SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads

Lipyeow Lim[1] Min Wang[2] Jeffrey Scott Vitter[3]

[1] Dept. of Computer Science
Duke University
Durham, NC 27708, USA.
lipyeow@cs.duke.edu

[2] IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
min@us.ibm.com

[3] Purdue University
150 N. University Street
West Lafayette, IN 47907 USA
jsv@purdue.edu

Abstract

Most RDBMSs maintain a set of histograms for estimating the selectivities of given queries. These selectivities are typically used for cost-based query optimization. While the problem of building an accurate histogram for a given attribute or attribute set has been well-studied, little attention has been given to the problem of building and tuning a set of histograms collectively for multidimensional queries in a self-managed manner based only on query feedback. In this paper, we present SASH, a Self-Adaptive Set of Histograms that addresses the problem of building and maintaining a set of histograms. SASH uses a novel two-phase method to automatically build and maintain itself using query feedback information only. In the online tuning phase, the current set of histograms is tuned in response to the estimation error of each query in an online manner. In the restructuring phase, a new and more accurate set of histograms replaces the current set of histograms. The new set of histograms (attribute sets and memory distribution) is found using information from a batch of query feedback. We present experimental results that show the effectiveness and accuracy of our approach.

1 Introduction

Estimating the result size of a given query is an important problem in query optimization and approximate query processing. Most RDBMSs maintain a set of histograms using a small amount of memory for selectivity estimation. While the problem of building an accurate histogram for a given attribute or attribute set has been well-studied [13, 12, 11, 16, 1, 3], little attention has been given to the more general problem of building and tuning a set of histograms collectively in a self-managed manner based only on query feedback. Building and tuning a set of histograms for multidimensional queries presents several unique challenges:

1. Which sets of attributes should histograms be built on?
2. How should the histograms be tuned to the query workload?
3. How should the fixed amount of memory be distributed among the histograms to achieve best overall accuracy for estimating the selectivities of a given workload?

In addition, we want to address all these issues by using only query feedback information without performing any offline scan of the underlying database relations.

Related Work. Most previous work [1, 3, 8, 6, 9] studied the above issues by treating them independently and/or assuming full access to the underlying database relations.

The idea of using query feedback information to update the statistics kept by the query optimizer first appeared in [4] where coefficients of a curve representing the underlying data distribution are adapted using query feedback. Self-tuning histograms [1, 3] successfully used this idea to build and maintain individual histograms; however, neither [1] nor [3] addresses the issues of finding which attributes to build histograms on and memory distribution among the set of histograms. Our work addresses these two important issues. Moreover, SASH addresses how to update low dimensional histograms using high dimensional query feedback which has not been addressed in the literature before.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

Graphical statistical models were first used for multidimensional query selectivity estimation in [8, 6]. Getoor et al. [8] use Probabilistic Relational Models (PRMs) for estimating the selectivities of point queries. PRMs [8] are based on directed graph models (Bayesian networks) and they are used in [8] to find which attribute sets to build conditional histograms on. However, the technique proposed in [8] is based on a complete offline scan of the underlying data. It does not address the issues of online construction of histograms, memory distribution among multiple histograms, and workload-driven tuning of the existing histograms. Dependency-based (DB) histograms of Deshpande et al. [6] are based on undirected graph models (in particular, junction trees) and are used for estimating the selectivity of range queries. The technique proposed in [6] addresses the issue of which attribute sets to build histograms on and the issue of memory distribution among multiple histograms. However, it treats these two issues independently. Moreover, similar to the PRM-based technique in [8], it is based on a complete offline scan of the underlying data and does not address online construction and workload-driven tuning.

Jagadish et al. [9] present several greedy and heuristic algorithms for distributing memory (buckets) among a set of single attribute histograms, but does not address the problem of finding the sets of attributes to build histograms on.

All three techniques [8, 6, 9] minimize some objective function that approximates the distance between the joint distribution associated with a set of histograms (or statistics) and the true data distribution of the database. The histograms (or statistics) that they maintain are obtained by scanning the database, and the objective functions that they minimize require accesses to the database to be computed. Because of the offline nature of these techniques, they build histograms without considering how the histograms are being used (i.e., query workload) and assume that all queries are equally likely to be asked. This assumption is rarely true in practice. Ideally, more storage resource should be spent on storing the statistics that are relevant to the most frequently queried portions of the underlying relations. These techniques are oblivious to workload distribution and consequently waste precious storage space in storing statistics of infrequently queried portions of the base data. Another consequence of the offline nature of these techniques is that the histograms they built are *static* in the sense that, after histograms are built, the histograms remain fixed regardless of any change in the data distribution. To ensure accuracy of the statistics when the base data change significantly, the histograms must be rebuilt by scanning the base data again. This rebuild approach is neither effective nor efficient because of the scanning cost associated with the size of the base data and the complexity associated with evaluating the objective functions that they minimize. Our work overcomes these drawbacks by building and maintaining histograms in a dynamic way based only on query workloads.

LEO by Stillger et al. [15] takes a different approach. It uses the actual selectivity feedback from each operator in the query execution plan to maintain *adjustments* that are used to correct the estimation errors from histograms. Note that LEO does not change the histograms themselves, while our work aims on building and maintaining better histograms using query feedback.

Our Contributions. In this paper we present SASH, a **S**elf-**A**daptive **S**et of **H**istograms, that addresses these three issues simultaneously. SASH is a two-phase method for the online construction and maintenance of a set of histograms for multidimensional queries (see Figure 1). In the online tuning phase, SASH uses the *delta rule* [14] to tune the current set of histograms in response to the estimation error of each query. The estimation error is computed from the true selectivity of a query obtained from the query execution engine, a *query feedback*. In the restructuring phase, SASH searches for a new and more accurate set of histograms to replace the current set of histograms. We extend graphical statistical models to model a set of histograms with memory constraints and search for the best model given a batch of query feedback. The best model found by SASH includes both the optimal set of histograms and the corresponding optimal memory allocation for each histogram. In other words, SASH addresses both the problem of finding the best attribute sets to build histograms on and the problem of finding the best memory distribution (of the given amount of memory) among the histograms. In contrast to previous model search methods [8, 6], SASH does not require access to the database relations to evaluate the candidate models (sets of histograms), but evaluates each candidate using only query feedback information from a query workload. The restructuring phase can be activated periodically or as needed when performance degrades. In summary, our contributions are:

- We develop a new method to build and maintain an optimal set of histograms using only query feedback information from a query workload, without accessing the base data. Because our method is dependent only on query feedback, it is able to adapt to workload and data distribution changes.
- We propose a unified framework that addresses the problem of which attribute sets to build histograms on, the problem of allocating memory to a set of histograms, and the problem of tun-

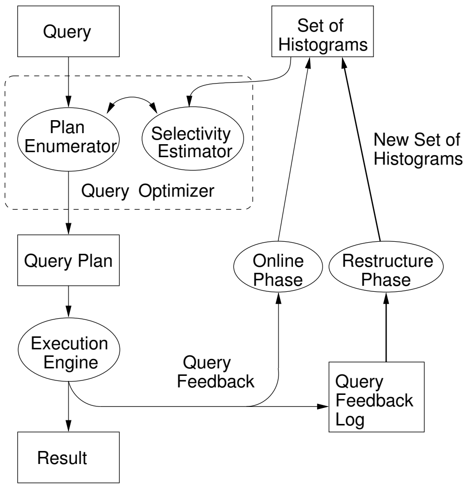


Figure 1: Workflow of SASH. The right path is for query processing and the feedback loop is for workload-driven selectivity estimation processing.

ing a set of histograms to the query workload.

- For a multidimensional query involving attributes spanning several histograms, we show how to perform online updates of the relevant histograms in a principled manner using the delta rule [14].

The rest of this paper is organized as follows. The next section introduces basic notations. We give an overview of SASH in Section 3. We describe the restructuring phase in Section 4 and the online tuning phase in Section 5. Section 6 gives the experimental evaluation of SASH. We draw conclusions and outline future work in Section 7.

2 Preliminaries

A database consists of a set of relations. Each relation R is a set of attributes. Since we are not dealing with joins between relations, we simplify our presentation by considering a single relation R. Without loss of generality, let relation $R = \{A_1, A_2, \ldots, A_k\}$. Each attribute A_i takes real values from the value domain D_i (a discrete set of real numbers). Let D_i be indexed by $\{1, 2, \ldots, |D_i|\}$ and let $D(R)$ denote the domain for a set of attributes in relation R, i.e., $D(R) = D_1 \times D_2 \times \ldots \times D_k$, where each D_i is the domain for attribute $A_i \in R$. The normalized frequency distribution of relation R is denoted by

$$P(R) = \frac{f(R)}{||R||} \tag{2.1}$$

where $f(R)$ is the frequency distribution for the attributes in R and $||R||$ denotes the total number of tuples in relation R. Equation (2.1) allows us to treat frequency distributions as if they are probabilities. The frequency distribution $f(R)$ is a shorthand for

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

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

$$\begin{aligned} f(\mathbf{X} = \mathbf{x}) &= \text{no. of } \langle x_0, x_1, \ldots, x_j, \ldots \rangle \text{ tuples} \\ &\quad\ \text{in relation } R \\ &= \sum_{\mathbf{b} \in D(\mathbf{R} - \mathbf{X})} f(\mathbf{R} = \langle \mathbf{x}, \mathbf{b} \rangle). \end{aligned} \tag{2.2}$$

Let $\mathbf{r}$ be a set of real intervals constraining $\mathbf{X}$. We adopt the shorthand of $\mathbf{X} \in \mathbf{r}$ to represent a range query on the attributes of $\mathbf{X}$. For example, if $\mathbf{X} = \{A_i\}$ and $\mathbf{r} = \{[l, h]\}$, then $\mathbf{X} \in \mathbf{r}$ denotes the range query $l \leq A_i \leq h$. Geometrically, the j intervals in a query range $\mathbf{r}$ form a hyper-rectangle in the j-dimensional domain space $D(\mathbf{X})$. The selectivity of a range query $\mathbf{X} \in \mathbf{r}$ on relation R is therefore the sum of all the j-dimensional points $\mathbf{x} \in D(\mathbf{X})$ that are enclosed by the hyper-rectangle defined by $\mathbf{r}$,

$$\sigma(\mathbf{X} \in \mathbf{r}) = \sum_{\substack{\mathbf{x} \in D(\mathbf{X}) \\ \mathbf{x} \in \mathbf{r}}} f(\mathbf{X} = \mathbf{x}). \tag{2.3}$$

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

A histogram h is an approximation to a frequency distribution. In the following description of SASH and in our experiments, we have use partition-based histograms (and in particular MHIST [13]) in order to make more meaningful comparisons, even though SASH is a general framework that does not assume any specific histogram type. A partition-based histogram is described by the set of attributes *attributes*(h), the number of buckets *nbuckets*(h), the frequency stored in each bucket, and the region of the domain space $D($*attributes*$(h))$ covered by each bucket.

3 Overview of SASH

In this paper, we propose SASH, a general framework aimed at building and maintaining a set of histograms that (1) covers all the attributes of interest, (2) optimizes some performance criteria, (3) fits in a given amount of memory, and (4) require only query feedback information for construction and maintenance. SASH is a two-phase framework to build and maintain a set of histograms. The workflow of our approach is shown Figure 1.

The *restructuring phase* takes as input a batch of query feedback and performs a search for the optimal set of histograms and the corresponding memory allocation for each histogram. The restructuring phase is used to reorganize the set of histograms either periodically or when performance degrades. The restructuring phase can also be used to obtain an initial set of histograms when a batch of query feedback is available[1]. The restructuring phase can be activated when sufficient query feedback has been collected. It is not feasible to run the restructuring phase after each query, because the restructuring phase involves the building of new histograms. The *online tuning phase* tunes the frequencies of a set of histograms obtained from restructuring phase using query feedback in an online manner without changing the structure of the histogram set and the memory allocation among histograms.

We describe the restructuring phase and the online tuning phase in details in the next two sections.

4 The Restructuring Phase

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

Once the type of histograms (e.g., equi-depth, equi-width, MHIST, etc.) is chosen, a set of histograms is characterized by (1) the attribute sets on which histograms are built, (2) the frequency or count stored in each bucket of each histogram, and (3) the amount of memory (or number of buckets) allocated to each histogram in the set. We describe how SASH addresses these factors in this section.

There are two frequently used ways of choosing attribute sets: (a) each set contains a single attribute (also called the attribute-value independence or AVI assumption), and (b) each set contains all the attributes of interest from a single relation (also known as the saturated model[2]). In the first case, one histogram is built for each attribute of interest in the database, and in the latter, one multidimensional histogram is built for each relation in the database. Under tight memory constraints, both ways are error-prone and perform poorly for high-dimensional data with complicated correlations [12, 7, 8, 6].

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

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

4.1 Graphical Models

Graphical models [10] are compact representations of high-dimensional joint data distributions. A graphical model $GM = (S, \Theta)$ consists of a graph $S = (V, E)$ and a set of parameters Θ encoding the associated distributions. The graph S encodes the statistical dependence relationships between attributes and is often called the *structure* of the data. Each vertex represents an attribute and each edge represents a statistical dependence between two attributes. Using the graph S and the associated distributions Θ, we can reconstruct the joint distribution of all the attributes losslessly. Examples of graphical models include Bayesian networks, Markov networks, and junction trees. For the rest of this paper, we will use the junction tree representation; however, the techniques described in this paper can be applied to other types of decomposable graphical models.

In the junction tree representation, the graph structure S is restricted to chordal graphs. An example of a chordal graph and the corresponding junction tree is shown in Figure 2. A *chordal graph* is an undirected graph where, for every simple cycle of more than three vertices, there is some edge that is not involved in the cycle but joins two vertices in the cycle (a chord). A *junction tree (or forest)* $J(S) = (V_J, E_J)$ of a chordal graph S is a tree (or forest) where each node corresponds to a clique (maximal complete subgraph) in S and each edge corresponds to a non-empty intersection between two cliques in S. Each node in a junction tree represents a set of attributes in the chordal graph S. Let the set of attributes associated with a node $u \in V_J$ be denoted by C_u. An edge $(u, v) \in E_J$ exists iff C_u and C_v has non-empty intersection. The set of parameters Θ associated with a junction tree consists of a set of distributions $\{P(C_v) : v \in V_J\}$, one for each node in the junction tree. Each $P(C_v)$ is called a *clique distribution*.

Selectivity Estimation Using a Junction Tree. Given a junction tree model $M = (J(S), \Theta)$ for a set

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

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

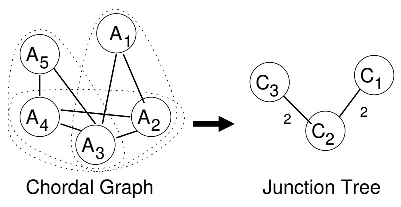


Figure 2: An example of a graphical model for five attributes $\{A_1, A_2, \ldots, A_5\}$. The chordal graph has five nodes corresponding to the five attributes. Each edge in the chordal graph models a statistical dependency between two nodes. The junction tree has three cliques: $C_1 = \{A_1, A_2, A_3\}$, $C_2 = \{A_2, A_3, A_4\}$, and $C_3 = \{A_3, A_4, A_5\}$. The weight of each edge in the junction tree is the number of shared attributes between the end nodes of that edge.

of attributes $R = \{A_1, \ldots, A_k\}$, the joint distribution can be computed as

$$P(\mathbf{R} = \mathbf{a}) = \frac{\prod_{u \in V_J} P(C_u)}{\prod_{(u,v) \in E_J} P(C_u \cap C_v)}. \qquad (4.4)$$

Each of the $P(C_u)$ terms (and the $P(C_u \cap C_v)$ terms) is a function of the attributes in the set C_u (and $C_u \cap C_v$) and each function has to be evaluated using the values in the given point $\mathbf{a} \in D(R)$ to instantiate the attributes in C_u (and in $C_u \cap C_v$). If a junction tree model is a good approximation of the joint distribution of a database, (4.4) can be used for selectivity estimation. In particular, for range queries on a subset of attributes, equations (2.2) and (2.3) can be used with (4.4) to obtain the corresponding selectivity estimates. For a query on a single table R, (2.1) can be used to simplify (4.4) and the selectivity can now be estimated using frequency counts:

$$f(\mathbf{R}) = \frac{\prod_{u \in V_J} f(C_u)}{\prod_{(u,v) \in E_J} f(C_u \cap C_v)}. \qquad (4.5)$$

We have described the junction tree model in the context of a single relation R. Since we do not consider joins, a junction tree/forest model of an entire database is simply a collection of junction trees for each relation. The parameter set Θ for this junction forest model is the collection of frequency distribution for each clique in this forest.

It is impractical to store the frequency distribution for each clique C_u exactly. Instead we use histograms to approximate $f(C_u)$, and we call these histograms *clique histograms*. We use MHIST histograms [12] in our presentation and in our experiments, although other histogram techniques can be used with minor modification.

4.2 Finding the Best Set of Histograms

The joint data distribution of a database can be described using graphical models. A set of histograms

```
Restructuring Phase (Qfb, C)
C : memory constraint
m = (J(S), $\mathcal{H}$, B) : histogram set model

 1   J(S) ← AVI
 2   $\mathcal{H}$ ← one bucket histograms
 3   B ← vector of one's
 4   c ← size_in_bytes(m)

 5   while (c < C) {
 6     for each $m_{struct}$ in S_Candidates(m) {
 7       score($m_{struct}$, Qfb)
       }
 8     $m^*_{struct}$ ← $m_{struct}$ with best score
 9     for each $m_{bucket}$ in B_Candidates(m) {
10       score($m_{bucket}$, Qfb)
       }
11     $m^*_{bucket}$ ← $m_{bucket}$ with best score
12     m ← best scoring model
           between ($m^*_{bucket}$, $m^*_{struct}$)
13     c ← size_in_bytes(m)
     }
```

Figure 3: Search algorithm for the best model. `S_Candidates(m)` refers to the set of candidate models obtained by making a small structural change on `m` and `B_Candidates(m)` refers to the set of candidate models obtained by adding one bucket to some histogram in `m`.

that approximates the joint data distribution can likewise be described using graphical models with memory constraints. In particular, we extend the junction tree representation to describe which sets of attributes to build histograms on.

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

Finding the best set of histograms for a database can now be cast as a search for the histogram set model that maximizes or minimizes some scoring criterion. Our search algorithm is outlined in Figure 3. We initialize the current model m to the simplest model (lines 1-3) and iteratively improve it. The simplest model assumes attribute value independence and builds a one-bucket histogram for each attribute (see Figure 4 for an example). At each iteration of the `while`-loop in line 5, the algorithm explores two options: (1) making the best local change in structure, i.e., the junction tree (lines 6-8), or (2) adding a bucket to a histogram, but keeping the cur-

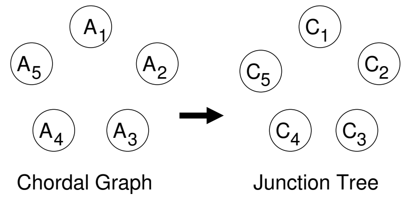


Figure 4: An example of the AVI graphical model for five attributes $\{A_1, A_2, \dots, A_5\}$. The mutual independence of the attributes is represented by the lack of edges in the chordal graph (left). The junction tree (right) reduces to singleton cliques: $C_i = \{A_i\}$ for $i = 1, \dots, 5$.

rent structure (lines 9-11). The option that gives the best scoring model is taken (line 12) and the algorithm proceeds to the next iteration. A local change in structure is defined to be the addition of an edge to the underlying chordal graph S that does not violate the chordal graph property. Computing the set of candidate junction trees that differ from $J(S)$ by one edge in the chordal graph S has been described in [5]. Making a structural change always results in some old clique histograms being discarded and one new clique histogram of higher dimension being created. The buckets of the discarded histograms are assigned to the new histogram and in addition, the algorithm also explores giving more buckets to the new histogram, because it has higher dimensionality than the discarded ones. Since the clique histograms discarded in some iterations of the `for`-loop (line 6) may be needed in future iterations, the discarded histograms should be kept in a cache to avoid recomputation.

At each iteration of the search (line 5), we add buckets to either existing clique histograms or the new one. When the (memory) size of the model reaches the desired threshold, the search algorithm terminates and outputs the best model that it finds. In addition to memory constraints, we also place a bound on the dimensionality of each clique histogram: candidate models with clique size greater than a given size (typically three) are not considered [6].

4.3 Scoring Criteria

In this section we describe some of the scoring functions we have used with the search algorithm outlined in Figure 3. Previous work on selectivity estimation using graphical models [6, 8] does the evaluation of candidate models against the true distribution, the database itself. Since only feedback information for a query workload is available in our case, scoring functions that are based on the Kullback-Liebler divergence from the true distribution cannot be used. Candidate models will have to be evaluated based on the information in the batch of query feedback. Two scoring techniques are outlined next.

Standard Error Measures. A candidate model m' can be evaluated based on its selectivity estimation performance on the queries in the batch of query feedback. Selectivity estimation performance can be measured using a variety of error measures [16]. Two examples are the 2-norm of the absolute error,

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

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

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

Minimum Description Length (MDL). Using the MDL principle [2], we can evaluate a candidate model m' by the number of bits required to encode the selectivity information in the batch of query feedback using m'. Given this encoding, the true selectivities in the batch of query feedback can be losslessly reconstructed. Since m' provides estimates to the true selectivities, we only need to encode the estimation errors and the model m' itself. The decoder can use the model m' to estimate the selectivity of each query in the batch and use the estimation errors to recover the true selectivity. Let $e_1, e_2, \dots, e_N$ be the estimation errors of model m' for the batch of query feedback Qfb (where N is the number of query feedback tuples in Qfb), and let $p(e)$ be the empirical probability distribution of the estimation errors,

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

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

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

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

4.4 Learning Clique Histograms from Query Feedback

We have described our search algorithm for the best histogram set model. In this section we address the problem of building a set of histograms from a batch of query feedback given a fixed model. This problem differs from that addressed in [1, 3] in that we consider multi-attribute queries whose query attributes are not covered by a single histogram, whereas [1, 3] assumes that the attributes of each query is covered by exactly one multidimensional histogram. For example, [1, 3] do not address how to build two one-dimensional histograms for attributes A_1 and A_2 given two dimensional queries on attributes A_1 and A_2. We show how to build low-dimensional histograms from high-dimensional queries using the delta rule [14].

A histogram h that approximates the frequency distribution of a set of attributes $attributes(h)$ consists of a set of buckets $B(h)$ indexed by $\{1, 2, \ldots, |B(h)|\}$. Each bucket i corresponds to a particular partition $box(i)$ of the domain space for $attributes(h)$ and is associated with c_i, the frequency count of tuples that occur in that partition.

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

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

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

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

If the query attributes $\mathbf{X}$ is a proper subset of $attributes(h)$, the attributes that are not in the query are marginalized out in the same way as computing a marginal probability distribution (Equation (2.2)).

Consider the example histogram in Figure 5. The selectivity of two dimensional query Q_2 is

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

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

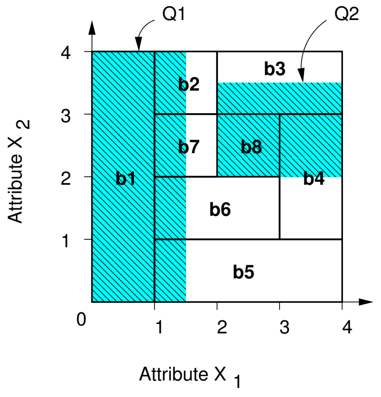


Figure 5: An example of a two dimensional histogram.

The selectivity of one dimensional query Q_1 is

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

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

$$\hat{\sigma}(\mathbf{X}\in\mathbf{r}) = \frac{\prod_{u\in V'}\hat{\sigma}(C_u)}{\prod_{(u,v)\in E'}\hat{\sigma}(C_u\cap C_v)}. \tag{4.6}$$

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

Suppose that the partitioning of the domain space by a clique histogram is fixed and we want to update the frequency count c_b of a particular bucket b of that histogram in response to an estimation error in a query $\mathbf{X}\in\mathbf{r}$. Let σ be the true selectivity of this query and $\hat{\sigma}$ be the selectivity of this query estimated by the model. The frequency for bucket b can appear at most once in the numerator and possibly many times in the denominator (in the following formula l times); hence, the estimated selectivity (Equation (4.6)) can be expressed in terms of c_b as

$$\hat{\sigma} = \frac{\alpha_0 c_b + k_0}{\prod_{i=1}^{l}(\alpha_i c_b + k_i)} \times \rho \tag{4.7}$$

$$= \frac{p_0}{\prod_{i=1}^{l} p_i} \times \rho \tag{4.8}$$

where ρ denotes the rest of the numerators and denominator terms that do not involve the bucket b, each α_j denotes the fraction of overlap of bucket b with the query range, each k_j denotes the rest of the bucket frequencies being summed, and each p_j is a shorthand for $\alpha_j c_b + k_j$. We use the delta rule [14] (gradient descent method) to update the frequency count c_b of bucket b in response to an estimation error $\hat{\sigma} - \sigma$. The delta rule states that for an error function $E(c_b)$, the update to the variable c_b should be proportional to the negative gradient of $E(c_b)$ with respect to c_b. Using the delta rule to minimize the squared error,

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

we update the bucket frequency c_b in proportion to

$$\frac{\partial E(c_b)}{\partial c_b} = 2(\hat{\sigma} - \sigma)\frac{\partial \hat{\sigma}}{\partial c_b} \tag{4.9}$$

where

$$\frac{\partial \hat{\sigma}}{\partial c_b} = \frac{\alpha_0 \hat{\sigma}}{p_0} - \sum_{i=1}^{l} \frac{\alpha_i \hat{\sigma}}{p_i}. \tag{4.10}$$

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

$$\begin{aligned} c_b^{(n+1)} &= c_b^{(n)} - \gamma \frac{\partial E(c_b)}{\partial c_b} \\ &= c_b^{(n)} - 2\gamma(\hat{\sigma} - \sigma)\left[\frac{\alpha_0 \hat{\sigma}}{p_0} - \sum_{i=1}^{l} \frac{\alpha_i \hat{\sigma}}{p_i}\right], \end{aligned} \tag{4.11}$$

where γ is a tunable parameter called the learning rate, each p_0 corresponds to a $\hat{\sigma}(C)$ term (in Equation (4.6)) that uses bucket b and each $p_j, j > 0$ corresponds to a $\hat{\sigma}(C_u \cap C_v)$ term that uses bucket b in (4.6). For a batch of query feedback, we iterate through the batch a fixed number of times, applying the update rule in (4.11) (see Figure 6).

```
Learn_bucket_freq(b_set, Qfb)
b_set : set of buckets
Qfb   : set of query feedback
niter : no. of update iterations

for i=1 to niter do {
  for each q in Qfb involving b_set do {
    for each bucket b in b_set do {
      update b.freq using Equation (4.11)
    }
  }
}
```

Figure 6: Algorithm for updating the frequency of a set of buckets (from the same histogram).

```
Learn_MHIST(h, nB, Qfb)
h   : a one bucket clique histogram
nB  : the target no. of buckets
Qfb : set of query feedback

 1  while (h.nbuckets < nB) {
 2    for each bucket b in h do {
 3      for each dimension d do {
 4        (b',b'') ← split b at the
                       mid point along d
 5        Learn_bucket_freq({b',b''},Qfb)
 6        score(b,d) = score this split
        }
      }
 7    find the (b*,d*) with the best score
 8    apply the split (b*,d*) on h
 9    h.nbuckets++
    }
```

Figure 7: Algorithm for building a MHIST histogram from a batch of query feedback.

The update rule in (4.11) learns the bucket frequencies of a clique histogram assuming that the set of buckets (i.e., the partition) is fixed. To find a suitable partition, we search in a restricted set of partition schemes for the partition that minimizes the squared error (with respect to a batch of query feedback). In the case of MHIST, we always split a bucket into two equal halves. The region covered by a bucket is d-dimensional, and hence there are d possible splits. The algorithm for learning a MHIST histogram (both partitioning and bucket frequencies) from a batch of query feedback is outlined in Figure 7. Basically, in each iteration of the `while`-loop (line 1), we score each candidate split (line 4-6) and choose the best candidate split to apply on the histogram. A candidate split is completely specified by the bucket to split and the dimension to split. Line 4 does not actually perform the split in the histogram, but simulates a split by returning the two buckets that would have resulted from the split.

5 The Online Tuning Phase

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

```
Online Update(⟨J(S), $\mathcal{H}$, B⟩, ⟨q, $\sigma$⟩)
⟨q, $\sigma$⟩ : query feedback
⟨J(S), $\mathcal{H}$, B⟩ : current histogram set model

1   for each h in $\mathcal{H}$ involved in q do {
2     for each bucket in h involved in q do {
3       update b.freq using Equation 4.11
      }
    }
```

Figure 8: Algorithm for online update in response to a query feedback.

current query workload.

The online update algorithm is based on the update rule given in (4.11) and is outlined in Figure 8. Given a query feedback $(q, \sigma(q))$, we update all the histograms that are involved in the computation of the selectivity estimate $\hat{\sigma}(q)$.

The complexity of the online update algorithm is bounded by the size of the set of histograms, which reside in a small constant amount of memory space. The overhead of the online update is therefore not a significant cost compared to the potential improvement in estimation accuracy.

6 Experimental Evaluation

In this section, we describe our experiments and present experimental results. Although we experimented with both synthetic and real data, we only present results for the real data in this paper in the interest of space.

Real Data. We use data from the Current Population Survey (CPS) of the US Census Bureau (www.census.gov) as in [5]. This data set consists of the following attributes from the Person Data Files of the March (2001) Supplement: race, native country of sample person, native country of mother, native country of father, citizenship and age. There are 128821 tuples, of which 13824 are distinct.

Query Generation. Our query workload generator takes as input a pair of normal distribution parameters (μ, σ) for each attribute A_i, where μ is the most frequently queried value of A_i and σ is the standard deviation that controls how skewed the query workload is going to be. For our experiments, each μ is randomly chosen. A query is generated by randomly picking the relation R to be queried, the number of attributes d to be queried, and the d attributes from relation R. For each of the d attributes, the low endpoint is picked randomly according to the normal distribution specified for the attribute. The high endpoint is picked using the uniform distribution from between the low endpoint and the maximum value for that attribute. The true selectivity of the query is evaluated using the actual data distribution. For our experiments, we generated workloads of 5000 queries each using the skew parameters from Table 1. All the queries have positive selectivity.

Comparisons. We compare the performance of five algorithms. **DBHist** is the method proposed in [6] that we extended to optimize the bucket allocation over all the clique histograms in the database. **SASH-avi** assumes attribute value independence and builds one MHIST histogram for each attribute. **SASH-sat** ('saturated') builds one MHIST histogram for each relation. Both **SASH-avi** and **SASH-sat** use our method to learn the MHIST and optimize the bucket allocation using the 2-norm absolute error score except that they assume a fixed structure S. In the case of **SASH-avi**, each clique contains one attribute, and in the case of **SASH-sat**, each clique contains all the attributes in a relation. **SASH-mdl** uses our search algorithm with the MDL scoring function. **SASH-abs2** uses our search algorithm with the 2-norm absolute error scoring criterion. In our implementation of the restructuring phase of the SASH algorithms, we have used a grid histogram to extract information from the query workload first. The `Learn_MHIST` algorithm (Figure 7) then uses the grid histograms to compute the bucket frequencies of the split buckets.

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

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

Absolute errors vary over different datasets. To make the results more meaningful we normalize the {1,2}-norm absolute errors using the largest selectivity in the query workload [16].

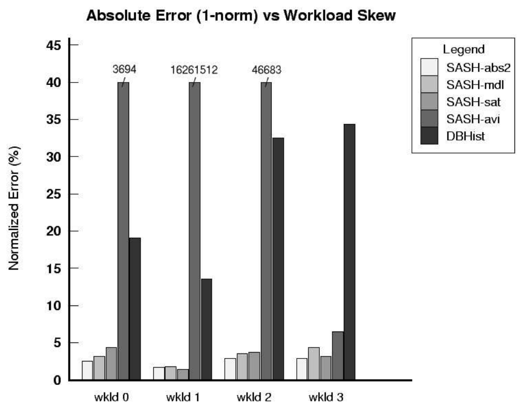
6.1 Restructuring Phase Performance

We evaluate how well the restructuring phase is able to learn the data distribution from a batch of query feedback. For a given query workload, we run the restructuring phase and measure the estimation errors for that given workload when the restructuring phase has completed.

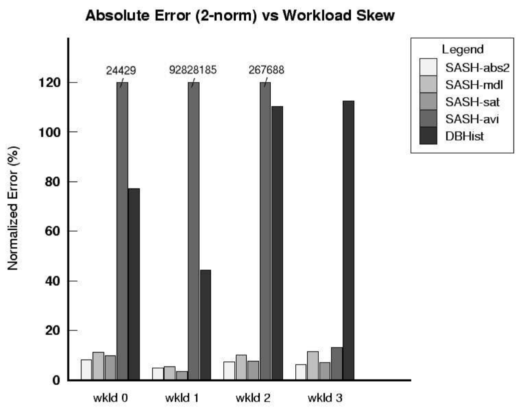
Varying workload skew. We vary the skew levels (the standard deviation parameter in workload generation) of the workload to determine the performance of the restructuring phase over workloads of different skew levels. Workload 0 has query ranges drawn uniformly from the domain space. Workloads 1, 2, and 3 have query ranges drawn using a Gaussian distribution with decreasing standard deviation

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

Table 1: The skew parameters (standard deviation) used for generating the four multidimensional query workloads for the restructuring phase.



(a) 1-norm



(b) 2-norm

Figure 9: Normalized absolute error performance for four query workloads with increasing workload skew. All methods are given 1500 bytes of memory. Workload 0 has queries drawn uniformly from the census data and workload 1, 2, and 3 have queries that are increasingly skewed.

(hence increasing skew). Figure 9 shows the performance of SASH with 1500 bytes of memory over the four different workloads. SASH is consistently more accurate than the offline **DBHist** over the different workload skew levels. Moreover, **SASH-abs2** is always better than using the AVI assumption (**SASH-avi**).

Varying memory constraint. For workload 3, Figure 10 shows the performance of SASH as the memory constraint is varied. SASH is consistently better than **DBHist** regardless of memory constraints.

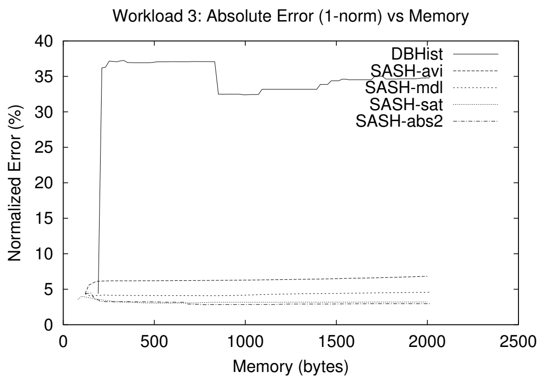
6.2 Online Tuning Phase Performance.

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

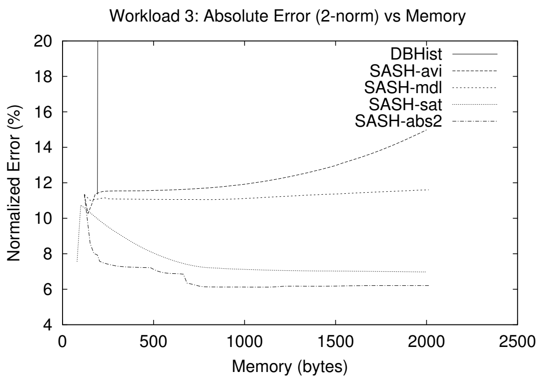
$$similarity(Q_1, Q_2) = \frac{vol(MBR(Q_1) \cap MBR(Q_2))}{vol(MBR(Q_1) \cup MBR(Q_2))}.$$

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

We run the online tuning phase using the histogram sets that were obtained by running the restructuring phase on workload 2 of our restructuring phase experiments. All the histogram sets are roughly 2000 bytes in size. The learning rate used is 0.0001. Three online workloads of 5000 queries each were generated with the same skew parameters as the training workload. The similarity scores of the online workloads with respect to the training workload are 66% for workload 1, 89% for workload 2, and 91% workload 3. We measure the 1-norm of the absolute errors (average absolute errors) over the *entire* online workload as SASH performs online tuning in response to the estimation error of

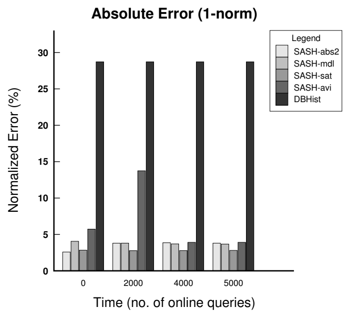


(a) 1-norm

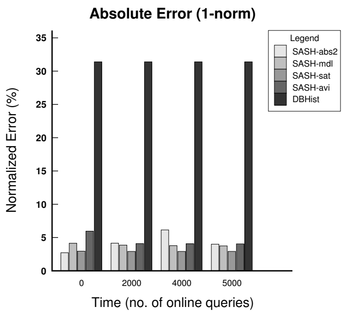


(b) 2-norm (zoomed in)

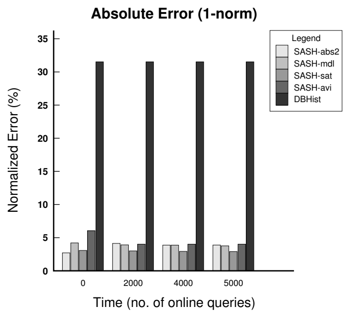
Figure 10: Normalized absolute error performance on workload 3 for different memory constraints. Note that the vertical ordering of labels in the legend is the same as the vertical ordering of the curves for memory greater than 200 bytes. We have truncated the **DBHist** curve and zoomed in on the SASH curves in the 2-norm plot to show the relative performance of the SASH methods.



(a) Online workload 1 (66% similar)



(b) Online workload 2 (89% similar)



(c) Online workload 3 (91% similar)

Figure 11: Online performance of SASH for three online multidimensional query workloads that differ in varying degrees from the workload used in the restructuring phase.

each query in the online workload. Figure 11 shows the online performance of SASH for four time slices. Time slice 0 gives the performance before any online tuning has been performed. Time slices 2000, 4000, and 5000 give the performance after the online tuning phase has seen 2000, 4000, and 5000 queries respectively. Note that since **DBHist** is static, its online performance does not improve with time. SASH consistently outperforms **DBHist** over the different online workloads. We have found that the histogram sets obtained from **SASH-mdl** restructuring phase tend to perform better than those from **SASH-abs2** during online tuning. The online performance of **SASH-abs2** tends to fluctuate more (as evidenced in Figure 11) compared to **SASH-mdl**. **SASH-mdl** seems to be better at capturing the correlations between attributes. Once correlations are sufficiently captured in the multidimensional histograms, online tuning is very effective in tuning the histograms to decrease the overall estimation errors (see Figure 12).

7 Conclusion

We have proposed SASH as a novel two-phase method that builds and maintains an optimal set of histograms using only query feedback information from a multidimensional query workload, without scanning the database. For queries involving attribute sets that span several histograms, we have shown how to perform online updates of the relevant histograms in a principled manner using the delta rule. SASH has also provided a unified frame-

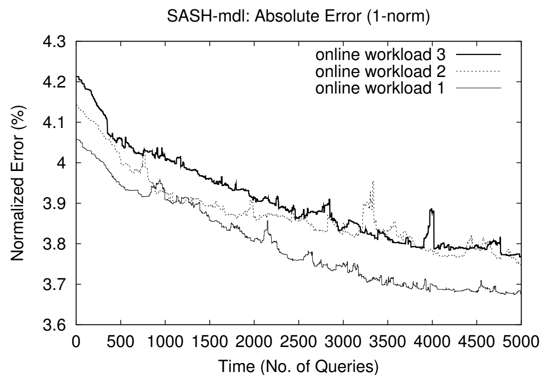


Figure 12: Online performance of **SASH-mdl** for the three online multidimensional query workloads. Online tuning is very effective for decreasing the overall errors, when correlations are adequately captured by the multidimensional histograms.

work that addresses the problem of which attribute sets to build histograms on, the problem of allocating memory to a set of histograms, and the problem of tuning a set of histograms to the query workload.

SASH has exposed many interesting issues. We have used one type of histograms in this paper (namely, MHIST) to facilitate comparisons. As future work, we plan to explore the use of other histogram techniques (such as wavelet histograms) in SASH. The restructuring phase currently finds a new set of histograms from scratch. We plan to investigate how to incorporate the information in the current set of histograms when doing restructuring. In this paper, we have also used a homogeneous set of histograms, i.e., all the histograms are of the same type. Since some histogram types are more suited to certain distributions than others, another direction for future work is to use a set of histograms that need not all be of the same type. SASH would need to optimize the heterogeneous set of histograms over the space of histogram types as well.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, 181–192, 1999.

[2] A. Barron, J. Rissanen, and B. Yu. The minimum description length principle in coding and modeling. *IEEE Trans. Information Theory*, 44(6), 2743–2760, 1998.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 211–222, 2001.

[4] C. M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, 161–172, 1994.

[5] A. Deshpande, M. Garofalakis, and M. I. Jordan. Efficient stepwise selection in decomposable models. In *Uncertainty in Artificial Intelligence: Proceedings of the 17th Conference (UAI-2001)*, 128–135. Morgan Kaufmann Publishers, 2001.

[6] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 199–210, 2001.

[7] C. Faloutsos and I. Kamel. Relaxing the uniformity and independence assumptions using the concept of fractal dimension. *Journal of Computer and System Sciences JCSS*, 55(2), 229–240, 1997.

[8] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 461–472, 2001.

[9] H. V. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 223–234, 2001.

[10] M. I. Jordan, editor. *Learning in Graphical Models.* MIT Press, 1999.

[11] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, 448–459, 1998.

[12] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of 23th Intl. Conf. on Very Large Data Bases*, 486–495. Morgan Kaufmann, 1997.

[13] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, 294–305, 1996.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, chapter 8, 318–362. MIT Press, 1986.

[15] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO – DB2 LEarning Optimizer. In *Proceedings of 27th Intl. Conf. on Very Large Data Bases*, 19–28. Morgan Kaufmann, 2001.

[16] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, 193–204, 1999.