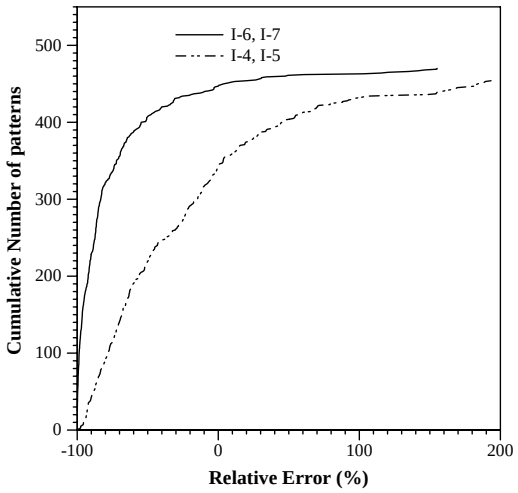


Figure 8.1: Performance of the independence-based strategies for 476 patterns.

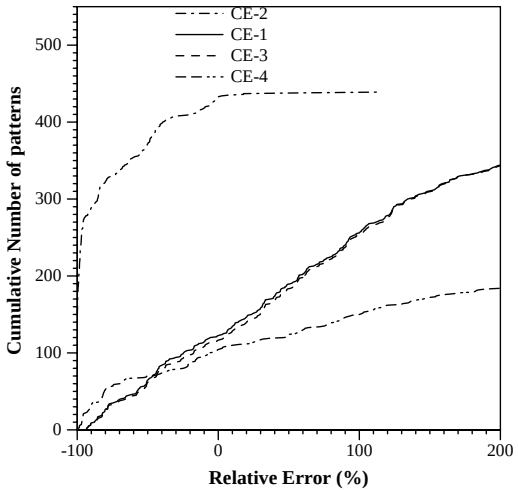


Figure 8.2: Performance of the child-based strategies for 476 patterns.

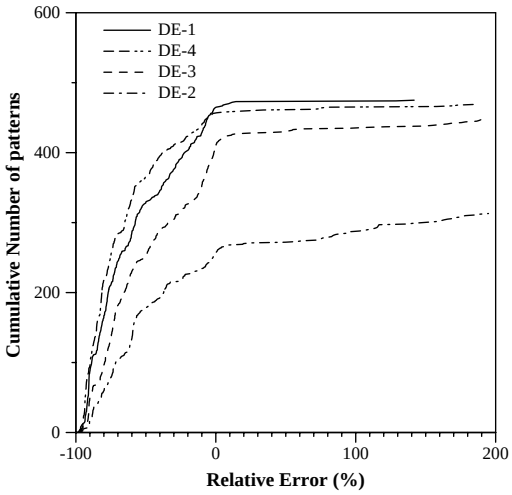


Figure 8.3: Performance of the depth-based strategies for 476 patterns.

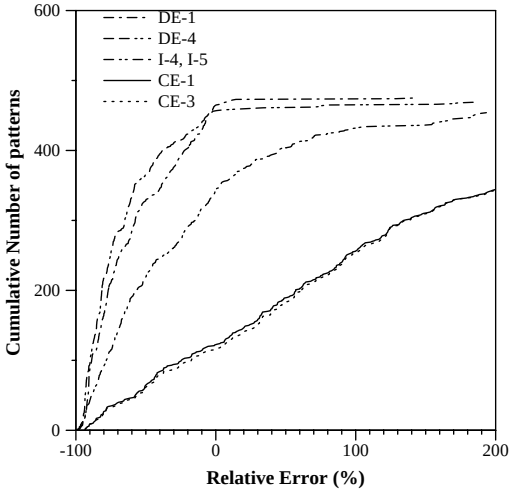


Figure 8.4: Graph of the best performing strategies for 476 patterns.

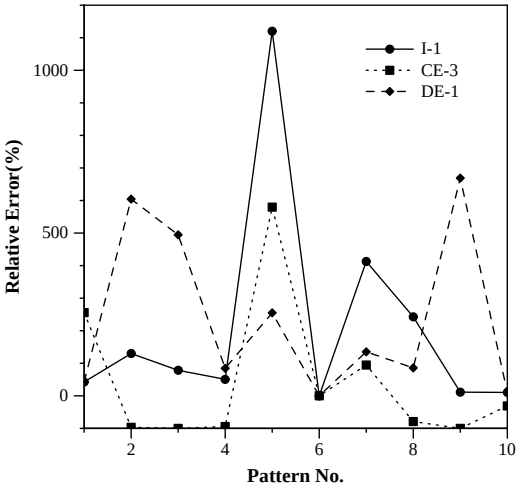


Figure 8.5: Performance of the best KVI strategies for the 10 patterns in the DOG query with a catalog of 100 nodes.

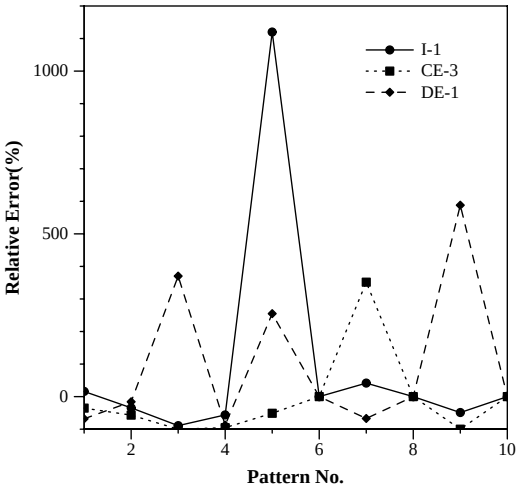


Figure 8.6: Performance of the best KVI strategies for the 10 patterns in the DOG query with a catalog of 500 nodes.

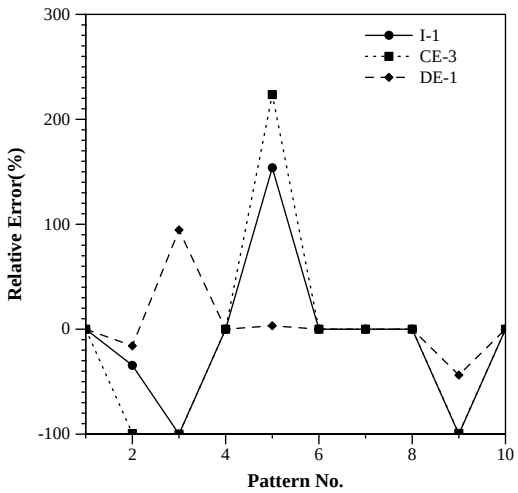


Figure 8.7: Performance of the best KVI strategies for the 10 patterns in the DOG query with a catalog of 3577 nodes.

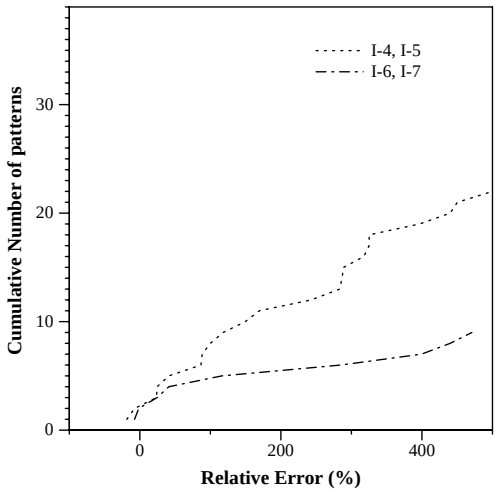


Figure 8.8: Performance of the independence-based strategies for 39 DOG query pairs.

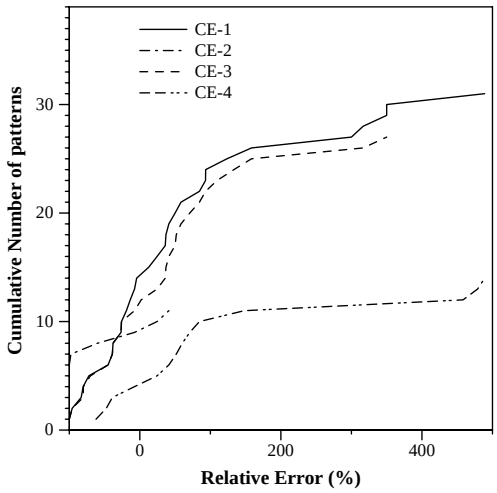


Figure 8.9: Performance of the child-based strategies for 39 DOG query pairs.

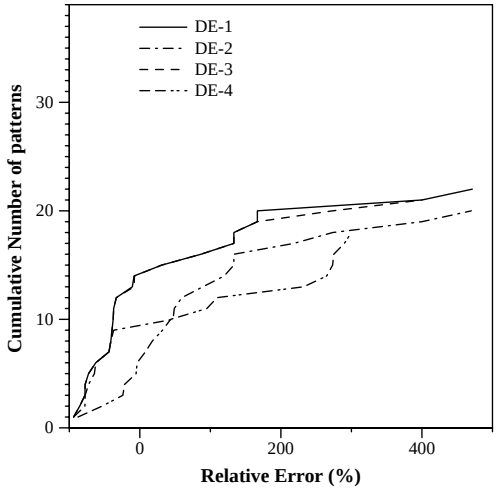


Figure 8.10: Performance of the depth-based strategies for 39 DOG query pairs.

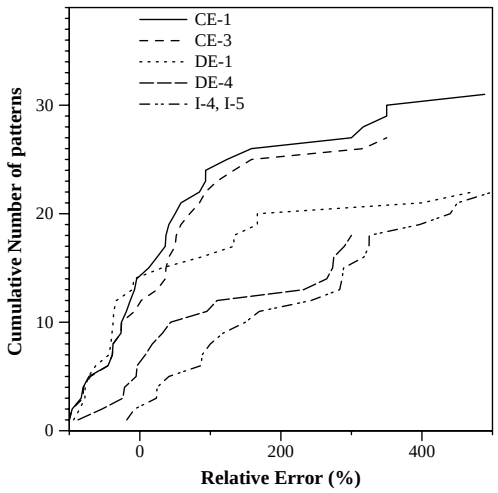


Figure 8.11: Graph of the best performing strategies for 39 DOG query pairs.

In our experiments, we generate all possible pairs of the 10 single patterns. There are $10 * 9/2 = 45$ of them. Among the 45 pairs, six of them have selectivity of exact 0. (The pattern pairs never appear simultaneously in the same row.) So we only consider the remaining 39 pairs. Their exact selectivities vary from 0.000005 to 0.010995.

We notice that, as the KVI method for the real data, to obtain a reasonable accurate estimation for the DOG query pattern pairs, we need to keep a much larger tree and matrix in the catalog. Figure 8.8–Figure 8.10 plot the performance of different strategies for a catalog tree of 1000 nodes. We observe that strategies I_4, I_5, CE_1, CE_3, DE_1 and DE_4 are the best performance strategies for real data, and we plot their performance in Figure 8.11.

We point out that DE_1 and DE_4 are very attractive because the size of the catalog used for epth-based estimation strategies is much smaller than that used for other strategies.

By combining the results of single-column estimation and two-column estimation, We can estimate the selectivities of the following types of predicates:

- C `LIKE` $*p_1*$ `OR` C `LIKE` $*p_2*$

- C_1 `LIKE` $*p_1*$ `OR` C_2 `LIKE` $*p_2*$
- C `LIKE` $*p_1*$ `AND` C `LIKE` $*p_2*$
- C_1 `LIKE` $*p_1*$ `AND` C_2 `LIKE` $*p_2*$

8.8 Conclusions

One important conclusion from our experiments is that estimation of alphanumeric selectivity for real data is considerably more difficult than that for artificial data. The data distribution can be complicated and larger space allocation in the catalog is usually required. Thus there is an opportunity for applying appropriate data compression techniques.

A very interesting problem is the selectivity estimation for queries involving correlated columns of mixed data types. For example, we may consider two columns, one is of alphanumeric type and the other is of numeric type.

Finally, we may consider using other data structures to construct catalog for alphanumeric data. Although the suffix tree is a linear space data structure for representing character strings, it is an irregular tree and the hidden constant is usully big. It would be very interesting to investigate alternative data structures to represent alphanumeric data.

Chapter 9

Scalable Learning Techniques for Data Mining Applications

Database technology has been used with great success in traditional data processing. There is an increasing desire to use this technology in new application domains. One application domain that is acquiring considerable significance in recent years is *data mining*. Data mining is the process of extracting valid, previously unknown, and ultimately comprehensible information from large databases and using it to make crucial business decisions. The extracted information can be used to form a prediction or classification model, or to identify relations between database records.

Classification is an important data mining problem [4, 5] and can be described as follows: We are given a *training set* (or *DETAIL* table) consisting of many training examples. Each training example is a row with multiple attributes, one of which is a *class label*. The objective of classification is to process the *DETAIL* table and produce a *classifier*, which contains a description (model) for each class. The model will be used to classify future *test data* for which the class labels are unknown (see [12, 69, 59, 14]). It can also be used to develop a better understanding of each class in the data. Applications of classification include credit approval, target marketing, medical diagnosis, treatment effectiveness, store location, etc..

A simple training set is shown in Table 9.1. The examples reflect the past experience of an organization extending credit. From those examples, we can generate the classifier shown in Figure 9.1.

salary	*age*	*credit rating*
65K	30	Safe
15K	23	Risky
75K	40	Safe
15K	28	Risky
100K	55	Safe
60K	45	Safe
62K	30	Risky

Table 9.1: Training set.

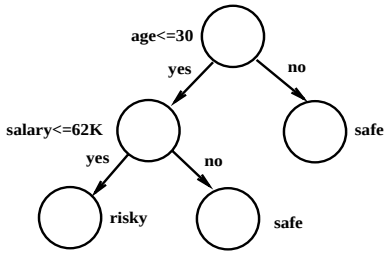


Figure 9.1: Decision tree for the data in Table 9.1

9.1 Previous Work

Classification has been studied extensively [99] and several classification models have been proposed over the years, e.g., neural networks [48], statistical models like linear/quadratic discriminants [42], decision trees [12, 69, 14], and genetic models [29]. Among these models, decision tree are particularly suited for data mining applications for the following reasons [77]:

- Decision tree can be constructed relatively fast compared to other methods.
- Decision tree models are simple and easy to understand [69].
- Decision trees can be easily converted into SQL statements that can be used to access database efficiently [4].
- Decision tree classifiers obtain similar and sometimes better accuracy when compared with other classification methods [57].

Most research work on classification in the context of data mining applications has therefore focused on decision tree model. However, the existing classification algorithms have two problems.

The first problem is *scalability*. Most previous algorithms have the restriction that the training data should fit in memory [55]. Although memory and CPU prices are plunging, the volume of data available for analysis is immense and getting larger. We may not assume that the data are memory-resident. Hence, an important research problem is to develop accurate classification algorithms that are scalable with respect to I/O and parallelism. Accuracy is known to be domain-specific (e.g., insurance fraud, target marketing). However, the problem of scalability for large amounts of data is more amenable to a general solution. A classification algorithm should scale well; that is, the classification algorithm should work well even if the training set is huge and vastly overflows internal memory. In data mining applications, it is common to have training sets with several million examples. However, even the state-of-art classifiers (SLIQ [55] and SPRINT [77]) need an in-memory data structure of size $O(N)$, where N is the size of the training data, to achieve efficiency. The best previously known classifier (SPRINT) does a quadratic number of I/Os for large N and thus it is not scalable at all.

Random sampling is often an effective technique in dealing with large data sets. For simple applications whose inherent structures are not very complex, this approach is efficient and gives good results. However, in our case, we do not favor random sampling for two main reasons:

1. In general, choosing the proper sample size is still an open question. The following factors must be taken into account:
    - The training set size.
    - The convergence of the algorithm. Usually, many iterations are needed

to process the sampling data and refine the solution. It's very difficult to estimate how fast the algorithm will give a satisfactory solution.

- The complexity of the model.

The best known theoretical upper bounds on sample size suggest that the training set size may need to be immense to assure good accuracy [21, 44].

2. In many real applications, customers insist that *all* data, not just a sample of the data, must be processed. Since the data are usually obtained from valuable resources at considerable expense, they should be used as a whole throughout the analysis.

Therefore, designing a scalable classifier may be necessary or preferable, although we can always use random sampling in places where it is appropriate.

The second problem is that *all existing classification methods need to extract data from the database to file before applying the classification algorithms.* Since extracting data to files before running data mining functions would require extra I/O costs, users as well as previous investigators [40, 39] have pointed to the need for the relational database managers to have these functions built in. Besides reducing I/O costs, this approach leverages over 20 years of research and development in DBMS technology, among them are:

- scalability,
- memory hierarchy management [72, 74],
- parallelism [9],
- optimization of the executions [11],
- platform independence, and
- client server API [60].

9.2 Our Approach

In this chapter, we propose a novel classification algorithm (classifier) called MIND (MINing in Databases).

MIND is truly scalable with respect to I/O efficiency, which is important since scalability is a key requirement for any data mining algorithm. We analyze and compare the I/O complexities of MIND and SPRINT. Our theoretical analysis and experimental results show that MIND scales well whereas SPRINT can exhibit quadratic I/O times.

MIND can be phrased in such a way that its implementation is very easy using the extended relational calculus SQL, and this in turn allows the classifier to be built into a relational database system directly. We have implemented MIND as a stored procedure, a common feature in modern DBMSs. In addition, since most modern database servers have very strong parallel query processing capabilities, MIND runs in parallel at no extra cost.

We have built a prototype of MIND in IBM's DB2 and have benchmarked its performance. We describe the working prototype and report the measured performance with respect to SPRINT. MIND scales not only with the size of datasets but also with the number of processors on an IBM SP2 computer system. Even on uniprocessors, MIND scales well beyond dataset sizes previously published for classifiers. We also give some insights that may have an impact on the evolution of the extended relational calculus SQL.

This chapter is organized as follows: We describe our MIND algorithm in Section 9.3; its database implementation is described in Section 9.4. An illustrative example is given in Section 9.5. A theoretical performance analysis is given in Section 9.6. We revisit MIND algorithm in Section 9.7 using a general extension of current SQL standards. In Section 9.8, we present our experimental results. We

make concluding remarks in Section 9.9.

9.3 The Algorithm

9.3.1 Overview

A decision tree classifier is built in two phases: a growth phase and a pruning phase. In the growth phase, the tree is built by recursively partitioning the data until each partition is either "pure" (all members belong to the same class) or sufficiently small (according to a parameter set by the user). The form of the split used to partition the data depends upon the type of the attribute used in the split. Splits for a numerical attribute A are of the form $value(A) \leq x$, where x is a value in the domain of A. Splits for a categorical attribute A are of the form $value(A) \in S$, where S is a subset of $domain(A)$. We consider only binary splits as in [55, 77] for purpose of comparisons. After the tree has been fully grown, it is pruned to remove noise in order to obtain the final tree classifier.

The tree growth phase is computationally much more expensive than the subsequent pruning phase. The tree growth phase accesses the training set (or *DETAIL* table) multiple times, whereas the pruning phase only needs to access the fully grown decision tree. We therefore focus on the tree growth phase. The following pseudocode gives an overview of our algorithm:

GrowTree(TrainingSet *DETAIL*)

Initialize tree T and put all records of *DETAIL* in the root;

while (some leaf in T is not a *STOP* node)

for each attribute i do

form the dimension table (or histogram) DIM_i;

evaluate *gini* index for each non-*STOP* leaf at each split value

with respect to attribute i;

for each non-*STOP* leaf do

get the overall best split for it;

partition the records and grow the tree for one more level

according to the best splits;

mark all small or pure leaves as *STOP* nodes;

return T;

9.3.2 Leaf node list data structure

A powerful method called SLIQ was proposed in [55] as a semi-scalable classification algorithm. The key data structure used in SLIQ is a *class list* whose size is linear in the number of examples in the training set. The fact that the *class list* must be memory-resident puts a hard limitation on the size of the training set that SLIQ can handle.

In the improved SPRINT classification algorithm [77], new data structures *attribute list* and *histogram* are proposed. Although it is not necessary for the attribute list data structure to be memory-resident, the histogram data structure must be in memory to insure good performance. To perform the split in [77], a *hash table* whose size is linear in the number of examples of the training set is used. When the hash table is too large to fit in memory, splitting is done in multiple steps, and SPRINT

does not scale well.

In our MIND method, the information we need to evaluate the split and perform the partition is stored in relations in a database. Thus we can take advantage of DBMS functionalities and memory management. The only thing we need to do is to incorporate a data structure that relates the database relations to the growing classification tree. We assign a unique number to each node in the tree. When loading the training data into the database, imagine the addition of a hypothetical column *leaf_num* to each row. For each training example, *leaf_num* will always indicate which leaf node in the current tree it belongs to. When the tree grows, the *leaf_num* value changes to indicate that the record is moved to a new node by applying a split. A static array called *LNL (*leaf node list) is used to relate the *leaf_num* value in the relation to the number assigned to the corresponding node in the tree. By using a labeling technique, we insure that at each tree growing stage, the nodes always have the identification numbers 0 through $N-1$, where N is the number of nodes in the tree. $LNL[i]$ is a pointer to the node with identification number i. For any record in the relation, we can get the leaf node it belongs to from its *leaf_num* value and *LNL* and hence we can get the information in the node (e.g. split attribute and value, number of examples belonging to this node and their class distribution).

To insure the performance of our algorithm, *LNL* is the only data structure that needs to be memory-resident. The size of *LNL* is equal to the number of nodes in the tree, so *LNL* can always be stored in memory.

9.3.3 Computing the *gini* index

A splitting index is used to choose from alternative splits for each node. Several splitting indices have recently been proposed. We use the *gini* index, originally proposed in [12] and used in [55, 77], because it gives acceptable accuracy. The

accuracy of our classifier is therefore the same as those in [55, 77].

For a data set S containing N examples from C classes, $gini(S)$ is defined as

$$gini(S) = 1 - \sum_{i=1}^{C} p_i^2 \tag{9.1}$$

where p_i is the relative frequency of class i in S. If a split divides S into two subset S_1 and S_2, with sizes N_1 and N_2 respectively, the *gini* index of the divided data $gini_{\text{split}}(S)$ is given by

$$gini_{\text{split}}(S) = \frac{N_1}{N} gini(S_1) + \frac{N_2}{N} gini(S_2) \tag{9.2}$$

The attribute containing the split point achieving the smallest *gini* index value is then chosen to split the node [12]. Computing the *gini* index is the most expensive part of the algorithm since finding the best split for a node requires evaluating the *gini* index value for each attribute at each possible split point.

The training examples are stored in a relational database system using a table with the following schema: $DETAIL(attr_1, attr_2, \ldots, attr_n, class, leaf\_num)$, where $attr_i$ is the ith attribute, for $1 \leq i \leq n$, *class* is the classifying attribute, and *leaf_num* denotes which leaf in the classification tree the record belongs to.[1] In actuality *leaf_num* can be computed from the rest of the attributes in the record and does not need to be stored explicitly. As the tree grows, the *leaf_num* value of each record in the training set keeps changing. Because *leaf_num* is a computed attribute, the *DETAIL* table is never updated, a key reason why MIND is efficient for the DB2 relational database. We denote the cardinality of the class label set by C, the number of the examples in the training set by N, and the number of attributes (not including class label) by n.

[1] *DETAIL* itself may be a summary of atomic transactional data, or the atomic data.

9.4 Database Implementation of MIND

To emphasize how easily MIND is embeddable in a conventional database system using SQL and its accompanying optimizations, we describe our MIND components using SQL.

9.4.1 Numerical attributes

For every level of the tree and for each attribute $attr_i$, we recreate the dimension table (or histogram) called DIM_i with the schema $DIM_i(leaf\_num,\ class,\ attr_i,\ count)$ using a simple SQL `SELECT` statement on *DETAIL*:

`INSERT INTO` DIM_i [2]

`SELECT` $leaf\_num, class, attr_i$, `COUNT(*)`

`FROM` *DETAIL* [3]

`WHERE` $leaf\_num <> STOP$

`GROUP BY` $leaf\_num, class, attr_i$

Although the number of distinct records in *DETAIL* can be huge, the maximum number of rows in DIM_i is typically much less and is no greater than (#leaves in tree) $\times$ (#distinct values on $attr_i$) $\times$ (#distinct classes), which is very likely to be of the order of several hundreds [56]. By including $leaf\_num$ in the attribute list for grouping, MIND collects summaries for every leaf in one query. In the case that the number of distinct values of $attr_i$ is very large, preprocessing is often done in practice to further discretize it [1, 51]. Discretization of variable values into a smaller number of classes is sometimes referred to as "encoding" in data mining practice [1].

[2] Note the structural transformation that takes an attribute name in a schema and turns it into the table name of an aggregate.

[3] *DETAIL* may refer to data either in a relation or a file (e.g. on tape). In the case of a file, *DETAIL* resolves to an execution of a user-defined function (e.g. fread in UNIX) [16]).

Roughly speaking, this is done to obtain a measure of aggregate behavior that may be detectable [56]. Alternatively, efficient external memory techniques can be used to form the dimension tables in a small number (typically one or two) linear passes, at the possible cost of some added complexity in the application program to give the proper hints to the DBMS, as suggested in Section 9.6.

After populating DIM_i, we evaluate the *gini* index value for each leaf node at each possible split value of the attribute i by performing a series of SQL operations that only involve accessing DIM_i.

It is apparent for each attribute i that its DIM_i table may be created in one pass over the *DETAIL* table. It is straightforward to schedule one query per dimension (attribute). Completion time is still linear in the number of dimensions. Commercial DBMSs store data in row-major sequence. I/O efficiencies may be obtained if it is possible to create dimension tables for all attributes in one pass over the *DETAIL* table. Concurrent scheduling of the queries populating the DIM_i tables is the simple approach. Existing buffer management schemes that rely on I/O latency appear to synchronize access to *DETAIL* for the different attributes. The idea is that one query piggy-backs onto another query's I/O data stream. Results from early experiments are encouraging [79].

It is also possible for SQL to be extended to insure that, in addition to optimizing I/O, CPU processing is also optimized. Taking liberty with SQL standards, we write the following query as a proposed SQL operator:

```
SELECT FROM DETAIL
INSERT INTO DIM_1{leaf_num, class, attr_1, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_1}
INSERT INTO DIM_2{leaf_num, class, attr_2, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_2}
   ⋮
INSERT INTO DIM_n{leaf_num, class, attr_n, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_n}
```

The new operator forms multiple groupings concurrently and may allow further RDBMS query optimization.

Since such an operator is not supported, we make use of the object extensions in DB2, the *user-defined function* (udf) [81, 15, 38]. User-defined functions are used for association in [6]. User-defined function is a new feature provided by DB2 version 2 [15, 38]. In DB2 version 2, the functions available for use in SQL statements extend from the system built-in functions, such as `avg, min, max, sum`, to more general categories, such as user-defined functions (udf). An external udf is a function that is written by a user in a host programming language. The `CREATE FUNCTION` statement for an external function tells the system where to find the code that implements the function. In MIND we use a udf to accumulate the dimension tables for all attributes in one pass over *DETAIL*.

For each leaf in the tree, possible split values for attribute i are all distinct values of $attr_i$ among the records that belong to this leaf. For each possible split value, we need to get the class distribution for the two parts partitioned by this value in order

to compute the corresponding *gini* index. We collect such distribution information in two relations, UP and $DOWN$.

Relation UP with the schema $UP(leaf\_num,\ attr_i,\ class,\ count)$ can be generated by performing a self-outer-join on DIM_i:

```
INSERT INTO UP
SELECT d1.leaf_num, d1.attr_i, d1.class, SUM(d2.count)
FROM (FULL OUTER JOIN DIM_i d1, DIM_i d2
ON d1.leaf_num = d2.leaf_num AND d2.attr_i ≤ d1.attr_i AND d1.class = d2.class
GROUP BY d1.leaf_num, d1.attr_i, d1.class)
```

Similarly, relation $DOWN$ can be generated by just changing the $\leq$ to $>$ in the `ON` clause. We can also obtain $DOWN$ by using the information in the leaf node and the *count* column in UP without doing a join on DIM_i again.

$DOWN$ and UP contain all the information we need to compute the *gini* index at each possible split value for each current leaf, but we need to rearrange them in some way before the *gini* index is calculated. The following intermediate view can be formed for all possible classes k: The following intermediate view can be formed for all possible classes k:

```
CREATE VIEW C_k_UP(leaf_num, attr_i, count) AS
SELECT leaf_num, attr_i, count
FROM UP
WHERE class = k
```

Similarly, we define view C_k_DOWN from $DOWN$.

A view $GINI\_VALUE$ that contains all *gini* index values at each possible split point can now be generated. Taking liberty with SQL syntax, we write

```
CREATE VIEW GINI_VALUE(leaf_num, attr_i, gini) AS
SELECT  u_1.leaf_num, u_1.attr_i, f_gini
FROM    C_1_UP u_1, ...,C_C_UP u_C, C_1_DOWN d_1, ...,C_C_DOWN d_C
WHERE   u_1.attr_i = ··· = u_C.attr_i = d_1.attr_i = ··· = d_C.attr_i
AND     u_1.leaf_num = ··· = u_C.leaf_num = d_1.leaf_num = ··· = d_C.leaf_num
```

where f_{gini} is a function of $u_1.count$, ..., $u_n.count$, $d_1.count$, ..., $d_n.count$ according to (9.1) and (9.2).

We then create a table *MIN_GINI* with schema *MIN_GINI*(*leaf_num*, *attr_name*, *attr_value*, *gini*):

```
INSERT INTO MIN_GINI
SELECT  leaf_num, :i[4], attr_i[5], gini
FROM GINI_VALUE a
WHERE a.gini=(SELECT MIN(gini)
              FROM GINI_VALUE b
              WHERE a.leaf_num = b.leaf_num)
```

Table *MIN_GINI* now contains the best split value and the corresponding *gini* index value for each leaf node of the tree with respect to $attr_i$. The table formation query has a nested subquery in it. The performance and optimization of such queries are studied in [11, 59, 25].

We repeat the above procedure for all other attributes. At the end, the best split value for each leaf node with respect to all attributes will be collected in table *MIN_GINI*, and the overall best split for each leaf is obtained from executing the following:

[4] i is a host variable, the value applies on invocation of the statement.

[5] Again, note the transformation for the table name DIM_i to column value i and column name $attr_i$.

```
CREATE VIEW BEST_SPLIT(leaf_num, attr_name, attr_value) AS
SELECT leaf_num, attr_name, attr_value
FROM MIN_GINI a
WHERE a.gini=(SELECT MIN(gini)
              FROM MIN_GINI b
              WHERE a.leaf_num = b.leaf_num)
```

9.4.2 Categorical attributes

For categorical attribute i, we form DIM_i in the same way as for numerical attributes. Relation DIM_i contains all the information we need to compute the *gini* index for any subset splitting. In fact, it is an analog of the *count matrix* in [77], but formed with set-oriented operators.

A possible split is any subset of the set that contains all the distinct attribute values. If the cardinality of attribute i is m, we need to evaluate the splits for all the 2^m subsets. Those subsets and their related counts can be generated in a recursive way. The schema of the relation that contains all the k-sets is $S_k\_IN(leaf\_num,\ class,\ v_1, v_2, ..., v_k,\ count)$. Obviously we have $DIM_i = S_1\_IN$. S_k_IN is then generated from S_1_IN and S_{k-1}_IN as follows:

```
INSERT INTO S_k_IN
SELECT p.leaf_num, p.class, p.v_1, ..., p.v_{k-1}, q.v_1, p.count + q.count
FROM (FULL OUTER JOIN S_{k-1}_IN p, S_1_IN q
ON p.leaf_num = q.leaf_num AND p.class = q.class AND q.v_1 > p.v_{k-1})
```

We generate relation S_k_OUT from S_k_IN in a manner similar to how we generate $DOWN$ from UP. Then we treat S_k_IN and S_k_OUT exactly as $DOWN$ and UP for numerical attributes in order to compute the *gini* index for each k-set split.

A simple observation is that we don't need to evaluate all the subsets. We only need to compute the k-sets for $k = 1, 2, \ldots, \lfloor m/2 \rfloor$ and thus save time. For large m, greedy heuristics are often used to restrict search.

9.4.3 Partitioning

Once the best split attribute and value have been found for a leaf, the leaf is split into two children. If *leaf_num* is stored explicitly as an attribute in *DETAIL*, then the following `UPDATE` performs the split for each leaf:

`UPDATE` *DETAIL*

`SET` $leaf\_num = Partition(attr_1, \ldots, attr_n, class, leaf\_num)$

The user-defined function *Partition* is as follows:

Partition(record r)

Use the *leaf_num* value of r to locate the tree node n that r belongs to;

Get the best split from node n;

Apply the split to r, grow a new child of n if necessary;

Return a new *leaf_num* according the result of the split;

However, *leaf_num* is not a stored attribute in *DETAIL* because updating the whole relation *DETAIL* is expensive. We observe that *Partition* is merely applying the current tree to the original training set. We avoid the update by replacing *leaf_num* by function *Partition* in the statement forming DIM_i. If *DETAIL* is stored on non-updatable tapes, this solution is required. It is important to note that once the dimension tables are created, the *gini* index computation for all leaves involves only dimension tables.

$attr_1$	$attr_2$	$class$	$leaf\_num$
65K	30	Safe	0
15K	23	Risky	0
75K	40	Safe	0
15K	28	Risky	0
100K	55	Safe	0
60K	45	Safe	0
62K	30	Risky	0

Table 9.2: Initial relation *DETAIL* with implicit *leaf_num*

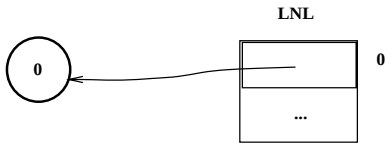


Figure 9.2: Initial tree

$leaf\_num$	$attr_1$	$class$	$count$
0	15	2	2
0	60	1	1
0	62	2	1
0	65	1	1
0	75	1	1
0	100	1	1

Table 9.3: Relation DIM_1

9.5 An Example

We illustrate our algorithm by an example. The example training set is the same as the data in Table 9.1.

Phase 0: Load the training set and initialize the tree and *LNL*. At this stage, relation *DETAIL*, the tree, and *LNL* are shown in Table 9.2 and Figure 9.2.

Phase 1: Form the dimension tables for all attributes in one pass over *DETAIL* using user-defined function. The result dimension tables are show in Table 9.3–9.4.

$leaf\_num$	$attr_2$	$class$	$count$
0	23	2	1
0	28	2	1
0	30	1	1
0	30	2	1
0	40	1	1
0	45	1	1
0	55	1	1

Table 9.4: Relation DIM_2

$leaf\_num$	$attr_1$	$class$	$count$
0	15	1	0
0	15	2	2
0	60	1	1
0	60	2	2
0	62	1	1
0	62	2	3
0	65	1	2
0	65	2	3
0	75	1	3
0	75	2	3
0	100	1	4
0	100	2	3

Table 9.5: Relation UP

Phase 2: Find the best splits for current leaf nodes. A best split is found through a set of operations on relations as described in Section 9.3.

First we evaluate the *gini* index value for $attr_1$. The procedure is depicted in Table 9.5–9.13.

We can see that the best splits on the two attributes achieve the same *gini* index value, so relation *BEST_SPLIT* is the same as *MIN_GINI* except that it does not contain the column *gini*. We store the best split in each leaf node of the tree (the root node in this phase). In case of a tie for best split at a node, any one of them ($attr_2$ in our example) can be chosen.

$leaf\_num$	$attr_1$	$class$	$count$
0	15	1	4
0	15	2	1
0	60	1	3
0	60	2	1
0	62	1	3
0	62	2	0
0	65	1	2
0	65	2	0
0	75	1	1
0	75	2	0

Table 9.6: Relation *DOWN*

$leaf\_num$	$attr_1$	$count$
0	15	0.0
0	60	1.0
0	62	1.0
0	65	2.0
0	75	3.0
0	100	4.0

Table 9.7: Relation C_1_UP

$leaf\_num$	$attr_1$	$count$
0	15	2.0
0	60	2.0
0	62	3.0
0	65	3.0
0	75	3.0
0	100	3.0

Table 9.8: Relation C_2_UP

Phase 3: Partitioning. According to the best split found in Phase 2, we grow the tree and partition the training set. The partition is reflected as *leaf_num* updates in relation *DETAIL*. Any new grown node that is pure or "small enough" is marked and reassigned a special *leaf_num* value *STOP* so that it is not processed further. The tree is shown in Figure 9.3 and the new *DETAIL* is shown in Table 9.14. Again,

$leaf\_num$	$attr_1$	$count$
0	15	4.0
0	60	3.0
0	62	3.0
0	65	2.0
0	75	1.0

Table 9.9: Relation C_1_DOWN

$leaf\_num$	$attr_1$	$count$
0	15	1.0
0	60	1.0
0	62	0.0
0	65	0.0
0	75	0.0

Table 9.10: Relation C_2_DOWN

$leaf\_num$	$attr_1$	$gini$
0	15	0.22856
0	60	0.40474
0	62	0.21428
0	65	0.34284
0	75	0.42856

Table 9.11: Relation $GINI\_VALUE$

$leaf\_num$	$attr\_name$	$attr\_value$	$gini$
0	1	62	0.21428

Table 9.12: Relation MIN_GINI after $attr_1$ is evaluated

$leaf\_num$	$attr\_name$	$attr\_value$	$gini$
0	1	62	0.21428
0	2	30	0.21428

Table 9.13: Relation MIN_GINI after $attr_1$ and $attr_2$ are evaluated

note $leaf\_num$ is never stored in $DETAIL$, so no update to $DETAIL$ is necessary.

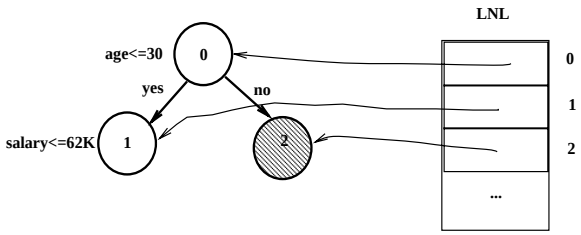


Figure 9.3: Decision tree at Phase 3

$attr_1$	$attr_2$	*class*	*leaf_num*
65K	30	Safe	1
15K	23	Risky	1
75K	40	Safe	2⇒*STOP*
15K	28	Risky	1
100K	55	Safe	2⇒*STOP*
60K	45	Safe	2⇒*STOP*
62K	30	Risky	1

Table 9.14: Relation *DETAIL* with implicit *leaf_num* after Phase 3

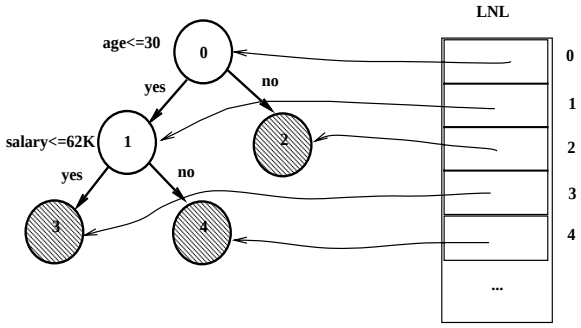


Figure 9.4: Final decision tree

Phase 4: Repeat Phase 1 through Phase 3 until all the leaves in the tree become *STOP* leaves. The final tree and *DETAIL* are shown in Figure 9.4 and Table 9.15.

9.6 Performance Analysis

Building classifiers for large training sets is an I/O bound application [55, 77]. In this section we analyze the I/O complexity of both MIND and SPRINT and compare their performances.

$attr_1$	$attr_2$	$class$	$leaf\_num$
65K	30	Safe	$4 \Rightarrow STOP$
15K	23	Risky	$3 \Rightarrow STOP$
75K	40	Safe	$STOP$
15K	28	Risky	$3 \Rightarrow STOP$
100K	55	Safe	$STOP$
60K	45	Safe	$STOP$
62K	30	Risky	$3 \Rightarrow STOP$

Table 9.15: Final relation *DETAIL* with implicit *leaf_num*

As we described in Section 9.3.1, the classification algorithm iteratively does two main operations: computing the splitting index (in our case, the *gini* index) and performing the partition. SPRINT [77] forms an *attribute list* (projection of the *DETAIL* table) for each attribute. In order to reduce the cost of computing the *gini* index, SPRINT presorts each attribute list and maintains the sorted order throughout the course of the algorithm. However, the use of attribute lists complicates the partitioning operation. When updating the leaf information for the entries in an attribute list corresponding to some attribute that is *not* the splitting attribute, there is no local information available to determine how the entries should be partitioned. A *hash table* (whose size is linear in the number of training examples that reach the node) is repeatedly queried by random access to determine how the entries should be partitioned. In large data mining applications, the hash table is therefore not memory-resident, and several extra I/O passes may be needed, resulting in highly nonlinear performance.

MIND avoids the external memory thrashing during the partitioning phase by the use of dimension tables DIM_i that are formed while the *DETAIL* table, consisting of all the training examples, is streamed through memory. In practice, the dimension tables will likely fit in memory, as they are much smaller than the *DETAIL* table, and often preprocessing is done by discretizing the examples to make the number of

distinct attribute values small. While vertical partitioning of *DETAIL* may also be used to compute the dimension tables in linear time, we show that it is not a must. Data in and data archived from commercial databases are mostly in row major order. The layout does not appear to hinder performance.

If the dimension tables cannot fit in memory, they can be formed by sorting in linear time, if we make the weak assumption that $(M/B)^c \geq D/B$ for some small positive constant c, where D, M, and B are the dimension table size, the internal memory size, and the block size [10, 88], respectively. This optimization can be obtained automatically if SQL has the multiple grouping operator proposed in Section 9.4.1 and with appropriate query optimization, or by appropriate restructuring of the SQL operations. The dimension tables themselves are used in a stream fashion when forming the *UP* and *DOWN* relations. The running time of the algorithm thus scales linearly in practice with the training set size.

Now we turn to the detailed analysis of the I/O complexity of both algorithms. We still use the I/O model defined in Section 6.1, but since the algorithms deal with data items of different sizes, we will use the parameters in Table 9.16 (all sizes are measured in bytes) in our analysis.

Each record in *DETAIL* has n attribute values of size r_a, plus a class label that we assume takes one (byte). Thus we have $r = nr_a + 1$. For simplicity we regard r_a as some unit size and thus $r = O(n)$. Each entry in a dimension table consists of one node number, one attribute value, one class label and one count. The largest node number is 2^L, and it can therefore be stored in L bits, which for simplicity we assume can fit in one word of memory. (Typically L is on the order of 10–20. If desired, we can rid ourselves of this assumption on L by rearranging *DETAIL* or a copy of *DETAIL* so that no *leaf_num* field is needed in the dimension tables, but in practice this is not needed.) The largest count is N, so $r_d = O(\log N)$. Counts are

Notation	*Definition*
M	size of internal memory
B	size of disk block
N	# of rows in *DETAIL*
n	# of attributes in *DETAIL* (not including class label)
C	# of distinct class labels
L	depth of the final classifier
D_k	total size of all dimension tables at depth k
V	# of distinct values for all attributes
r	size of each record in *DETAIL*
r_a	size of each attribute value in *DETAIL* (for simplicity, we assume that all attribute values are of similar size.)
r_d	size of each record in a dimension table
r_h	size of each record in a hash table (used in SPRINT)

Table 9.16: Parameters used in analysis of classification algorithms.

used to record multiple instances of a common value in a compressed way, so they always take less space than the original records they represent. We thus have

$$D_k \leq \min\{nN, VC2^k r_d\}. \tag{9.3}$$

In practice, the second expression in the min term is typically the smaller one, but in our worst-case expressions below we will often bound D_k by nN.

Lemma 9.1 *If all dimension tables fit in memory, that is, $D_k \leq M$ for all k, the I/O complexity of MIND is*

$$O\left(\frac{LnN}{B}\right), \tag{9.4}$$

which is essentially best possible.

Proof: If all dimension tables fit in memory, then we only need to read *DETAIL* once at each level. Dimension tables for all attributes are accumulated in memory when each *DETAIL* record is read in. When the end of *DETAIL* table is reached, we'll have all the unsorted dimension tables in memory. Then sorting and *gini* index

computation are performed for each dimension table, best split will be found for each current leaf node.

The I/O cost to read in *DETAIL* once is $rN/B = O(nN/B)$, and there are L levels in the final classifier, so the total I/O cost is $O(LnN/B)$. □

Lemma 9.2 *In the case when not all dimension tables fit in memory at the same time, but each individual dimension table does, the I/O complexity of MIND is*

$$O\left(\frac{LnN}{B}\log_{M/B} n\right). \tag{9.5}$$

Proof: In the case when not all dimension tables fit in memory at the same time, but each individual dimension table does, we can form, use and discard each dimension table on the fly. This can be done by a single pass through the *DETAIL* table when $M/n > B$ (which is always true in practice).

MIND keeps a buffer of size $O(M/n)$ for each dimension. In scanning *DETAIL*, for each dimension, its buffer is used to store the accumulated information. Whenever a buffer is full, it is written to disk. When the scanning of *DETAIL* is finished, many blocks have been obtained for each dimension based on which the final dimension table can be formed easily. For example, there might be two entries $(1, 1, 1, count_1)$, $(1, 1, 1, count_2)$ in two blocks for $attr_1$. They are corresponding to an entry with $leaf\_num = 1$, $class = 1$, $attr_1 = 1$ in the final dimension table for $attr_1$ and will become a entry $(1, 1, 1, count_1 + count_2)$ in the final dimension table. All those blocks that corresponds to one dimension are collectively called an *intermediate* dimension table for that dimension.

Now the intermediate dimension table for the first attribute is read into memory, summarized, and sorted into a final dimension table. Then MIND calculates the *gini* index values with respect to this dimension for each leaf node, and keeps the current

minimum *gini* index value and the corresponding $(attribute\_name, attribute\_value)$ pair in each leaf node. When the calculation for the first attribute is done, the in-memory dimension table is discarded. MIND repeats the same procedure for the second attribute, and so on. Finally, we get the best splits for all leaf nodes and we are ready to grow the tree one more level. The I/O cost at level k is scanning *DETAIL* once, plus writing out and reading in all the intermediate dimension tables once. We denote the total size of all intermediate dimension tables at level k by D'_k. Note that the *intermediate* dimension tables are a compressed version of the original *DETAIL* table, and they take much less space than the original records they represent. So we have

$$D'_k \leq nN.$$

The I/O cost at each level is

$$O(\frac{1}{B}\sum_{0\leq k<L} D'_k + \frac{LnN}{B}) = O(\frac{LnN}{B}).$$

In the very unlikely scenario where $M/n < B$, a total of $\log_{M/B} n$ passes over *DETAIL* are needed, resulting in a total I/O complexity in (9.5). □

Now we consider the worst case in which some individual dimension tables do not fit in memory. We employ a merge sort process. An interesting point is that the merge sort process here is different from the traditional one: After several passes in the merge sort, the lengths of the runs will not increase anymore; they are upper bounded by the number of rows in the final dimension tables, whose size, although too large to fit in memory, is typically small in comparison with N.

We formally define the special sort problem:

Definition 9.1 Consider N date items, each is of form $(k_{x(i)}, count_i)$, for $1 \leq i \leq N$, where $k_{x(i)}$ is the *key* of the ith data item. There are $N'(N' \ll N)$ distinct keys, $\{k_1, k_2, \ldots, k_{N'}\}$, and we assume $k_1 < k_2 < \ldots < k_{N'}$ for simplicity.

The goal is to obtain N' data items with the keys in sorted increasing order and the corresponding *count* summarized; that is, $(k_i, COUNT_i)$, where

$$COUNT_i = \sum_{1 \le k \le N, x(k)=i} count_k$$

for $1 \le i \le N'$.

Lemma 9.3 *The I/O complexity of the special sort problem is*

$$O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N'}{B}\right) \tag{9.6}$$

Proof: Denote $n = N/B$, $m = M/B$, and $n' = N'/B$.

We perform a modified merge sort procedure for the special sort problem. The first N/M sorted "runs" are formed by repeatedly filling up the internal memory, sorting the records according to their key values, combining the records with the same key and summarizing their counts, and writing the results to disk. This requires $O(N/B)$ I/Os. Next M/B runs are continually merged and combined together into a longer sorted run, until we end up with one sorted run containing all the N' records.

In a traditional merge sort procedure, the crucial property is that we can merge M/B runs together in a linear number of I/Os. To do so we simply load a block from each of the runs and collect and output the B/c smallest elements, where c is the size (in bytes) of each data item. We continue this process until we have processed all elements in all runs, loading a new block from a run every time a block becomes empty. Since there are $O(\log_m n/m)$ levels in the merge process, and each level requires $O(n)$ I/O operations, we obtain the $O(n \log_m n)$ complexity for the normal sort problem.

An important difference between the special sort procedure and the traditional one is that in the former, the length of each sorted run will not go beyond N' while

in the later, the length of sorted runs at each level keeps increasing (doubling) until reaching N.

In the special sort procedure, at and after level $k = \lceil \log_{M/B} N'/B \rceil$, the length of any run will be bounded by N' and the number of runs is bounded by $\lceil n/m^k \rceil$. (For simplicity, we will ignore all the floors and ceilings in the following discussion.) From level $k+1$ on, the operation we perform at each level is basically combining each m runs (each with a length less than or equal to N') into one run whose length is still bounded by N'. We repeat this operation at each level until we get a single run. At level $k+i$, we combine n/m^{k+i-1} runs into n/m^{k+i} runs and the I/O at this level is

$$\frac{n}{m^{i-1}} \left(1 + \frac{1}{m}\right).$$

We shall finish the combining procedure at level $k+p$ where $p = \log_m n/n'$. So the I/O for the whole special sort procedure is:

$$\begin{aligned}
&2nk + n(1+1/m) + \tfrac{n}{m}(1+1/m) + \ldots + \tfrac{n}{m^{p-1}}(1+1/m) \\
&\leq 2n \log_m n' + n(1+1/m)\tfrac{1}{1-1/m} \\
&\approx 2n \log_m n' + n \\
&= O(n \log_m n' + n) \\
&= O(\tfrac{N}{B} \log_{\frac{M}{B}} \tfrac{N'}{B}).
\end{aligned}$$

□

Now we are ready to give the I/O complexity of MIND in the worst case.

Theorem 9.1 *In the worst case the I/O complexity of MIND is*

$$O\left(\frac{nNL}{B} + \frac{nN}{B} \sum_{0 \leq k < L} \log_{M/B} \frac{D_k}{B}\right), \tag{9.7}$$

which is

$$O\left(\frac{LnN}{B} \frac{\log \frac{nN}{B}}{\log \frac{M}{B}}\right). \tag{9.8}$$

In most applications, the log *term is negligible, and the I/O complexity of MIND becomes*

$$O\left(\frac{LnN}{B}\right), \tag{9.9}$$

which matches the optimal time of (9.4).

Proof: The proof here is similar to that of Lemma 9.2. At level k of the tree growth phase, MIND first forms all the intermediate dimension tables with total size $D_k^{'}$ in external memory. This can be done by a single pass through the *DETAIL* table, as follows. MIND keeps a buffer of size $O(M/n)$ for each dimension. In scanning *DETAIL*, MIND accumulates information for each dimension in its corresponding buffer; whenever a buffer is full, it is written to disk. When the scanning of *DETAIL* is finished, MIND performs the special merge sort procedure for the disk blocks corresponding to all (not individual) dimension tables. At the last level of the special sort, the final dimension table for each attribute will be formed one by one. MIND calculates the *gini* index values with respect to each dimension for each leaf node, and keeps the current minimum *gini* index value and the corresponding $(attribute\_name, attribute\_value)$ pair in each leaf node. When the calculation for the last attribute is done, we get the best splits for all leaf nodes and we are ready to grow the tree one more level.

The I/O cost at level k is scanning *DETAIL* once, which is $O(nN/B)$, plus the cost of writing out all the *intermediate* dimension tables once, which is bounded by $O(nN/B)$, plus the cost for the special sort, which is $O(\frac{N}{B}\log_{M/B} D_k/B)$.

So the I/O for all levels is

$$\frac{LnN}{B} + \frac{1}{B}\sum_{0\le k<L} D_k^{'} + \frac{nN}{B}\sum_{0\le k<L}\log_{M/B}\frac{D_k}{B}$$

which is

$$O\left(\frac{LnN}{B} + \frac{nN}{B}\sum_{0\le k<L}\log_{M/B}\frac{D_k}{B}\right).$$

□

Now we analyze the I/O complexity of the SPRINT algorithm. There are two major parts in SPRINT: the pre-sorting of all attribute lists and the constructing/searching of the corresponding hash tables during partition. Since we are dealing with a very large *DETAIL* table, it is unrealistic to assume that N is small enough to allow hash tables to be stored in memory. Actually those hash tables need to be stored on disk and brought into memory during the partition phase. It is true that hash tables will become smaller at deeper levels and thus fit in memory, but at the early levels they are very large; for example, the hash table at level 0 has N entries.

Each entry in a hash table contains a *tid*(transaction identifier) which is an integer in the range of 1 to N, and one bit that indicates which child this record should be partitioned to in the next level of the classifier. So we have

$$r_h = \frac{1 + \log N}{8}.$$

We can estimate when the hash tables will fit in memory, given the optimistic assumptions that all memory is allocated to hash tables and all hash tables at each node have equal size; that is, a hash table at level k contains $N/2^k$ entries. Thus, a hash table at level k fits in memory if $r_h N/2^k \leq M$, or

$$2^k \geq \frac{N}{M}\left(\frac{1 + \log N}{8}\right). \tag{9.10}$$

For sufficiently large k, (9.10) will be satisfied, that is, hash tables become smaller at deeper nodes and thus fit in memory. But it is clear that even for moderately large detail tables, hash tables at upper levels will not fit in memory.

During the partition phase, each non-splitting attribute list at each node needs to be partitioned into two parts based on the corresponding hash table. One way to

do this is to do a random hash table search for each entry in the list, but this is very expensive. Fortunately, there is a better way: First, we bring a large portion of the hash table into memory. The size of this portion is limited only by the availability of the internal memory. Then we scan the non-splitting list once, block by block, and for each entry in the list, we search the in-memory portion of the hash table. In this way, the hash table is swapped into memory only once, and each non-splitting attribute list is scanned N/M times. For even larger N, it is better to do the lookup by batch sorting, but that approach is completely counter to the founding philosophy of the SPRINT algorithm.

A careful analysis gives us the following estimation:

Theorem 9.2 *The I/O complexity of SPRINT is*

$$O\left(\frac{nN^2 \log N}{BM}\right) \tag{9.11}$$

Proof: To perform the pre-sort of the SPRINT algorithm, we need to read *DETAIL* once, write out the unsorted attribute lists, and sort all the attribute lists. So we have

$$IO_{presort} = O\left(\frac{nN}{B} \log_{\frac{M}{B}} \frac{N}{B}\right).$$

From level 0 through level $k-1$, hash tables will not fit in memory. At level i $(0 \leq i \leq k-1)$, SPRINT will perform the following operations:

1. Scan the attribute lists one by one to find the best split for each leaf node.
2. According to the best split found for each leaf node, form the hash tables and write them to disk.
3. Partition the attribute list of the splitting attribute for each leaf node.

4. Partition the attribute lists for the $n-1$ non-splitting attributes for each leaf node.

Among these operations, the last one incurs the most I/O cost and we perform it by bringing a portion of a hash table into memory first. The size of this portion is limited only by the availability of the main memory. Then we scan each non-splitting list once, block by block, and for each entry in the list, we search the in-memory portion of the hash table and decide which child this entry should go in the next level. In this way, the hash table is swapped into memory only once, and the non-splitting list is scanned multiple times. The I/O cost of this operation is

$$O\left(\frac{nNh_i}{B}\right)$$

where h_i is the number of portions we need to partition a hash table into due to the limitation of the memory size.

From level k to level L the hash table will fit in memory, and the I/O costs for those levels is $O((L-k)nN/B)$, which is significantly smaller than those for the previous levels.

So the I/O cost of SPRINT becomes

$$O\left(\frac{nN}{B}\log_{\frac{M}{B}}\frac{N}{B}+\sum_{0\le i\le k-1}\frac{nNh_i}{B}+\frac{(L-k)nN}{B}\right) \tag{9.12}$$

Note that we have

$$h_i=\frac{r_hN}{2^iM}=\frac{N}{2^iM}\left(\frac{1+\log N}{8}\right)$$

So

$$\frac{N}{M}\left(\frac{1+\log N}{8}\right)\le\sum_{0\le i\le k-1}h_i\le\frac{2N}{M}\left(\frac{1+\log N}{8}\right) \tag{9.13}$$

Applying (9.13) to (9.12), we get the I/O complexity of SPRINT in (9.11). □

Examination of (9.8) and (9.11) reveals that MIND is clearly better in terms of I/O performance. For large N, SPRINT does a quadratic number of I/Os, whereas MIND scales well.

9.7 Algorithm Revisited Using SchemaSQL

In Section 9.4.1, we described the MIND algorithm using SQL-like statements. Due to the limitation of current SQL standards, most of those SQL-like statements are not supported directly in today's DBMS products. Therefore, we need to convert them to currently supported SQL statements, augmented with new facilities like user defined functions. Putting logic within a user-defined function hides the operator from query optimization. If classification was a subquery or part of a large query, it would not be possible to obtain all join reorderings, thereby risking suboptimal execution.

Current SQL standards are mainly designed for efficient OLTP (On-Line Transactional Processing) queries. For non-OLTP applications, it is true that we can usually reformulate the problem and express the solution using standard SQL. However, this approach often results in inefficiency. Extending current SQL with ad-hoc constructs and new optimization considerations might solve this problem in some particular domain, but it is not a satisfactory solution. Since supporting OLAP (On-Line Analytical Processing) applications efficiently is such an important goal for today's RDBMSs, the problem deserves a more general solution.

In [47] an extension of SQL, called SchemaSQL, is proposed. SchemaSQL offers the capability of uniform manipulation of data and meta-data in relational multi-database systems. By examining the SQL-like queries in Section 9.4.1, we can see that this capability is what we need in the MIND algorithm. To show the power of extended SQL and the flexibility and general flavor of MIND, in this section, we

rewrite all the queries in Section 9.4.1 using SchemaSQL.

First we give an overview of the syntax of SchemaSQL. For more details see [47].

In a standard SQL query, the *tuple* variables are declared in the `FROM` clause. A variable declaration has the form $\langle range \rangle \langle var \rangle$. For example, in the query below, the expression *student* T declares T as a variable that ranges over the (tuples of the) relation $student(student\_id, department, GPA)$:

`SELECT` *student_id*

`FROM` *student* T

`WHERE` $T.department = CS$ `AND` $T.GPA = A$

The SchemaSQL syntax extends SQL syntax in several directions:

1. The federation consists of databases, with each database consisting of relations.
2. To permit meta-data queries and reconstruction views, SchemaSQL permits the declaration of other types of variables in addition to the tuple variables permitted in SQL.
3. Aggregate operations are generalized in SchemaSQL to make horizontal and block aggregations possible, in addition to the usual vertical aggregation in SQL.

SchemaSQL permits the declaration of variables that can range over any of the following five sets:

1. names of databases in a federation,
2. names of the relations in a database,
3. names of the columns in the scheme of a relation,
4. tuples in a given relation in database, and
5. values appearing in a column corresponding to a given column in a relation.

Variable declarations follow the same syntax as $\langle range \rangle \langle var \rangle$ as in SQL, where *var* is any identifier. However, there are two major differences:

1. The only kind of *range* permitted in SQL is a set of tuples in some relation in the database, where in SchemaSQL any of the five kinds of *range* can be used declare variables.
2. The *range* specification in SQL is made using constant, i.e. and identifier referring to a specific relation in a database. By contrast, the diversity of *range*s possible in SchemaSQL permits *range* specifications to be *nested*, in the sense that it is possible to say, for example, that R is a variable ranging over the relation names in database D, and that T is a tuple in the relation denoted by R.

Range specifications are one of the following five types of expressions, where *db*, *rel*, *col* are any *constant* or *variable identifiers*.

1. The expression $\rightarrow$ denotes a *range* corresponding to the set of database names in the federation.
2. The expression $db \rightarrow$ denotes the set of relation names in the database *db*.
3. The expression $db :: rel \rightarrow$ denotes the set of names of column in the schema of the relation *rel* in the database *db*.
4. $db :: rel$ denotes the set of tuples in the relation *rel* in the database *db*.
5. $db :: rel.col$ denotes the set of values appearing in the column named *col* in the relation *rel* in the database *db*.

For example, consider the clause `FROM` $db1 \rightarrow R$, $db1 :: R\ T$. It declares R as a variable ranging over the set of relation names in the database $db1$ and T as a variable ranging over the tuples in each relation R in the database $db1$

Now we are ready to rewrite all the SQL-like queries in Section 9.4.1 using SchemaSQL. Assume that our training set is stored in relation *DETAIL* in a database

named *FACT*. We first generate all the dimension tables with the schema (*leaf_num*, *class*, *attr_val*, *count*) in a database named *DIMENSION*, using a simple SchemaSQL statement:

```
CREATE VIEW DIMENSION :: R(leaf_num, class, attr_val, count) AS
SELECT T.leaf_num, T.class, T.R, COUNT(*)
FROM FACT :: DETAIL → R, FACT :: DETAIL T
WHERE R <>' class' AND R <>' leaf_num' AND T.leaf_num <> STOP
GROUP BY T.leaf_num, T.class, T.R
```

The variable R is declared as a column name variable ranging over the column names of relation *DETAIL* in the database *FACT*, and the variable T is declared as a tuple variable on the same relation. The conditions on R in the `WHERE` clause make the variable R range over all columns except the columns named *class* and *leaf_num*. If there are n columns in *DETAIL* (excluding columns *class* and *leaf_num*), this query generates n `VIEW`s in database *DIMENSION*, and the name of each `VIEW` is the same as the corresponding column name in *DETAIL*. Note that the attribute name to relation name transformation is done in a very natural way, and the formation of multiple `GROUP BY`s is done by involving *DETAIL* only once.

Those views will be materialized, so that in the later operations we do not need to access *DETAIL* any more.

Relations corresponding to *UP* with the schema (*leaf_num*, *attr_val*, *class*, *count*) can be generated in a database named *UP* by performing a self-outer-join on dimension tables in database *DIMENSION*:

```
CREATE VIEW UP :: R(leaf_num, attr_val, class, count) AS
SELECT d1.leaf_num, d1.attr_val, d1.class, SUM(d2.count)
FROM (FULL OUTER JOIN DIMENSION :: R d1,
```

$DIMENSION :: R\ d_2,$

$DIMENSION \rightarrow R$

ON $d_1.leaf\_num = d_2.leaf\_num$ AND

$d_1.attr\_val \leq d_2.attr\_val$ AND

$d_1.class = d_2.class$

GROUP BY $d_1.leaf\_num, d_1.attr\_val, d_1.class$)

The variable R is declared as a relation name variable ranging over all the relations in database *DIMENSION*. Variables d_1 and d_2 are both tuple variables over the tuples in each relation R in database *DIMENSION*. For each relation in database *DIMENSION*, a self-outer-join is performed according to the conditions specified in the query, and the result is put into a VIEW with the same name in database *UP*.

Similarly, relations corresponding to *DOWN* can be generated in a database named *DOWN* by just changing the $\leq$ to $>$ in the ON clause.

Database *DOWN* and database *UP* contain all the information we need to compute all the *gini* index values. Since standard SQL only allows vertical aggregations, we need to rearrange them before the *gini* index is actually calculated as in Section 9.4.1. In SchemaSQL, aggregation operations are generalized to make horizontal and block aggregations possible. Thus, we can generate views that contain all *gini* index values at each possible split point for each attribute in a database named *GINI_VALUE* directly from relations in *UP* and *DOWN*:

CREATE VIEW $GINI\_VALUE :: R(leaf\_num, attr\_val, gini)$ AS

SELECT $u.leaf\_num, u.attr\_val, f_{gini}$

FROM $UP :: R\ u,\ DOWN :: R\ d,\ UP \rightarrow R$

WHERE $u.leaf\_num = d.leaf\_num$ AND $u.attr\_val = d.attr\_val$

GROUP BY $u.leaf\_num, u.attr\_val$

where f_{gini} is a function of $u.class$, $d.class$, $u.count$, $d.count$ according to (9.1) and (9.2).

Variable R is declared as a variable ranging over the set of relation names in database UP, u is a variable ranging over the tuples in each relation in database UP, and d is a variable ranging over the tuples in the relation with the same name as R in database $DOWN$. Note that the set of relation names in databases UP and $DOWN$ are the same. For each of the relation pairs with the same name in UP and $DOWN$, this statement will create a view with the same name in database $GINI\_VALUE$ according to the conditions specified. It is interesting to note that f_{gini} is a block aggregation function instead of the usual vertical aggregation function in SQL. Each view named R in database $GINI\_VALUE$ contains the *gini* index value at each possible split point with respect to attribute named R.

Next, we create a *single* view MIN_GINI with the schema $MIN\_GINI(leaf\_num,$ $attr\_name,$ $attr\_val,$ $gini)$ in a database named $SPLIT$ form the *multiple* views in database $GINI\_VALUE$:

```
CREATE VIEW SPLIT :: MIN_GINI( leaf_num, attr_name, attr_val, gini) AS
SELECT T1.leaf_num, R1, T1.attr_val, gini
FROM GINI_VALUE → R1, GINI_VALUE :: R1 T1
WHERE T1.gini =(SELECT MIN(T2.gini)
                FROM GINI_VALUE → R2, GINI_VALUE :: D2 T2
                WHERE R1 = R2 AND T1.leaf_num = T2.leaf_num)
```

Variables R_1 and R_2 are variables ranging over the set of relation names in database $GINI\_VALUE$. Variables T_1 and T_2 are tuple variables ranging over the tuples in relations specified by R_1 and R_2, respectively. The clause $R_1 = R_2$ enforces R_1 and R_2 to be the same relation. Note that relation name R_1 in database $GINI\_VALUE$ becomes the column value for the column named $attr\_name$ in re-

lation *MIN_GINI* in database *SPLIT*. Relation *MIN_GINI* now contains the best split value and the corresponding *gini* index value for each leaf node of the tree with respect to all attributes.

The overall best split for each leaf is obtained from executing the following:

```
CREATE VIEW SPLIT::BEST_SPLIT( leaf_num, attr_name, attr_val) AS
SELECT T1.leaf_num, T1.attr_name, T1.attr_val
FROM SPLIT::MIN_GINI T1
WHERE T1.gini =(SELECT MIN(gini)
                FROM SPLIT::MIN_GINI T2
                WHERE T1.leaf_num = T2.leaf_num)
```

This statement is similar to the statement generating relation *BEST_SPLIT* in Section 9.4.1. T_1 is declared as a tuple variable ranging over the tuples of relation *MIN_GINI* in database *SPLIT*. For each *leaf_num*, (*attr_name*, *attr_val*) pair that achieving the minimum *gini* index value is inserted into relation *BEST_SPLIT*.

We have shown how to rewrite all the SQL-like queries in MIND algorithm using SchemaSQL. In our current prototype of MIND, the first step, generating all the dimension tables from *DETAIL*, is most costly and all the later steps only need to access small dimension tables. We use udf to reduce the cost of the first step. All the SQL-like queries in Section 9.4.1 in the later steps are translated into equivalent SQL queries. Those translations usually lead to poor performance. But since those queries only access small relations in MIND, the performance loss is negligible. While udf provides a solution to our classification algorithm, we a believe general extension of SQL is needed for efficient support of OLAP applications.

An alternative way to generate all the dimension tables from *DETAIL* would be using the newly proposed *data cube* operator [30] since dimension tables are different subcubes. But it usually takes a long time to generate the data cube without

precomputation and the fact that the *leaf_num* column in *DETAIL* keeps changing from level to level when we grow the tree makes precomputation infeasible.

9.8 Experimental Results

There are two important metrics to evaluate the quality of a classifier: *classification accuracy* and *classification time*. We compare our results with those of SLIQ [55] and SPRINT [77]. (For brevity, we include only SPRINT in this chapter; comparisons showing the improvement of SPRINT over SLIQ are given in [77].) Unlike SLIQ and SPRINT, we use the classical database methodology of summarization. Like SLIQ and SPRINT, we use the same metric (*gini* index) to choose the best split for each node, we grow our tree in a breadth-first fashion, and we prune it using the same pruning algorithm. Our classifier therefore generates a decision tree identical to the one produced by [55, 77] for the same training set, which facilitates meaningful comparisons of run time. The accuracy of SPRINT and SLIQ is discussed in [55, 77], where it is argued that the accuracy is sufficient.

For our scaling experiments, we ran our prototype on large data sets. The main cost of our algorithm is that we need to access *DETAIL* n times (n is the number of attributes) for each level of the tree growth due to the absence of the multiple `GROUP BY` operator in the current SQL standard. We recommend that future DBMSs support the multiple `GROUP BY` operator so that *DETAIL* will be accessed only once regardless of the number of attributes. In our current working prototype, this is done by using user-defined function as we described in Section 9.4.1.

Owing to the lack of a classification benchmark, we used the synthetic database proposed in [4]. In this synthetic database, each record consists of nine attributes as shown in Table 9.17. Ten classifier functions are proposed in [4] to produce databases with different complexities. We run our prototype using function 2. It generates a

Attribute	*Value*
salary	uniformly distributed from 20K to 150K
commission	$salary \geq 75K \Rightarrow commission = 0$ else uniformly distributed from 10K to 75K
age	uniformly distributed from 20 to 80
loan	uniformly distributed from 0 to 500K
elevel	uniformly chosen from 0 to 4
car	uniformly chosen form 1 to 20
zipcode	uniformly chosen from 10 available zipcodes
hvalue	uniformly distributed from $0.5k100000$ to $1.5k100000$, where $k \in \{0, \ldots, 9\}$ is zipcode
hyear	uniformly distributed from 1 to 30

Table 9.17: Description of the synthetic data.

database with two classes: Group A and Group B. The description of the class predicate for Group A is shown below.

Function 2, Group A

$((\text{age} < 40) \wedge (50\text{K} \leq \text{salary} \leq 100\text{K})) \vee$

$((40 \leq \text{age} < 60) \wedge (75\text{K} \leq \text{salary} \leq 125\text{K})) \vee$

$((\text{age} \geq 60) \wedge (25\text{K} \leq \text{salary} \leq 75\text{K}))$

Our experiments were conducted on an IBM RS/6000 workstation running AIX level 4.1.3. and DB2 version 2.1.1. We used training sets with sizes ranging from 0.5 million to 5 million records. The relative response time and response time per example are shown in Figure 9.5 and Figure 9.6 respectively. Figure 9.5 hints that our algorithm achieves linear scalability with respect to the training set size. Figure 9.6 shows that the time per example curve stays flat when the training set size increases. The corresponding curve for [77] appears to be growing slightly on the largest cases. Figure 9.7 is the performance comparison between MIND and SPRINT. MIND ran on a processor with a slightly slower clock rate. We can see that MIND performs better than SPRINT does even in the range where SPRINT scales well, and MIND continues to scale well as the data sets get larger.

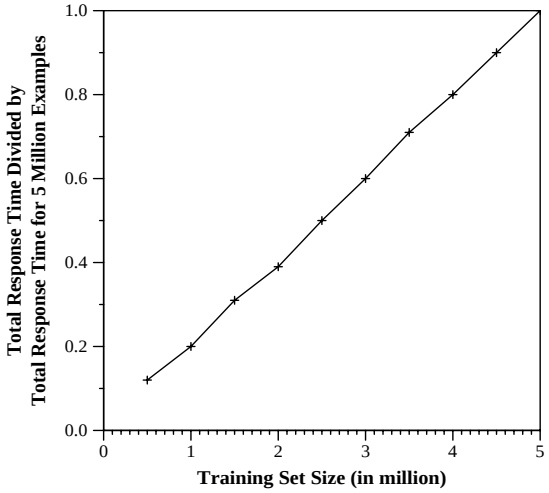


Figure 9.5: Relative total response time. The y-value denotes the total response time for the indicated training set size, divided by the total response time for 5 million examples.

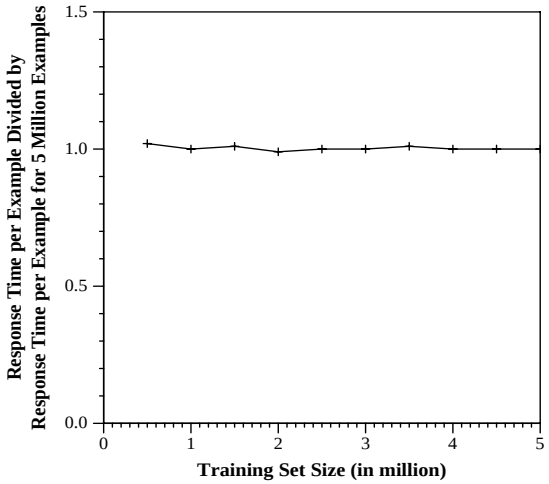


Figure 9.6: Relative response time per example. The y-value denotes the response time per example for the indicated training set size, divided by response time per example when processing 5 million examples.

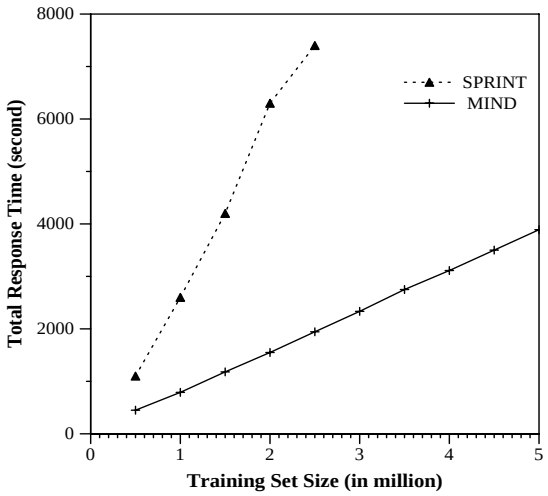


Figure 9.7: Performance comparison of MIND and SPRINT

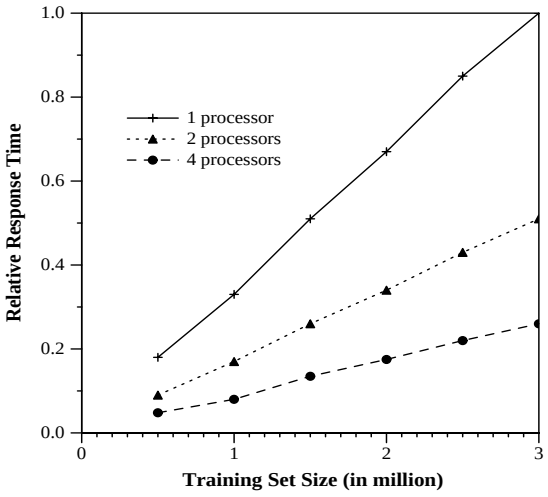


Figure 9.8: Speedup of MIND for multiprocessors. The y-value denotes the total response time for the indicated training set size, divided by the total response time for 3 million examples.

We also ran MIND on an IBM multiprocessor SP2 computer system. Figure 9.8 shows the parallel speedup of MIND.

Another interesting measurement we obtained from uniprocessor execution is that accessing *DETAIL* to form the dimension tables for all attributes takes 93%–96% of the total execution time. To achieve linear speedup on multiprocessors, it is critical that this step is parallelized. In the current working prototype of MIND, it is done by user-defined function with a scratch-pad accessible from multiple processors.

9.9 Conclusions

The MIND algorithm solves the problem of classification within the relational database manager. Our performance measurements show that MIND demonstrates scalability with respect to the number of examples in training sets and the number of parallel processors. We believe MIND is the first classifier to successfully run on datasets of $N = 5$ million examples on a uniprocessor and yet demonstrate effectively non-increasing response time per example as a function of N. It also runs faster than previous algorithms on file systems.

There are four reasons why MIND is fast, exhibits excellent scalability, and is able to handle data sets larger than those tackled before:

1. MIND rephrases the data mining function classification as a classic DBMS problem of summarization and analysis thereof.
2. MIND avoids any update to the *DETAIL* table of examples. This is of significant practical interest; for example, imagine *DETAIL* having billions of rows.
3. In the absence of a multiple concurrent grouping SQL operator, MIND takes advantage of the user-defined function capability of DB2 to achieve the equivalent functionality and the resultant performance gain.
4. Parallelism of MIND is obtained at little or no extra cost because the RDBMS

parallelizes SQL queries.

We recommend that extensions be made to SQL to do multiple groupings and the streaming of each group to different relations. Most DBMS operators currently take two streams of data (tables) and combine them into one. We believe that we have shown the value of an operator that takes a single stream input and produces multiple streams of outputs.

Bibliography

[1] P. Adrians and D. Zantinge. *Data Mining.* Addison-Wesley, 1996.

[2] S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi, and R. Ramakrishnan. On the computation of multidimensional aggregates. In *Proceedings of the 1996 International Conference on Very Large Databases*, Mumbai, India, 1996.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of the 1992 International Conference on Very Large Databases*, pages 560–573, Vancouver, Canada, August 1992.

[5] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, Dec. 1993.

[6] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[7] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49:208–222, 1994.

[8] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, , V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *Bulletin of the Technical Committee on Data Engineering*, 20(4), 1997.

[9] C. K. Baru et al. DB2 parallel edition. *IBM Systems Journal*, 34(2), 1995.

[10] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 1997.

[11] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reordering of outer join queries with complex predicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[12] L. Breiman et al. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[13] U.S. Census Bureau. Census bureau databases. The online data are available on the web at `http://www.census.gov/`.

[14] J. Catlett. *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, University of Sydney, 1991.

[15] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann, 1996.

[16] D. Chamberlin. Personal communication, 1997.

[17] D. Chamberlin et al. Seqel: A structured english query language. In *Proc. of ACM SIGMOD Workshop on Data Description, Access, and Control*, May 1974.

[18] S. Christodoulakis. Estimating block transfers and join results. In *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*, pages 40–50, 1983.

[19] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6), June 1970.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Press, 1993.

[21] T. Dietterich, M. Kearns, and Y. Mansour. Applying the weak learning framework to understand and improve C4.5. In *Proceedings of the 13th International Conference on Machine Learning*, pages 96–104, 1996.

[22] D. L. Donoho. Unconditional bases are optimal bases for data compression and statistical estimation. Technical report, Department of Statistics, Stanford University, 1992.

[23] C. Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering information from summary data. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[24] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[25] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queried revisited. In *Proceeding of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987.

[26] E. Gelenbe and D. Gardy. The size of projection of relations satisfying a functional dependency. In *Proceedings of the 1981 International Conference on Very Large Databases*, pages 325–333, 1981.

[27] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998.

[28] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[29] D. E. Goldberg. *Generic algorithms in Search, Optimization and Machine Learning.* Morgan Kaufmann, 1989.

[30] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and subtotals. In *Proceedings of the 12th Annual IEEE Conference on Data Engineering (ICDE '96)*, pages 131–139, 1996.

[31] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, U.K., April 1997.

[32] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 1992.

[33] P. Haas and A. Swami. Sampling-based selectivity for joins using augmented frequent value statistics. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, March 1995.

[34] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[35] S. Hasty. Mining databases. *Apparel Industry Magazine*, 57(5), 1996.

[36] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[37] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[38] IBM. *IBM DATABASE 2 Application Programming Guide-for common servers*, version 2 edition.

[39] T. Imielinsk and H. Mannila. A database perspective on knowledge discovery. *Communication of the ACM*, 39(11), November 1996.

[40] T. Imielinski. From file mining to database mining. In *Proceedings of the 1996 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, May 1996.

[41] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 249–260. ACM Press, 1999.

[42] M. James. *Classification Algorithms*. Wiley, 1985.

[43] B. Jawerth and W. Sweldens. An overview of wavelet based multiresolution analyses. *SIAM Rev.*, 36(3):377–412, 1994.

[44] M. Kearns and Y. Mansour. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 459–468, 1996.

[45] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*. PhD thesis, Brown University, 1995.

[46] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcard. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 282–293, Montreal, Canada, May 1996.

[47] L. Lakshmanan, F. Sadri, and I. Subramanian. SchemaSQL–a language for interoperability in relational multi-database systems. In *Proceedings of the 1996 International Conference on Very Large Databases*, 1996.

[48] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(22), April 1987.

[49] R. Lipton and J. Naughton. Query size estimation by adaptive sampling. *J. of Comput. Sys. Sci.*, 51:18–25, 1985.

[50] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceeding of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 1–11, 1990.

[51] H. Lu et al. On preprocessing data for efficient classification. In *Proceedings of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, May 1996.

[52] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

[53] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, Seattle, WA, June 1998.

[54] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[55] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, March 1996.

[56] H. Messatfa. Personal communications, 1997.

[57] D. Michie, D. J. Spiegelhalter, and C. C. Tayler. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[58] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 28–36, 1988.

[59] S. K. Murthy. *On Growing Better Decision Trees from Data.* PhD thesis, Johns Hopkins University, 1995.

[60] T. Nguyen and V. Srinivasan. Accessing relational databases from the world

web web. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

[61] N. Pendse and R. Creeth. The OLAP report, 1998. The online report is available on the web at `http://www.olapreport.com/Analyses.htm/`.

[62] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 256–276, 1984.

[63] V. Poosala. *Histogram-Based Estimation Techniques in Database Systems*. Ph. D. dissertation, University of Wisconsin-Madison, 1997.

[64] V. Poosala. Personal communication, 1997.

[65] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 1996 International Conference on Very Large Databases*, Bombay, India, September 1996.

[66] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[67] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[68] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[69] J. Ross Quilan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[70] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 1997 International Conference on Very Large Databases*, pages 476–485, Athens, Greece, August 1997.

[71] W. Rudin. *Functional Analysis*. McGraw-Hill, 1973.

[72] S. Sarawagi. Query processing in tertiary memory databases. In *Proceedings of the 1995 International Conference on Very Large Databases*, 1995.

[73] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the 11th Annual IEEE Conference on Data Engineering (ICDE '94)*, Houston, Texas, 1994.

[74] S. Sarawagi and M. Stonebraker. Benefits of reordering execution in tertiary memory databases. In *Proceedings of the 1996 International Conference on Very Large Databases*, 1996.

[75] J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research, Volume 4*, pages 117–146. JAI Press, 1987.

[76] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[77] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 1996 International Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996.

[78] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramaswamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 1996 International Conference on Very Large Databases*, pages 522–531, 1996.

[79] J. B. Sinclair. Personal communication, 1997.

[80] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann, 1996.

[81] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[82] W. Sun, Y. Ling, N. Rishe, and Yi Deng. An instant and accurate size estimation method for joins and selections in a retrieval-intensive environment. In *Proceedings of the 1983 ACM SIGMOD International Conference on Management of Data*, pages 79–88, 1983.

[83] W. Sweldens. Personal communication, 1997.

[84] TPC benchmark D (decision support), 1995.

[85] J. Ullman. *Principles of Database Systems.* Computer Science Press, second edition, 1982.

[86] D. E. Vengroff. A transparent parallel I/O environment. In *Proceedings of the 1994 DAGS Symposium on Parallel Computa tion*, July 1994.

[87] D. E. Vengroff. *TPIE User Manual and Reference.* Duke University, 1997. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[88] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

[89] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985.

[90] J. S. Vitter. An efficient algorithm for sequential random sampling. *ACM Transactions on Mathematical Software*, 13(1):58–67, March 1987.

[91] J. S. Vitter. External memory algorithms. In *Proceedings of the 1998 ACM Symposium on Principles of Database Systems*, June 1998. Invited tutorial.

[92] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, to appear 1999. The updated and expanded version is available via the author's web page `http://www.cs.duke.edu/~jsv/`.

[93] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994. Special double issue on Large-Scale Memories.

[94] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 193–204, Phildelphia, June 1999.

[95] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of Seventh International Conference on Information and Knowledge Management*, pages 96–104, Washington D.C., November 1998.

[96] M. Wang and B. Iyer. Efficient roll-up and drill-down analysis in relational databases. In *Proceedings of the 1997 ACM-SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, Tucson, AZ, May 1997.

[97] M. Wang, B. Iyer, and J. S. Vitter. Scalable mining for classification rules in relational databases. In *Proceedings of the 1998 International Database Engineering and Applications Symposium*, pages 58–67, Cardiff, Wales, U.K., July 1998.

[98] M. Wang, J. S. Vitter, and B. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proceedings of the 13th Annual IEEE Conference on Data Engineering*, pages 169–180, Birmingham, England, April 1997.

[99] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.

[100] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

Biography

Min Wang received her B.S. and M.S. degrees, both in computer science, from Tsinghua University, Beijing, China in 1988 and 1990, respectively. She was on the faculty of Tsinghua University from 1990 through 1993. She came to Duke University as a Ph.D. student in August 1994. She was a recipient of an IBM Graduate Fellowship from 1997 to 1999.

Min Wang is interested in query optimization, data mining, and online analytical processing (OLAP). Her research has focused on selectivity estimation, approximate query processing, and scalable data mining.

Selected Publications

1. "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets" (with J. S. Vitter), *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, Phildelphia, June 1999, 193–204.

2. "Data Cube Approximation and Histograms via Wavelets" (with J. S. Vitter and B. Iyer), *Proceedings of Seventh International Conference on Information and Knowledge Management (CIKM'98)*, Washington D.C., November 1998, 96–104.

3. "Scalable Mining for Classification Rules in Relational Databases" (with B. Iyer and J. S. Vitter), *Proceedings of International Database Engineering & Applications Symposium (IDEAS'98)*, Cardiff, Wales, U.K., July 1998, 58–67.

4. "Wavelet-Based Histograms for Selectivity Estimation" (with Y. Matias and J. S. Vitter), *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, Seattle, Washington, June 1998, 448–459.

5. "MIND: A Scalable Classifier in Relational Databases" (with B. Iyer and J. S. Vitter), *Proceedings of ACM-SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, June 1998.

6. "Efficient Roll-Up and Drill-Down Analysis in Relational Databases" (with B. Iyer), *Proceedings of ACM-SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'97)*, Tucson, Arizona, May 1997.

7. "Selectivity Estimation in the Presence of Alphanumeric Correlations" (with J. S. Vitter and B. Iyer), *Proceedings of the 13th Annual IEEE Conference on Data Engineering (ICDE'97)*, Birmingham, England, April 1997, 169–180.

8. "A Method for Estimating Filter Factors for Multi-Column Queries" (with T. Beavin, B. Iyer, and H. Tie), *Technical Report*, IBM Santa Teresa Laboratory, August 1995.

9. "An Experimental Study of Alphanumeric Selectivity Estimation" (with T. Beavin, B. Iyer, and H. Tie), *Technical Report*, IBM Santa Teresa Laboratory, August 1995.

Patents

1. "Improved Query Optimization Through the Use of Multi-Column Statistics to Avoid the Problems of Column Correlation," ST9-96-067. Filed in Jan. 1997 (for IBM's DB2).

2. "Scalable Set-Oriented Classifier," ST8960103. Filed in Jan. 1997 (for IBM's Intelligent Miner).