Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Query Size Estimation using Machine Learning

Banchong Harangsri John Shepherd Anne Ngu

School of Computer Science and Engineering,
The University of New South Wales, Sydney 2052, AUSTRALIA.
Telephone: +61 2 9385 3980 Fax: +61 2 9385 1813
Email: {bjtong,jas,anne}@cse.unsw.edu.au

Abstract

We propose two novel notions in this paper: the first is that machine learning techniques can be used to solve the problem of query size estimation and the second is a new generic algorithm to correct the training set of queries in response to updates. The main advantage for machine learning is that no database scan is required to collect statistics for query size estimation. The training set correction algorithm is useful in that it allows us to "re-vitalise" some existing query size estimation methods whose performance previously deteriorated in the presence of high update loads. A by-product of this is that the length of training sets can be fixed – the size of the training set determines the level of error in query estimation. Our experimental results show that (1) the machine learning technique is superior to a recent curve fitting method in approximating query result sizes and (2) the machine learning technique still performs as well after the correction algorithm is applied.

Keywords Query Size Estimation, Query Optimisation, Machine Learning

1 Introduction

Query optimisers for database systems aim to determine the most efficient query execution plan to be executed by the database system. Choosing an efficient plan relies on cost estimates derived from the statistics maintained by the underlying database system. Work by [10] pointed out that inaccurate estimates derived from such statistics may cause the optimiser to choose a very poor plan. Although the initial error might be negligible for the first subplan (such as the first join/selection), the subsequent errors (errors in the next subplans) can grow very rapidly (i.e., exponentially). Good estimates for the cost of database operations are thus critical to the effective operation of query optimisers and ultimately of the database systems that rely on them. This paper proposes a novel method to improve such cost estimation.

There has been a considerable amount of work on the issue of selectivity estimation over one and a half decades [19, 5, 6, 16, 11, 9, 14, 15, 13, 7, 20, 4]. This previous work can be classified into four categories [20, 4], namely *non-parametric, parametric, sampling* and *curve fitting.* Let us briefly describe each of them; the reader can find more details in the references given above.

The non-parametric method is table- or histogram-based [16, 15]. A histogram is built by dividing an attribute domain into intervals and counting the number of tuples which fall into the ranges of the intervals. The method requires scanning an entire relation to build up the histogram and the performance (the accuracy of the size estimation) will not be satisfactory if the number of intervals used is too small.

The parametric method [19, 5, 6] is one which depends upon underlying assumptions about the data distribution (e.g. uniform, normal, Poisson, Zipf, etc.). The method will give accurate query size estimates if the actual data distribution follows the *a priori* assumption. In reality, data distributions in real databases may not fit well with the assumptions and, consequently, the quality of the size estimates could be unreliable.

The sampling method [13, 7] has recently received considerable interest. The accuracy of this method depends upon the size of samples; the higher the sample size, the better the estimation. Given complex queries which consist of several selection and join operations, the method may require a nontrivial amount of time to do a number of samplings (perhaps one for each operation). Compared with other estimation methods which require no extra delay for the samplings, this could be a significant disadvantage.

The curve-fitting method [20, 4] is based on polynomial regression to find the best-fit set of coefficients to minimise the criterion of least-squared error.

The curve fitting method proposed by [20], scans entire relations and uses regression to determine the

distributions of attribute values in each relation. This approach is effective only for low-update database systems. That is, as long as the distributions of attribute values remain fixed, the method performs satisfactorily. However, if the distributions change considerably, then the quality of the size estimates may deteriorate significantly.

The curve-fitting method proposed by [4] uses query feedback to construct cost estimation functions. It uses queries of the form $low \leq a1 \leq high$, where $a1$ is an attribute, and the result sizes of the queries as the basis for regression. An advantage of this method over others mentioned above is that it requires no scan over the database to build up statistics. As more and more queries have been processed and their feedback becomes available to perform regression, the method will give more accurate query size estimation.

However, this second curve fitting method, *adaptive selectivity estimation* (or ASE) has problems in dealing with updates. The method uses "fading weights" to gradually reduce the significance of old query feedback in query size estimation. However, fine-tuning for the best set of fading weights is a difficult optimisation problem.

The method that we propose in this paper aims to overcome most of the difficulties mentioned above. Our overall approach is to derive size estimation functions using machine learning techniques. Specifically, we proceed as follows:

1. use feedback from a training set of queries to construct a model tree (or regression tree) [3],
2. when we need to estimate the result size of a given query, determine three most similar queries to the given query from the training set.
3. approximate the result size of the given query by using the model tree and the result sizes of the most similar queries.

The following advantages are common to both ASE and our method:

- **no relation scan:** We do not need to scan relations to collect statistics on which to base query size estimation. All of the methods above, except the parametric method, require scanning of relations.
- **adaptiveness:** The estimation accuracy improves as more and more queries have been processed and stored in the training set. In the ASE method, extra query feedback assists in adjusting the data distribution curve to better fit the actual distribution of attribute values. In our method, the extra feedback is used to assist in better picking up the three most similar queries.

  However, in the presence of very high loads of updates, ASE and our method use different approaches to maintain size estimation accuracy. We believe that our approach is more effective, and more widely applicable than the approach used by ASE. Except for the sampling method, the other schemes lose their accuracy as the database changes.

Our method has the following advantages over the ASE method:

- **generic algorithm for updates:** The algorithm we give in [8] with some slight modification can also be used with the size estimation methods proposed in [20, 4]. In other words, some size estimation methods proposed for retrieval-only or retrieval-intensive environments can be adapted for use with databases with high loads of updates.

  Given a list of records affected by updates (either inserts or deletes), the algorithm can correct:

  - The distinct-value-frequency list $(x_i, f(x_i))$, where x_i is a distinct value in an attribute domain and $f(x_i)$ is the frequency of x_i before the original size estimation method can be applied.
  - The query feedback list (l_i, h_i, s_i), where l_i and h_i are lower and upper bounds of the ith query (such as $l_i \leq b1 \leq h_i$) and s_i is the result size of this query. After the feedback list has been corrected, then the original size estimation method can be applied.

  This ensures that the lists always reflect the frequency $f(x_i)$ and size s_i. Obviously, assuming that the original size estimation methods are accurate in query size estimation, the additional approach to deal with updates proposed in [4] would be only ad-hoc and not necessarily as effective as the original size estimation method.

- **static list:** Since our algorithm for correcting lists of query feedback and distinct-value-frequency makes the current lists always up-to-date, the length of the lists would not necessarily be extended. In other words, after the length of lists has reached a certain size and the error in query size estimation has dropped to a satisfactory level, then the length remains constant. The approach to deal with updates in [4] combines the

outdated and up-to-date list of query feedback[1], and thus requires the old list to be retained.

This paper is structured as follows:

- **notations and definitions (section 2):** We describe some notation which appears throughout the paper and clarify some frequently used terms.
- **size estimation with retrieval queries (section 3):** This section establishes a framework for using machine learning techniques for the query size estimation problem.
- **size estimation with updates:** This section provides the background of the relationship between queries and frequency distribution of attribute values before we proceed to give an algorithm to correct the list of queries in a training set.
- **experimental results:** This section demonstrates the performance of our method and compares it to the ASE method.
- **conclusions:** We give some final remarks and address some issues for future investigation.

Due to the space limit, the sections on size estimation with updates and experimental results are not given here; we refer the reader to the complete version of the paper in [8].

2 Notations and definitions

The following notations and terminology either appears throughout the paper or needs clarification:

simple query q_i A selection query on a single attribute of the form: b *relopt* x, where b is an attribute of relation R and *relopt* is one of the relational operators in $\{<, >, =, \neq\}$. Note that these four relational operators are sufficient to cover other types of simple queries, such as $b \leq x$, $b \geq x$ or $low < b < high$. For example, $b \leq x$ can be replaced by $(b < x) \cup (b = x)$. Similarly, $b \geq x$ can be replaced by $(b > x) \cup (b = x)$. And $low < b < high$ can be replaced by $(b < high) \cap (b > low)$.

attribute The simple query above consists of 3 attributes, b being counted as 1st attribute, *relopt* as 2nd attribute, and x as 3rd attribute. The term *attribute* will occasionally be used with its normal meaning "attribute of a relation". The intended meaning is always clear from the context.

continuous attribute The simple query above has constant x which would generally be either numeric or string. Numeric values have a natural order. For string values, we sort all strings in the training set in ascending lexicographic order, and assign a rank (i.e. 1,2,...) to each string to convert them to numeric values.

discrete attribute The simple query above has b and *relopt* as discrete attributes. Discrete attributes have no natural ordering on their possible values. In other words, we cannot say attribute $b1$ of relation R is less than or more than attribute $b2$ of the same relation. Likewise, we cannot say that relational operator $<$ is less or more than relational operator $>$.

training set of queries Q A set of simple queries which have been collected from previous user-submitted queries[2] and their result sizes; namely, each query q_i in the set is of the form:

$$q_i : \quad (b, relopt, x_{scaled}, S_{q_i})$$

where S_{q_i} is the result size of this query and x_{scaled} is an x value linearly scaled by:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

where x_{min} is the minimum value in the domain of attribute b and x_{max} the maximum value in the same attribute domain. Note that only values of the continuous attribute will be scaled. Thus, for any simple query, only the third attribute x is continuous (while the other two attributes b and *relopt* are discrete) so its x value will be scaled by the formula above .

unseen query q_u A simple query whose result size we want to estimate. After this unseen query has been processed by the database system, it may be added to the training set of queries.

sd(Q) This is the standard deviation of the result sizes of queries in the training set Q computed by:

$$sd(Q) = \sqrt{\frac{\sum_{i=1}^{|Q|}(S_{q_i} - \bar{S})}{|Q| - 1}}$$

where $\bar{S} = \frac{\sum_{i=1}^{|Q|} S_{q_i}}{|Q|}$.

[1] and use the fading weights to reduce the effect of the outdated feedback in query size estimation

[2] The user-submitted queries may have contained other subqueries such as join queries; we assume here that there are ways to extract only subquery b *relopt* x out of its entire query.

3 Size estimation with retrieval queries

In this section, we describe a machine learning technique to solve the query size estimation problem. This technique was originally proposed by Quinlan [18] as a combination of *model-based learning* and *instance-based learning*. Quinlan's experimental work suggested that the *model tree* approach to model-based learning was the most effective. A model tree is a *regression tree* and originated in work by Breiman et. al. [3]. Instance-based learning herein on which we are based was originally proposed in [12, 1, 2]. The instance-based learning technique can do either a classification or regression task while the model-tree learning technique does only the regression task.

To approximate the size of an unseen query, there are two main steps. First, a model tree is constructed through a training set of queries Q. Second, using (1) the model tree and (2) three queries picked up from the training set which are of the most similarity to the unseen query, the result size of the unseen query can be approximated. The details of each step are described in Sections 3.1 and 3.2, respectively.

3.1 Constructing model tree (leaf node functions)

Given in Figure 1 is the **Partition** algorithm to construct a model tree — the tree with linear regression functions in its leaf nodes. The algorithm was implemented to suit our own use in the query size estimation problem. Two major differences are that the version we implemented has no pruning procedure and no smoothing procedure. Furthermore, there may be other different internal fine-tunings between our version and the original version [18] such as the minimum number of queries in leaf nodes, the minimum error reduction to suppress the recursive partitioning, etc.

The idea to construct a model tree is similar to growing a decision tree by C4.5 [17]. The difference is that the latter is based on maximising information gain while the former is based on minimising intra-subset variation of class values, i.e., query result sizes in our case. The minimisation of the intra-subset variation in the algorithm (see lines 13 and 25) is implemented by:

$$\sum_{i=1}^{partition} \frac{|Q_i|}{|Q|} * sd(Q_i)$$

where Q is a query set and $Q_i, i = 1..partition$ is each query subset of Q. The fundamental rationale behind in doing that is that the query result sizes in each partitioned query subset Q_i will be most similar to one another; in other words, the variation of the sizes in the same query subset will be small.

The algorithm recursively partitions the training set of queries Q into query subsets $Q_1, Q_2, \ldots, Q_{partition}$ (see the details of the algorithm in Figure 1). The recursion stops in 2 cases (see lines 1 and 33). The first case is when there are not enough a number of queries in Q[3]. The second is when the standard deviation of result sizes of queries in Q is too low. In other words, the variation of the result sizes in each partitioned query subset Q_i is similar to one another, i.e.,

$$\left(\frac{sd(Q) - \sum_{i=1}^{partition} \frac{|Q_i|}{|Q|} * sd(Q_i)}{sd(Q)}\right) * 100 < 1.0e^{10^{-2}}$$

The partitioning of Q into its query subsets depends upon whether the current ith attribute is continuous or discrete. If continuous, then do a binary partitioning on Q into Q_1 and Q_2 (see line 11). If not, then do a multi-way partitioning on Q into $Q_1, Q_2, \ldots, Q_d$ (see line 21), where d is the number of distinct values of ith attribute in query set Q.

Figure 2 shows an example of a model tree constructed by the **Partition** algorithm. Query subsets after the algorithm has terminated reside in the leaf (rightmost) nodes of the tree. The labels **A**, **B**, **C**, ..., **L** show all the leaf nodes of the tree.

Here is the description of how the model tree in Figure 2 was built. Recall that query set Q contains queries of the form b *relopt* x. Starting from the root node, entire query set Q was first multi-way partitioned (multi-way partitioning) into 4 query subsets, i.e., the query subset with "=" only as its relational operator, the query subset with "<" only as its operator, and so on. In the next level (after the relational operator level), those 4 query subsets were then binary partitioned (binary partitioning) on their constant values, i.e., values of x. For instance, after the "<" level, at node **E** the constant values of b1 must be less than or equal to 0.27 while at node **F** those of b1 must be more than 0.27. It's possible that any query subset at this stage can still be recursively partitioned further until the two stopping conditions of the algorithm become true. For example, at node $b1 \leq 0.36$ the query subset at this node was binary partitioned into 2 query subsets – the subsets with $b1 \leq 0.09$ and with $b1 > 0.09$.

3.2 Estimating query result sizes

3.2.1 Result size estimation function in leaf node

Suppose we have an unseen query:

$$q_u : \quad b1, =, 0.29$$

[3] In our implementation, we use 60 as a minimum number of queries in a leaf node.

```
1   if |Q| < 60 then
2     return
3   endif
4   Q_best = ∅
5   δ_best = −∞
6   partition = 0
7   for each attribute i ∈ {1...n} do
8     if attribute i is continuous then
9       Q = sort Q on ith attribute's values in ascending order
10      for each sorted value v of attribute i in Q do
11        partition Q into Q_1 and Q_2 where all values of attribute i on Q_1 ≤ v and on Q_2 > v
12        compute an expected error reduction δ:
13              δ = sd(Q) − Σ_{i=1}^{2} (|Q_i|/|Q|) * sd(Q_i)
14        if δ > δ_best then
15          Q_best = {Q_1, Q_2}
16          δ_best = δ
17          partition = 2
18        endif
19      endfor
20    else
21      partition Q into Q_1, Q_2, ..., Q_d where d is the number of distinct values
22         of attribute i; namely, in each query subset Q_i after partitioning,
23         the values of attribute i now will be the same
24      compute an expected error reduction δ:
25            δ = sd(Q) − Σ_{i=1}^{d} (|Q_i|/|Q|) * sd(Q_i)
26      if δ > δ_best then
27        Q_best = {Q_1, Q_2, ..., Q_d}
28        δ_best = δ
29        partition = d
30      endif
31    endif
32  endfor
33  if ((sd(Q) − Σ_{i=1}^{partition} (|Q_i|/|Q|) * sd(Q_i)) / sd(Q)) * 100 < 1.0e^{10^{-2}} where each Q_i ∈ Q_best then
34    return
35  endif
36  for each query subset Q_i ∈ Q_best do
37    Partition(Q_i)
38  endfor
```

Figure 1: **Algorithm** Partition (Q)

and we want to estimate its result size. The query will be parsed down the constructed model tree towards a leaf node. Shown in Figure 2, the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ terminated in the oval leaf node **C** is the one through which the query q_u traverses. The traversal proceeds as follows: path 1 stems from the fact that this model tree is for attribute b1 and the query has "=" as its relational operator, path 2 is due to the fact that the constant value 0.29 of b1 is less than 0.36, path 3 is due to the fact that the constant 0.29 of b1 is more than 0.09 and path 4 is due to the fact that the constant 0.29 of b1 is more than 0.27.

The oval and other leaf nodes with their own query subsets Q_i have their own query size estimation (linear regression) functions; namely, each leaf node has:

$$M(q) = \beta_0 + \beta_1 a_1 + \beta_2 a_2 + \ldots + \beta_n a_n \qquad (1)$$

where $a_i, i = 1..n$ takes on either (1) a value of ith attribute if the ith attribute is continuous or (2) a rank[4] of this value if the ith attribute is discrete. The value of n in the case of simple queries (b *relopt* x) is 3, b being counted as the 1st attribute, *relopt* as the 2nd attribute, and x as the 3rd attribute. $\beta_i, i = 0..n$ is a coefficient of the *least square error* found by minimising the least square error of the estimated and

[4]Here is how to give the ranks of a discrete attribute. Supposing in the training set Q, all the values of the second attribute (containing relational operators) are either of $\{=, <, >, \neq\}$, then the respective ranks of each of those values can be given by: $\{1, 2, 3, 4\}$.

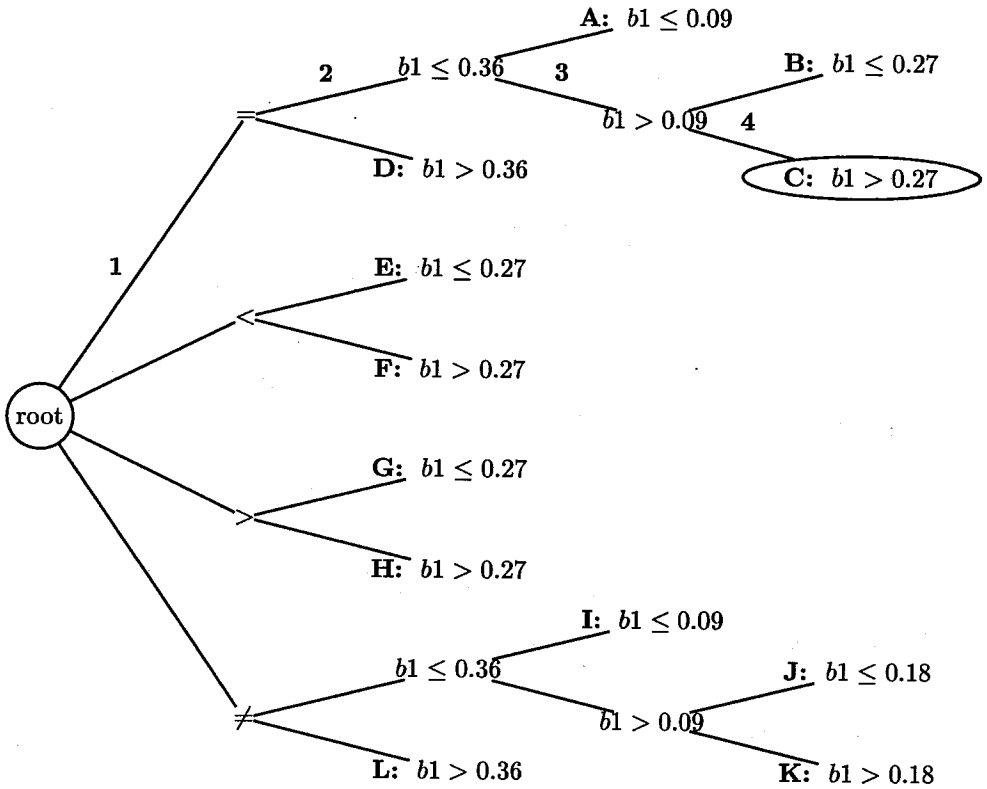

Figure 2: A model tree constructed by the algorithm **Partition**

actual values, i.e.,:

$$\sum_{j=1}^{|Q_i|}(M(q_j) - S_{q_j})^2$$

where S_{q_j} is the actual result size of the jth query in query subset Q_i.

In summary, after parsing a query q_u to a leaf node, its estimated result size can be approximated by the function in equation 1 in that leaf node.

3.2.2 Choosing most similar queries

We compute a similarity value *simval* between the unseen query q_u and a query q_j in the training set Q by the algorithm shown in Figure 3.

Three queries from the training set Q which have the highest similarity values will be chosen as the most similar to the unseen query. The reason in choosing 3 queries instead of other numbers can be described as follows:

- Choosing any number has a tradeoff between bias and variance (see the CART book [3]); namely, higher numbers have higher bias but lower variance.
- Generally, the instance-based learning or KNN (K Nearest Neighbor) methods avoid even numbers, since that is more likely to lead to ties. Using 1 neighbour (query) is generally considered to have too high variance.

3.2.3 Combining leaf node functions and most similar queries

We then adjust the actual result sizes S_{q_i} of the three queries of the most similarity to the unseen query q_u by:

$$\tilde{S}_{q_i} = S_{q_i} - (M(q_i) - M(q_u)) \qquad ; i = 1, 2, 3$$

prior to combining them to produce the estimated value $\hat{S}_{q_u}$ of the result size of query q_u. The combination of the adjusted values $\tilde{S}_{q_i}$ is done by:

$$\hat{S}_{q_u} = \sum_{i=1}^{3} \tilde{S}_{q_i} * w_{q_i} \qquad ; w_{q_i} = \frac{simval_{q_i}}{\sum_{i=1}^{3} simval_{q_i}}. \quad (2)$$

At this point, it is valid to ask:

- Why don't we use $M(q_u)$ in equation 1 as the query size estimate $\hat{S}_{q_u}$?

input: unseen query q_u and query q_j in the training set Q
output: *simval* similarity value of query q_j to q_u

distance = 0
for each attribute $i \in \{1 \ldots n\}$ **do**
  if attribute i is discrete **then**
    if ith attribute's value of the unseen query q_u is different from that of query q_j **then**
      distance = *distance* + 1
    endif
  else
    Δ = the difference between ith attribute's values of the unseen query and query q_j
    distance = *distance* + Δ^2
  endif
endfor
$simval = \frac{1}{\sqrt{distance}}$

Figure 3: **Algorithm** Compute_Simval(Q)

- Why don't we use:

$$\hat{S}_{q_u} = \sum_{i=1}^{3} S_{q_i} * w_{q_i} \qquad ; \; w_{q_i} = \frac{simval_{q_i}}{\sum_{i=1}^{3} simval_{q_i}}.$$

  as the query size estimate $\hat{S}_{q_u}$?

The answer is twofold:

- The author described in [18] that equation $M(q)$ allows taking into account the difference between the unseen query q_u and a most similar query q_i, i.e.:

$$M(q_i) - M(q_u)$$

  and if the equation $M(q)$ is correct, then the adjusted value ($\tilde{S}_{q_i}$):

$$S_{q_i} - (M(q_i) - M(q_u))$$

  should be a better value in favour of the unseen query than the quantity S_{q_i} alone. However, since each most similar query q_i is not the unseen query q_u itself, the combination of their adjusted values in equation 2 based on their weights (this is the main principle of KNN) would produce a good estimate for the size of the unseen query.

- The second answer is pragmatic: both the experiments in [18] and our own experiments show that equation 2 yields the best results.

4 Conclusions

The following is what we have achieved in this paper:

- We have proposed a machine learning technique to solve the problem of query size estimation. The learning machine M5 has demonstrated its superior performance to the ASE method (see the results in [8]).

- In [8] (the complete version of this paper), we have given a novel generic algorithm to correct the distinct-value-frequency list $(x_i, f(x_i))$ [20], the query feedback list (l_i, h_i, s_i) [4] and our training set of queries. This algorithm revalidates the original query size estimation methods in [20, 4] which otherwise will be invalid, i.e., poor in query size estimation in the presence of very high loads of updates.

The following is what we plan to do:

- extend the current machine learning technique to deal with join queries.

- perform more experiments to demonstrate the performance of M5 in approximating sizes of more complicated selection queries such as ones specifying on more than one attribute.

Acknowledgements

We appreciate Dr. Andrew Taylor for pointing us to the learning machine M5. Zijian Zheng helped us a lot at the beginning of this project to enable us to understand how M5 constructs a model tree. Lastly, the very nice line numbering in all the algorithms appearing in the paper comes from the great help of Stephan Böttcher. Stephan spent his valuable time modifying his original package `lineno.sty` to enable the line numbering in figure environment, while running out of time in writing his PhD dissertation.

References

[1] D. W. Aha. *A Study of Instance-Based Algorithms for Supervised Learning Tasks: Mathematical, Empirical, and Psychological Evaluations.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA 92717, Nov 27 1990.

[2] D. W. Aha, D. Kibler, and M. K. Albert. Instance-Based Learning Algorithms. *Machine Learning*, 6(1):37–66, 1991.

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Tress.* Chapman & Hall, Inc., 1984.

[4] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation using Query Feedback. In *Proceedings of 1994 ACM-SIGMOD International Conference on Management of Data*, 1994.

[5] S. Christodoulakis. Estimating Block Transfers and Join Sizes. In *Proceedings of the ACM SIGMOD Conference*, pages 40–54, 1983.

[6] S. Christodoulakis. Estimating Record Selectivities. *Information System*, 8(2):105–115, 1983.

[7] P. Haas and A. Swami. Sequential Sampling Procedures for Query Size Estimation. In *ACM SIGMOD Conference on the Management of Data*, pages 341–350, 1992.

[8] B. Harangsri, J. Shepherd, and A. Ngu. Query Size Estimation using Machine Learning. Technical report, The University of New South Wales, School of Computer Science and Engineering, Sydney 2052, AUSTRALIA, 1996.

[9] W. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 276–287, 1988.

[10] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the ACM-SIGMOD Intl. Conf. on Management of Data*, pages 268–277, 1991.

[11] N. Kamel and R. King. A Method of Data Distribution Based on Texture Analysis. In *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–325, 1985.

[12] D. Kibler, D. W. Aha, and M. K. Albert. Instance-Based Prediction of Real-Valued Attributes. *Computational Intelligence*, 5:51–57, 1989.

[13] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical Selectivity Estimation through Adaptive Sampling. In *Proceedings of ACM SIGMOD*, pages 1–12, 1990.

[14] M. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221, september 1988.

[15] M. Muralikrishma and D. DeWitt. Equi-depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 28–36, 1988.

[16] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of the ACM SIGMOD Conference*, pages 256–276, 1984. Boston, Mass, June, ACM, New York.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, California, 1993.

[18] J. R. Quinlan. Combining Instance-Based and Model-Based Learning. In *Proceedings of Machine Learning.* Morgan Kaufmann, 1993.

[19] 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 *ACM SIGMOD*, pages 23–34, 1979. Boston, MA, June 1979.

[20] W. Sun, Y. Ling, N. Rishe, and Y. Deng. An Instant and Accurate Size Estimation Method for Joins and Selection in a Retrieval-Intensive Environment. In *Proceedings of ACM SIGMOD*, pages 79–88, 1993.