A Piggyback Method to Collect Statistics for Query Optimization in Database Management Systems*

Qiang Zhu† Brian Dunkel‡ Nandit Soparkar‡
Suyun Chen§ Berni Schiefer§ Tony Lai§

Abstract

A database management system (DBMS) usually performs query optimization based on statistical information about data in the underlying database. Out-of-date statistics may lead to inefficient query processing in the system. Existing solutions to this problem have some drawbacks such as heavy administrative burden, high system load, and tardy updates. To overcome these drawbacks, our new approach, called the piggyback method, is proposed in this paper. The key idea is to piggyback some additional retrievals during the processing of a user query in order to collect more up-to-date statistics. The collected statistics are used to optimize the processing of subsequent queries. To specify the piggybacked queries, basic piggybacking operators are defined in this paper. Using the operators, several types of piggybacking such as vertical, horizontal, mixed vertical and horizontal, and multi-query piggybacking are introduced. Statistics that can be obtained from different access methods by applying piggyback analysis during query processing are also studied. In order to meet users' different requirements for the associated overhead, several piggybacking levels are suggested. Other related issues including initial statistics, piggybacking time, and parallelism are discussed. Our analysis shows that the piggyback method is promising in improving the quality of query optimization in a DBMS as well as in reducing the user's administrative burden for maintaining an efficient DBMS.

Keywords: Database management system, query optimization, cost estimation, statistics collection, piggybacking query, access method

1 Introduction

It is well-known that query optimization is crucial in achieving efficient query processing for a *database management system* (DBMS), especially for the relational, object-oriented, and distributed DBMSs[8, 18, 20, 24, 26]. There are two types of query optimizers in DBMSs: heuristic-based and cost-based. Most query optimizers in commercial DBMSs are cost-based, i.e., based on cost analysis for queries[9, 12, 14, 15, 19]. The more accurate the cost analysis is, the more efficient the query processing. However, accurate cost analysis requires the statistics about the underlying database to be up-to-date. Unfortunately, existing methods for collecting and maintaining statistics in the system catalog are not entirely satisfactory.

*Research was supported by the Centre for Advanced Studies at IBM Toronto Laboratory and the University of Michigan.
†Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA.
‡Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, USA.
§IBM Toronto Laboratory, North York, Ontario, M3C 1H7, Canada.

A typical statistics-collection method, which is used in many DBMS products, such as DB2, Oracle, Informix and Sybase, is to periodically invoke a utility that collects and updates statistics about the underlying database[9, 12, 14, 15, 19]. We term this method as *the utility method*. The major disadvantages of this method are:

- *Heavy system load.* The utility competes for system resources with other components in a DBMS and, therefore, significantly increases the system load.
- *Out-of-date statistics.* To avoid a heavy system load, the utility cannot be invoked very often. As a result, out-of-date statistics may be used by the query optimizer quite often, and this leads to inefficient query processing.
- *Incomplete statistics* Some statistics, such as the costs of user-defined functions in recent object-relational DBMSs, cannot be collected by the utility method. Users are required to manually update such statistics in the system catalog.
- *High cost for analyzing large databases.* Unless a user[1] specifies a data subset to analyze, which is perforce a subjective assessment, the utility typically analyzes the whole database. Clearly, analyzing statistics for a large database is very expensive and unnecessary for data that never changes.
- *Inconvenience for users.* Users have to manually invoke the utility whenever significant changes have been made to the database. Otherwise, obsolete statistics may be used by the query optimizer.

Due to the problems described above, it is quite common for database users to hardly ever invoke the utility to update the database statistics progressively once the database is initialized. As a result, system performance may become progressively worse as the data changes and evolves.

Recently, sampling techniques have been incorporated into the utility method in some commercial DBMSs[14, 19]. The idea is to use sampled data to estimate statistics instead of analyzing a complete data set. This approach mitigates some of the above disadvantages. However, the problems are not completely eliminated. For instance, the overhead for analyzing statistics for the whole database is still significant, some statistics are still not obtainable, and users still need to manually invoke the utility. Furthermore, sampling itself may involve accessing a significant part of the database.

Another common method for collecting statistics, so-called the dynamic query optimization method[1, 21, 22], or simply the *dynamic method*, is to obtain up-to-date statistics by analyzing the result or intermediate result of a user query during its processing. The up-to-date statistics can be used not only to dynamically improve the processing of the current query but also to optimize subsequent queries. This method provides an approach to obtain up-to-date statistical information without significantly increasing the system load. However, a shortcoming of this method is that not all useful statistics can be updated. Consider the following example.

Example 1 A user issues the following query on a DBMS:

Q_1:
```
SELECT R1.a2
FROM   R1
WHERE R1.a3 IN ( SELECT R2.b1
                 FROM   R2     ) ;
```

[1] A database administrator (DBA) is considered as one type of user in this paper.

where $R_1(a_1,\ a_2,\ a_3)$ and $R_2(b_1,\ b_2)$ are two tables in the underlying database. Note that the equivalent relational algebra expression[2], which will be used in the discussions below, for query Q_1 is:

$$\pi_{R_1.a_2}(R_1 \underset{R_1.a_3=R_2.b_1}{\bowtie} (\pi_{R_2.b_1}(R_2))$$

where π and $\bowtie$ denote the project and join operations, respectively. One feasible access plan to execute Q_1 is performing the following subquery first:

$Q_1^{(2)}$: SELECT $R_2.b_1$
FROM R_2

(i.e., $\pi_{R_2.b_1}(R_2)$), then performing the outer query block by using the result of subquery $Q_1^{(2)}$. Clearly, the DBMS can collect and update statistics about the column $R_2.b_1$ by analyzing the result of $Q_1^{(2)}$ during the query processing. However, statistics about $R_2.b_2$ cannot be obtained during the query processing since it is not referenced by $Q_1^{(2)}$. ∎

To meet the performance requirement of modern information processing, a DBMS should employ a statistics-collection method that (1) collects up-to-date statistics for all pertinent data; (2) incurs as low an overhead as possible; and (3) reduces the user's burden of invoking a utility manually. In this paper, we propose such a new approach, called the piggyback query optimization method, or simply *the piggyback method*. The key idea of the piggyback method is to collect statistics by piggybacking some additional retrievals during the processing of a user query in order to improve the quality and quantity of collected statistics. Although these side retrievals are not related to the processing of the given query (and may slow down the query processing slightly), the statistics collected from the results of the side retrievals can be used to improve the processing of subsequent queries.

A comprehensive survey on existing techniques for statistics collection and cost estimation in DBMSs was given by Mannino *et al.* in [13]. They divided statistics into base statistics (profile) for the tables in a database, intermediate statistics for the intermediate result tables during query processing, and target statistics for the final result table of a query. They discussed the relationships between statistics and cost estimates in query processing. In [4, 11, 23], Zander, Copeland, and Mackert *et al.* proposed a number of techniques to estimate physical statistics (e.g., page references) for data and index files in a database. In [6], Haas *et al.* introduced several sampling-based estimators to estimate the number of distinct valus of a column in a table. In [3, 10, 16, 17], Christodoulakis, Selinger, Shapiro, and Lipton *et al.* proposed various techniques including parametric methods, table-based methods, and sampling methods to estimate the intermediate and target table sizes from some base statistics. Gardy *et al.* studied the evolution of the table size statistic under queries and updates in [5]. They showed that the statistic behaves asymptotically as Gaussian processes. In [21], Yu *et al.* suggested a number of dynamic (adaptive) query optimization techniques and classified them into direct ones, which dynamically optimize the current query based on runtime information, and indirect ones, which collect dynamic information from the current query to optimize subsequent quries. Yu *et al.* proposed an adaptive query optimization algorithm to dynamically complete a partial access plan based on latest statistics collected at runtime in [22]. The piggybacking idea for collecting/updating statistics was first mentioned in our previous work [25] for a multidatabase environment, but technical details for the approach were not studied. In this paper, we explore the piggyback technique in detail and study how it can be incorporated into a DBMS. To our knowledge, there is no similar work that has been reported in the literature.

[2] There may be an inessential difference about duplicates in the result table.

The rest of the paper is organized as follows. Section 2 defines basic piggybacking operators and introduces different types of piggybacking. Section 3 studies the statistics obtainable from different access methods via piggyback analysis. Section 4 suggests several piggybacking levels to meet users' different requirements for overhead. Section 5 discusses some other related issues. Section 6 concludes the paper.

2 Types of Piggybacking

There are several types of piggybacking, which will be discussed in the following subsections. Among them, vertical piggybacking and horizontal piggybacking form the basic ones. Using the basic types of piggybacking, more complex forms of piggybacking may be generated.

2.1 Vertical Piggybacking

Vertical piggybacking includes extra columns in query processing. Consider the following example.

Example 2 To obtain statistics about column $R_2.b_2$ in Example 1, the DBMS can perform a slightly different subquery:

$Q_1^{(2)'}$: SELECT $R_2.b_1$, $R_2.b_2$
FROM R_2 ;

(i.e., $\pi_{R_2.b_1,R_2.b_2}(R_2)$) on the underlying database instead of $Q_1^{(2)}$ and analyze its result. Since both $Q_1^{(2)}$ and $Q_1^{(2)'}$ usually scan table R_2 once (and, effectively, also access $R_2.b_2$), $Q_1^{(2)'}$ increases the processing cost of query Q_1 only slightly. In fact, we have piggybacked the side retrieval:

Q_1': SELECT $R_2.b_2$
FROM R_2 ;

(i.e., $\pi_{R_2.b_2}(R_2)$) during the processing of user query Q_1 to obtain necessary statistics with a slightly additional overhead. ∎

In general, consider a user query Q with operand tables $R_1, R_2, ..., R_n$. To process Q, a DBMS usually performs a subquery[3]

$$Q^{(i)} = \pi_{CL_i}(\sigma_{F_i}(R_i)) \tag{1}$$

on each table R_i $(1 \leq i \leq n)$ to retrieve data that is required to process Q, where σ denotes the select operation in the relational algebra. We call $Q^{(i)}$ an *access subquery* for table R_i. For instance, for Q_1 in Example 1, two access subqueries are

$$Q_1^{(1)} = \pi_{R_1.a_2,R_1.a_3}(\sigma_{'true'}(R_1))$$

and

$$Q_1^{(2)} = \pi_{R_2.b_1}(\sigma_{'true'}(R_2)).$$

Using intermediate results from the access subqueries, the DBMS can evaluate query Q. Hence,

$$Q = F(Q^{(1)}, Q^{(2)}, ..., Q^{(n)}), \tag{2}$$

[3]Note that a DBMS may execute such a subquery with other operations via pipelining. We also assume that the columns referenced in qualification F_i but not needed in the further processing of the query are included in project target list CL_i, which simplifies the definition of horizontal piggybacking in the next subsection.

where $F(...)$ is a query formula that generates the result of Q by using the results of $Q^{(1)}, Q^{(2)}, ..., Q^{(n)}$. For instance, for Q_1 in Example 1,

$$F(Q^{(1)},\ Q^{(2)}) \ = \ \pi_{Q^{(1)}.a2}(Q^{(1)} \underset{Q^{(1)}.a3=Q^{(2)}.b1}{\bowtie} Q^{(2)}) \ .$$

Let ω be an operator such that

$$\omega_X(Q^{(i)}) = \pi_{CL_i \ \cup \ X}(\sigma_{F_i}(R_i)),$$

where X is a set of columns in R_i. $\omega_X(Q^{(i)})$ is called a *vertically-piggybacked subquery* for $Q^{(i)}$ with a piggybacked column set X, and ω is called the *vertical piggybacking operator*. If $X - CL_i \neq \emptyset$, $\omega_X(Q^{(i)})$ is *non-trivial*. Otherwise, it is *trivial*. For example,

$$\omega_{R2.b2}(Q_1^{(2)}) = \pi_{R_2.b_1, R2.b2}(\sigma_{'true'}(R_2)).$$

is a non-trivial vertically-piggybacked subquery in Example 1.

Let ω^{-1} be an operator such that

$$\omega_X^{-1}(\omega_X(Q^{(i)})) \ = \ \pi_{CL'_i \ - \ X}(\omega_X(Q^{(i)})) = Q^{(i)}.$$

where $CL'_i = CL_i \ \cup \ X$. ω^{-1} is called the *inverse vertical piggybacking operator* of ω.

For a given query $Q \ = \ F(Q^{(1)},\ Q^{(2)},\ ...,\ Q^{(n)})$ in (2), the following query with ω and ω^{-1} applied to its access subqueries

$$Q' \ = \ F(\omega_{X_1}^{-1}(\omega_{X_1}(Q^{(1)})),\ \omega_{X_2}^{-1}(\omega_{X_2}(Q^{(2)})),\ ...,\ \omega_{X_n}^{-1}(\omega_{X_n}(Q^{(n)})))$$

is called a *vertically-piggybacked query* for Q. If at least one vertically-piggybacked subquery $\omega_{X_i}(Q^{(i)})$ $(1 \leq i \leq n)$ is non-trivial, Q' is non-trivial. Otherwise, it is trivial. Note that, unlike a subquery and its vertically-piggybacked counterpart, the original query Q has the same result as its vertically-piggybacked counterpart Q', i.e., $Q \ \equiv \ Q'$. However, processing the vertically-piggybacked query may generate more intermediate results (see Figure 1), and these can be used to produce useful statistics that improve the quality of query optimization.

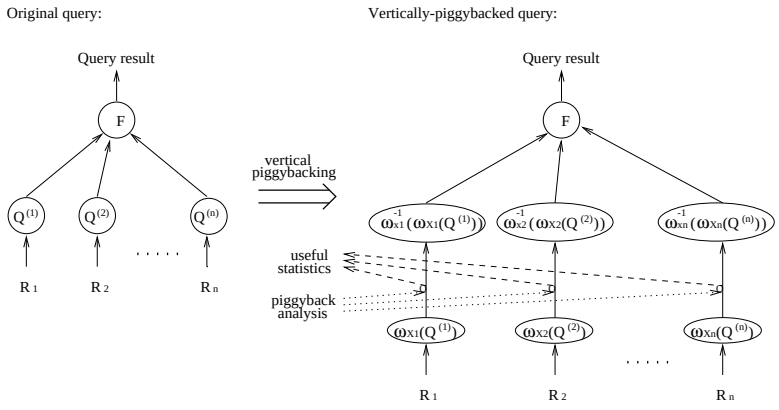

Figure 1: Idea of Vertical Piggybacking

Note that, at the low level in a DBMS, a complete tuple is usually fetched from the data file into a buffer although only part of the tuple may be needed for processing the given query. Hence piggybacking

side retrievals for extra columns from a table during the processing of a user query usually does not incur additional I/O cost. However, the piggyback analysis on intermediate results to obtain useful statistics requires some additional CPU time. Although CPU time is relatively small compared with I/O cost, it should be kept as low as possible. To achieve this goal, we propose to allow including extra columns for piggyback analysis with different degrees, depending on the user's constraint to piggyback overhead. The more the overhead allowed, the more number of extra columns may be included in piggyback analysis.

With regard to different degrees of vertical piggybacking, we divide the columns of an operand table R_i for a given query Q into the following four classes:

$$\begin{aligned}
AC_1 &= \{\ x \mid x \textit{ is a column in } R_i \ \wedge\ x \textit{ is referenced in } Q\ \},\\
AC_2 &= \{\ x \mid x \textit{ is an indexed column in } R_i\ \} - AC_1,\\
AC_3 &= \{\ x \mid x \textit{ is a column in } R_i \ \wedge\ (\ x \textit{ is part of the primary key}\\
&\quad \vee\ x \textit{ is part of a foreign key} \ \vee\ x \textit{ is referenced by a foreign key}\ )\ \} - AC_2,\\
AC_4 &= \{\ x \mid x \textit{ is a column in } R_i\ \} - AC_3.
\end{aligned}$$

The principle for including piggybacked columns is to include those columns that are more likely to be referenced by user queries. Since the columns in AC_1 are known to be referenced in at least one user query, they have the highest priority to be included in piggyback analysis. Since an index on a column indicates that users intend to use the column in their queries quite often, the columns in AC_2 have the next higher priority to be included in piggyback analysis. The next preferred class of columns are those related to primary and/or foreign keys, i.e., those in class AC_3. We term such a column as a *key-related* column. The remaining columns are in class AC_4, which may be included for piggyback analysis if the user's requirement on piggyback overhead is not high.

Let

$$X_k = \cup_{j=1}^{k}\ AC_j, \qquad (1 \le k \le 4).$$

The following subquery

$$\omega_{X_k}(Q^{(i)}) = \pi_{CL_i \cup X_k}(\sigma_{F_i}(R_i))$$

is called the vertically-piggybacked subquery *at level* i. A vertically-piggybacked query Q is at level k if at least one of its vertically-piggybacked subqueries is at level k. However, to simplify implementation, we recommend having all vertically-piggybacked subqueries in Q at the same level, and this is assumed in the rest of this paper.

Note that there exist some queries whose results may be obtained completely from the referenced indexes without accessing the data files of operand tables. Many DBMSs support such an "index-only access" method. In this case, a complete tuple may not be available in memory during query processing. Therefore, including extra columns in piggyback analysis may incur additional I/O cost. To deal with such a situation, class AC_2 can be further divided into two subclasses for given query Q:

$$\begin{aligned}
AC_{2\,1} &= \{\ x \mid x \textit{ is a column in } R_i \textit{ with an index referenced in } Q\ \} - AC_1,\\
AC_{2\,2} &= \{\ x \mid x \textit{ is an indexed column in } R_i\ \} - AC_{2\,1}.
\end{aligned}$$

An index is said to be referenced in a query if the query references at least one column on which the index is built. The values of for the columns in subclass $AC_{2\,1}$ can be obtained from the referenced indexes when they are fetched into memory during query processing. Since the columns in subclass $AC_{2\,2}$ only have non-referenced indexes, to obtain their values requires additional I/O cost. Hence, $AC_{2\,1}$ is assigned a higher priority than $AC_{2\,2}$.

2.2 Horizontal Piggybacking

Horizontal piggybacking includes extra rows in query processing. Consider the following example.

Example 3 For the following user query:

Q_2: SELECT $R_1.a_1$, $R_1.a_2$
FROM R_1
WHERE $R_1.a_1 > 3$;

(i.e., $\pi_{R_1.a_1,\ R_1.a_2}(\sigma_{R_1.a_1 > 3}(R_1))$ where $R_1.a_1$ is indexed while $R_1.a_2$ is not, a feasible access plan is to retrieve the qualified rows from R_1 via the index on $R_1.a_1$. Several statistics on $R_1.a_1$ can be obtained via analyzing information contained in the index if the index, which is relatively small compared with the data file, is completely fetched into memory. However, statistics on $R_1.a_2$ may not be accurately known since not all data values of $R_1.a_2$ are obtained (e.g., if it is a sparse index). An improvement can be made by using all rows in the retrieved pages from the data file as sample rows[4] to estimate the statistics on $R_1.a_2$. Although it is possible that some rows in the retrieved pages are not qualified for the query, they may be usable to improve the accuracy of statistical estimates. Since the rows in a retrieved page are available in memory, such horizontal piggybacking may provide better statistics without incurring much additional cost. Furthermore, additional random sample pages that contain no qualified rows may also be fetched into memory to obtain even better statistics since the randomness and size of the sample set are improved. Trade-offs need to be made between the overhead and accuracy. ∎

In general, consider a user query Q with operand tables $R_1, R_2, ..., R_n$. Let $Q^{(i)}$ be the access subquery for table R_i $(1 \leq i \leq n)$ as in (1), and $Q = F(Q^{(1)}, Q^{(2)}, ..., Q^{(n)})$ as in (2).

Let γ be an operator such that

$$\gamma_Y(Q^{(i)}) = \pi_{CL_i}(\sigma_{F_i \vee Y}(R_i))$$

where Y is a qualification condition for the extra rows to be retrieved from table R_i. $\gamma_Y(Q^{(i)})$ is called a *horizontally-piggybacked subquery* with a piggyback qualification condition Y, and γ is called the *horizontal piggybacking operator*. If

$$\{\ x \mid x\ satisfies\ Y \ \wedge\ \neg(x\ satisfies\ F_i)\ \} \neq \emptyset,$$

then $\gamma_Y(Q^{(i)})$ is *non-trivial*. Otherwise, it is *trivial*.

Let γ^{-1} be an operator such that

$$\gamma_Y^{-1}(\gamma_Y(Q^{(i)})) = \sigma_{F_i}(\gamma_Y(Q^{(i)})) = Q^{(i)}.$$

γ^{-1} is called the *inverse horizontal piggybacking operator* of γ.

For a given query $Q = F(Q^{(1)}, Q^{(2)}, ..., Q^{(n)})$ in (2), the following query with γ and γ^{-1} applied to its access subqueries

$$Q'' = F(\gamma_{Y_1}^{-1}(\gamma_{Y_1}(Q^{(1)})),\ \gamma_{Y_2}^{-1}(\gamma_{Y_2}(Q^{(2)})),\ ...,\ \gamma_{Y_n}^{-1}(\gamma_{Y_n}(Q^{(n)})))$$

is called a *horizontally-piggybacked query* for Q. If at least one horizontally-piggybacked subquery $\gamma_{Y_i}(Q^{(i)})$ $(1 \leq i \leq n)$ is non-trivial, Q'' is non-trivial. Otherwise, it is trivial.

[4] Note that such a *convenience sampling technique* may yield a biased sample unless the convertional assumptions about the uniform distribution of column values and the independence of different columns hold. Biased samples are adopted in many applications although they may not lead to good statistical estimates.

To determine a horizontally-piggybacked subquery $\gamma_{Y_i}(Q^{(i)})$, we need to choose the piggyback qualification condition Y. Condition Y should be chosen in such a way that the extra rows can be used to derive good statistical estimates and the piggyback overhead is kept as low as possible.

At one extreme, $Y \Rightarrow F_i$, i.e., Y logically implies F_i. In this case, no extra rows will be fetched; that is, the horizontally-piggybacked subquery is trivial. Although there is no piggyback overhead in this case, statistics that we obtain may be poor. At the other extreme, $Y = `true'$; that is, all rows in the operand table will be fetched. Although we can obtain all statistics about the table in this case, the piggyback overhead may be high.

There are two useful cases between the two extreme cases. Let

$$RB = \{\ n \ \mid\ \exists y\ \ (y\ \in\ R_i\ \ \wedge\ \ F_{i\ |y} = `true'\ \ \wedge\ \ y.pid = n)\ \},$$

where $F_{i\ |y}$ denotes the truth value of condition F_i when instantiated by row y from table R_i, and $y.pid$ is the identification number[5] of the page that contains row y in the data file. In other words, RB is the set of identification numbers of retrieved pages. In the first case, we define the piggyback qualification condition Y on table R_i as follows

$$Y_{|x}\ =\ true \qquad if\ and\ only\ if \qquad x.pid\ \in\ RB, \tag{3}$$

where x is a row from table R_i. This condition qualifies all rows in the retrieved (file) pages for the query. Since the newly qualified rows are in the retrieved pages, including them in piggyback analysis does not incur any additional I/O cost. For example, the following is the horizontally-piggybacked subquery[6] in Example 3

$$\gamma_{R_1.pid\ \in\ RB}(Q_2) = \pi_{R_1.a_1, R_1.a_2}(\sigma_{R_1.a_1>3\ \ \vee\ \ R_1.pid\ \in\ RB}(R_1)). \tag{4}$$

In the second case, we take a small set SB of extra sample pages and define the piggyback qualification condition Y as follows:

$$Y_{|x}\ =\ true \qquad if\ and\ only\ if \qquad (x.pid\ \in\ RB\ \ \vee\ \ x.pid\ \in\ SB).$$

This condition qualifies not only the rows in the retrieved pages but also the rows in chosen sample pages. Retrieving rows in the sample pages requires some additional I/O cost. However, statistical estimates can be improved by using these additional sample rows. The sample pages can be chosen randomly from the data file.

2.3 Mixed vertical and horizontal piggybacking

Note that vertical piggybacking increases the quantity (number) of statistics to be collected/estimated, while horizontal piggybacking improves the quality (accuracy) of statistics to be estimated. Applying only vertical or horizontal piggybacking alone may be insufficient. More suitable may be the case in which a mixture of vertical and horizontal piggybacking is adopted. For example, the vertical piggybacking operator can be applied to the horizontally-piggybacked subquery in (4) to yield a mixed vertical and horizontal piggybacked subquery:

$$\omega_{R_1.a_3}(\gamma_{R_1.pid\ \in\ RB}(Q_2)) = \pi_{R_1.a_1, R_1.a_2, R_1.a_3}(\sigma_{R_1.a_1>3\ \ \vee\ \ R_1.pid\ \in\ RB}(R_1)),$$

which can be used to estimate statistics on column $R_1.a_3$ in addition to columns $R_1.a_1$ and $R_1.a_2$.

[5] *pid* can be considered as an implicit attribute (column) of the operand table stored in a database.

[6] By convention, a table name is used as a tuple variable in the qualification condition of a relational algebra expression.

For a general query $Q = F(Q^{(1)}, Q^{(2)}, ..., Q^{(n)})$, the following is a mixed vertical and horizontal piggybacked query:

$$\begin{aligned} Q^* = \; & F(\omega_{X_1}^{-1}(\gamma_{Y_1}^{-1}(\omega_{X_1}(\gamma_{Y_1}(Q^{(1)})))), \;\; \omega_{X_2}^{-1}(\gamma_{Y_2}^{-1}(\omega_{X_2}(\gamma_{Y_2}(Q^{(2)})))), \\ & ....., \;\; \omega_{X_n}^{-1}(\gamma_{Y_n}^{-1}(\omega_{X_n}(\gamma_{Y_n}(Q^{(n)}))))). \end{aligned}$$

Piggyback analysis is to be performed after γ and ω are applied and before γ^{-1} and ω^{-1} are applied.

In fact, piggyback analysis and query processing can be done in a pipelined manner, i.e., the complete result of a piggybacked subquery is not required before the query is further processed.

2.4 Multi-Query Piggybacking

From Section 2.2, we know that the data values obtained via the horizontal piggybacking can be used as a sample set for statistical analysis. Sample pages in addition to the retrieved pages in a data file may be used to improve the properties (randomness and size) of a sample set. However, fetching extra sample pages requires additional overhead.

To improve the properties of a sample set without incurring much additional overhead, another type of piggybacking, called *multi-query piggybacking*, may be utilized. The idea is to use the retrieved data from multiple queries as one sample set. In this way, the sample size is increased. The randomness of the sample set is also improved if user queries are assumed to be independent, which is true in many cases. Furthermore, piggyback analysis may be performed on a query-by-query base (i.e., via pipelining) without waiting for the final sample set to be formed. For example, consider the average length of a column, which is a statistic used in query optimization. Let S_i be the set of values for column $R.a$ retrieved by query Q_i $(1 \leq i \leq n)$. Clearly, the total length $len(S_i)$ and total number $|S_i|$ of values in sample subset S_i can be obtained via analyzing S_i alone without other S_j where $j \neq i$. The average length $avg\_len(R.a)$ of column $R.a$ for the final sample set $S = \cup_{i=1}^{n} S_i$ can be calculated as follows:

$$avg\_len(R.a) = \frac{\sum_{i=1}^{n} len(S_i)}{\sum_{i=1}^{n} |S_i|}.$$

Sample subset S_i can be discarded after intermediate statistics $len(S_i)$ and $|S_i|$ are obtained.

Note that the mixed vertical and horizontal piggybacking is a general type of piggybacking for a single query, while the multi-query piggybacking, which combines multiple individual piggybacked queries, is even more general. The vertical and horizontal piggybacking operators are the basic building blocks for all piggybacked queries.

3 Statistics Obtainable via Piggyback Analysis

3.1 Statistics for Query Optimization

Different DBMSs may maintain different types of statistics in their catalogs for query optimization. Table 1 shows the typical statistics maintained in a DBMS. There are some other statistics that are not shown in the table, which are mainly used by a DBA to monitor the system performance and decide how to reconfigure and/or reorganize the database.

The statistics for query optimization can be classified into logical and physical types:

$$\begin{cases} \textit{Logical statistics}: & C_1,\ C_2,\ C_3,\ C_4,\ C_5,\ T_1,\ I_3,\ I_4 \\ \textit{Physical statistics}: & T_2,\ T_3,\ T_4,\ I_1,\ I_2,\ I_5,\ I_6,\ I_7 \end{cases}$$

Type	Label	Description
Column Statistics	C_1	max value of a column (or second max value)
	C_2	min value of a column (or second min value)
	C_3	number of distinct values of a column
	C_4	distribution (frequent values and quantiles)
	C_5	average column length
Table Statistics	T_1	number of rows in a table
	T_2	number of pages used by a table
	T_3	percentage of active rows compressed for a table
	T_4	number of overflow rows
Index Statistics	I_1	number of leaf pages
	I_2	number of B-tree index levels
	I_3	number of distinct values for the 1st column of index key
	I_4	number of distinct values for the full index key
	I_5	percentage of rows in the clustered order
	I_6	average number of leaf pages per index value
	I_7	average number of data pages per index value

Table 1: Typical Statistics Maintained in a Catalog

The logical statistics can be determined by the data values in a database, while physical ones are determined by the properties of physical organizations for the database on a storage medium.

3.2 Statistics obtainable from access methods

In a DBMS, a user query is implemented by one or more access methods such as the sequential scan method and the hash join method. In principle, the access methods involving more than one table can be implemented by the ones involving a single table, i.e., the ones used for access subqueries. We hence mainly consider unary access methods, and the common ones are:

$$\begin{cases} \textit{Sequential scan } (SS): & \textit{scan the rows in a table sequentially} \\ \textit{Index scan } (IS): & \textit{retrieve the qualified rows via an index} \\ \textit{Index} - \textit{only access } (IOA): & \textit{get all requested values from an index tree} \\ \textit{Hash access } (HA): & \textit{retrieve the qualified rows via a hash table} \end{cases}$$

Statistics can be obtained during the execution of an access method although not all statistics are obtainable via every method. In fact, statistical information is obtained at different levels of accuracy. At the top level, an accurate statistic is obtained via calculation using a complete data set. At the second level, an estimated statistic is obtained via sampled data. At the next level, validity information on a statistic rather than the statistic itself is obtained. In other words, we only know whether the statistic is up-to-date or not. At the lowest level, no statistic can be obtained. Table 2 shows what statistics may be obtained (at what level) during the execution of different access methods.

			C_1	C_2	C_3	C_4	C_5	T_1	T_2	T_3	T_4	I_1	I_2	I_3	I_4	I_5	I_6	I_7
SS			$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
IS	(I)	full	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
	(II)	$a < \beta_1,\ a > \beta_2$	$\surd$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
	(III)	$a < \beta_1$	$\odot$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
	(IV)	$a > \beta_2$	$\surd$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
	(V)	partial	$\odot$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
IOA	(I)	full	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\times$
	(II)	$a < \beta_1,\ a > \beta_2$	$\surd$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\times$	$\times$	$\times$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\times$
	(III)	$a < \beta_1$	$\odot$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\times$	$\times$	$\times$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\times$
	(IV)	$a > \beta_2$	$\surd$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\times$	$\times$	$\times$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\times$
	(V)	partial	$\odot$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\times$	$\times$	$\times$	$\oplus$	$\surd$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\times$
HA			$\odot$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

'$\surd$' — accurate statistics; '$\oplus$' — estimated statistics via sampling; '$\odot$' — validity information of statistics;
'$\times$' — no obtainable/applicable statistics; 'a' — an indexed column; 'β_1', 'β_2' — constants;

Table 2: Statistics Obtainable via Access Methods

For the sequential scan method, since the whole data file of a table is scanned, all column and table statistics can be accurately calculated during its execution. However, since indexes are not accessed, no index statistics can be obtained.

For the index scan method, there are several cases. *(I)* The first case occurs when an index tree is used as a means to scan the whole corresponding table in the sorted order of the indexed column. Since both the index and the table are fully scanned, all statistics can be obtained. *(II)* The second case occurs when the retrieved values of an indexed column a from the index tree cover at least both range $a < \beta_1$ and range $a > \beta_2$, where β_1 and β_2 are constants. Since $a < \beta_1$ (if not empty) implies $min(a)$ is retrieved and $a > \beta_2$ (if not empty) implies $max(a)$ is retrieved, both the maximum and minimum statistics (C_1 and C_2) can be obtained accurately. Since the index tree is accessed, statistic I_2 can also be obtained accurately. Other statistics can be estimated by using the set of retrieved data as a set of sample data. For example, statistic C_3 for column a in table R can be estimated as:

$$C_3 = |R| * \frac{n(a)}{|S|},$$

where $|R|$, $n(a)$, and $|S|$ are the cardinality of R, the number of distinct values in the sample set, and the cardinality of the sample set, respectively. Note that a sample set can be improved by applying horizontal piggybacking or multi-query piggybacking. *(III)* The third case occurs when the retrieved values of an indexed column cover range $a < \beta_1$ but not range $a > \beta_2$. In this case, statistic C_2 can be obtained accurately, but not statistic C_1. For C_1, the following condition[7] can be used to check if current C_1 is out-of-date:

$$(\exists x \in S \; such \; that \; x > C_1) \quad \Rightarrow \quad (C_1 \; is \; out-of-date), \tag{5}$$

where S is the set of retrieved values and '$\Rightarrow$' denotes the logic implication. The obtainability of other statistics is the same as that in Case (II). *(IV)* The fourth case occurs when the retrieved values of an indexed column cover range $a > \beta_2$ but not range $a < \beta_1$. In this case, statistic C_1 can be obtained accurately, but not statistic C_2. For C_2, the following condition can be used to check if current C_2 is out-of-date:

$$(\exists x \in S \; such \; that \; x < C_2) \quad \Rightarrow \quad (C_2 \; is \; out-of-date). \tag{6}$$

The obtainability of other statistics is the same as that in Case (II). *(V)* The last case occurs when it is unknown whether the retrieved values of a column (indexed or not) contain maximum or minimum. For example, it is not known what condition is satisfied for the values of a non-indexed column that are obtained from the rows retrieved via the indexed column. In this case, statistics C_1 and C_2 can be validated by using the conditions (5) and (6), respectively. The obtainability of other statistics is the same as that in Case (II).

For the index-only access method, the obtainability of statistics is similar to the index scan method. The only difference is that no statistics related to the data file of a table or the table itself can be obtained since the data file is not accessed. Only the statistics about the referenced index(es) (excluding the related data file) and the corresponding indexed column(s) can be obtained, estimated or validated. Note that table statistic T_1 may be obtained or estimated when the referenced index(es) is a unique index.

For the hash access method, conditions (5) and (6) can be used to validate statistics C_1 and C_2, respectively. Other column statistics and most table statistics can be estimated by taking data in the hit bucket(s) of the hash file as a set of sample data. It is clear that no index statistics can be obtained.

[7] Unfortunately, it is only a sufficient condition for an out-of-date statistic, not a necessary and sufficient condition.

Note that there is another class of statistics — the ones for user-defined functions, such as the number of I/O's required for execution of such a function. This type of statistics are supported in recent object-relational DBMSs They are not listed in Table 1. It is easy to see that this type of statistics cannot be collected or estimated via scanning the data objects in a database. Therefore, they cannot be updated by using the utility method. The current solution adopted in commercial products to this problem is to ask users manually update such statistics in the catalog by using update commands[2, 7]. However, such statistics can be obtained via piggyback analysis during the execution of a query that invokes the user-defined function(s). For example, the measured elapse time of a user-defined function can be used as an estimate of the cost statistic for the function. This is another advantage of the piggyback method.

4 Piggybacking Levels

Piggyback analysis can be performed at different levels. At one end of the spectrum, no piggyback analysis is performed during query processing. Statistics for query optimization can be collected by applying the utility method. There is no piggyback overhead in this case. However, as mentioned in Section 1, there are a number of serious drawbacks with this approach. At the other end of the spectrum, a full piggyback analysis is performed during query processing, i.e., all pertinent statistics related to the accessed data objects are obtained. A full piggyback analysis usually requires significant overhead since all relevant data (no matter qualified or not for the query) for the accessed data objects is retrieved. As pointed out before, it is possible to perform piggyback analysis at some level such that useful statistics are obtained with slightly additional cost because unqualified data may exist in the retrieved pages of a data file.

Our goal is to obtain as many statistics as possible within a given tolerance of piggyback overhead. The more the overhead allowed, the more statistics, together with accuracy, may be obtained. To achieve this goal, we define six levels of piggybacking. In general, the higher the piggybacking level, the more and better statistics may be obtained — however, more overhead may also be incurred.

Level 0: No piggyback analysis is performed during query processing.

Level 1: Frequencies of tables/indexes accessed by queries are recorded, and the validity of relevant statistics is checked during query processing.

Level 2: Statistics on the accessed index(es), column(s) and table(s) are collected and/or estimated during query processing.

Level 3: Statistics at level 2 as well as those on other indexed columns in a referenced table are collected and/or estimated during query processing.

Level 4: Statistics at level 3 as well as those on other key-related columns in a referenced table are collected and/or estimated during query processing.

Level 5: Statistics at level 4 as well as those on the remaining columns in a referenced table are collected and/or estimated during query processing.

At piggybacking level 0, the piggybacking option of the system is disabled by a user. Since no piggyback analysis is performed during query processing, there is no overhead. Statistics have to be collected by invoking the statistics collecting utility manually.

At piggybacking level 1, no statistics are directly collected or estimated during query processing. Only the access frequency of each data object (table or index) is counted during query processing. The set S of

most frequently accessed data objects is identified. Clearly, it is important to keep the statistics on data objects in S up-to-date because they are used frequently to optimize user queries. In fact, there is no need to update statistics for the data objects that are never accessed by users. On the other hand, it is possible that the statistics on a data object x in S is already up-to-date. In this case, there is no need to update the statistics on x. To determine which data object whose statistics are out-of-date, the validity of statistics is checked by inspecting the retrieved data during query processing. Let W be the set of data objects whose statistics are found to be out-of-date. Then the set $V = S \cap W$ contains the data objects whose statistics should be updated. The statistics collecting utility is now invoked to collect statistics only for data objects in V. Since the statistics collecting utility is not invoked for all data objects indiscriminately or for some data objects selected subjectively, the utility is used more effectively. Furthermore, the utility can be automatically invoked by the system without user's interference once set V is found. Hence the user's burden of manually invoking the utility is relieved. Figure 2 shows such a useful lightweight piggybacking procedure.

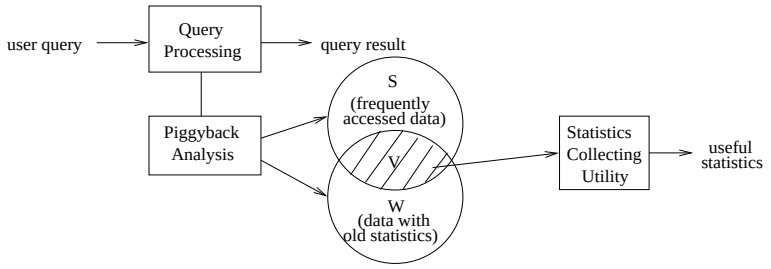

Figure 2: Useful Lightweight Piggybacking

At piggybacking level 2, statistics are collected and/or estimated for the data objects (index(es), column(s) and table(s)) that are referenced in a query. These are, in fact, the statistics that can be obtained via traditional dynamic query optimization.

At piggybacking levels $3-5$, statistics on extra columns that are included by the vertically-piggybacked queries at levels 2 to 4 (respectively, see Section 2.1) are collected and/or estimated.

Note that piggybacking levels $2-5$ can be further divided into sublevels by using horizontally piggybacked queries. At the first sublevel, all data in the retrieved pages from a file are used in piggyback analysis during query processing. At the second sublevel, data in additional sample pages are used in piggyback analysis. At the third level, data from multiple queries are combined and utilized by the piggyback statistical analysis. Usually, the higher the sublevel is, the higher the quality of collected statistics would be.

For the sublevel 2 at piggybacking level 5, if all pages in relevant data and index files are chosen as sample pages, it actually is the full piggybacking. Such a full piggybacking may not be practically feasible due to its high overhead. A proper piggybacking level in a system can be determined according to a user-specified tolerance of overhead.

5 Related Issues

Initial statistics

The piggyback method collects statistics during query processing. A basic question is how to get the initial statistics for processing the first query. There are several possible solutions to this problem.

One solution is to use the default values for statistics. The piggyback method provides the system with a self-tuning capability. In other words, the system will generate increasingly better access plans since more accurate statistics are collected or estimated by the piggyback method as the database is used. A shortcoming with this approach is that the system may produce poor access plans at the initial stage.

A better way to solve the problem is to perform heuristic-based query optimization initially. After the system has collected sufficient statistics by using the piggyback method, it starts to optimize queries based on cost analysis. Unfortunately, not all DBMSs supports both heuristic-based and cost-based query optimization.

Another approach to solve the problem is to use the utility method to get the initial statistics. The piggyback method is then used to maintain up-to-date statistics automatically.

Piggybacking time

There is no need to perform piggyback analysis for every user query, since statistics may not be out-of-date and are not required to be exact either. However, query performance may degrade severely if statistics are far from accurate. A question arises as to when is the proper time to perform piggyback analysis.

There are several possible ways to solve this problem. The simplest way is to perform piggyback statistical analysis periodically, e.g., every 100 queries. This method may not be effective since the change of statistics may not follow a fixed pattern. Another way is to let the DBA to manually enable and disable the piggybacking option at proper times. The drawback of this method is that the user's burden is not relieved.

A better way is to let the system automatically activate and deactivate the piggybacking option based on the system performance and load. When the performance of a query become worse, it has a reason to believe that the relevant statistics may not be up-to-date. Piggybacking at a proper level can be activated. The heavier the current system load, the lower the piggybacking level should be activated. If the piggybacking level 1 is activated, the statistics collecting utility can be invoked automatically for the identified data objects when the system load is not heavy.

Alternatively, the piggybacking level 1 can be set as the default level. When a significant amount of statistics are found to be out-of-date during query processing, either the piggybacking level is raised or the statistics collecting utility is invoked to update statistics. When it is found that not many statistics need to be updated for a long time, the piggybacking option for the system can be deactivated automatically.

Note that the piggyback method may not completely replace the utility method. They can complement each other. The piggyback method usually reduces the invocation frequency of the statistics collecting utility and makes the utility automatically collect statistics only for frequently accessed data objects, while the utility method can be used to collect statistics that cannot (or too expensive) be collected via piggyback analysis. Using them effectively can greatly improve the system efficiency and reduce user's burden.

Parallelism in piggybacking

Typically, the statistical information obtained via piggybacking is not used for processing the current query. Hence, query processing and piggyback analysis can be performed in parallel. In this way, system performance and resource utilization can be improved. In a uniprocessor system, the piggyback analysis task can be run in background. In a multiprocessor system, the query processing task and the piggyback analysis task can be run on separate processors in parallel. Note that since statistical information needs not be exact, locks on data items can be released once query processing is done. There is no need to hold locks for piggyback analysis. Due to the limitation of the paper length, how to perform parallel piggyback analysis

and synchronize parallel tasks will be discussed in a separate paper.

6 Conclusion

In today's database applications, data tends to change very frequently. Existing techniques for maintaining statistics on data cannot meet this demand. Out-of-date statistics may lead to poor system performance. On the other hand, it is not convenient for a user to manually update statistics as required in most current commercial DBMSs. To solve these problems, a new *piggyback method* for collecting query optimization statistics is proposed in this paper.

The key idea of the piggyback method is to ride some side retrievals piggyback on the processing of a user query. Statistics obtained via analyzing the results of both the given query and the side retrievals are used to update the system catalog. The improved statistics can be utilized to optimize subsequent queries.

Several types of piggybacking have been introduced in this paper. Vertical piggybacking, which retrieves extra unrequested columns from an operand table, can increase the quantity of obtainable statistics; while horizontal piggybacking, which retrieves extra unrequested rows from an operand table, can improve the quality of obtained statistics. Typically, a mixed vertical and horizontal piggybacking should be employed to provide good statistics in terms of both quantity and quality. Multi-query piggybacking, which makes use of data retrieved by multiple (piggybacked) queries in piggyback analysis, can be used to provide higher quality statistics with low overhead. The basic vertical and horizontal piggybacking operators are defined to describe different types of piggybacked queries.

Piggyback analysis can be performed at different levels in a system. The higher the piggybacking level is, the more the piggyback overhead although more statistics could be obtained. A proper piggybacking level can be chosen according to the system load and the user-specified tolerance of piggyback overhead. A useful lightweight piggyback analysis is to count the access frequencies of data objects and check the validity of their statistics without actually updating the statistics during query processing. The statistics collection utility is then automatically invoked for the identified data objects that are frequently accessed by user queries and whose statistics are out-of-date. Such an integration of the piggyback method and the utility method provides an effective solution to the problem of collecting statistics for query optimization in a DBMS. The discussion in the paper also shows that statistics can be obtained with different degrees of accuracy (i.e., accurately calculated, approximately estimated, or effectively validated) via piggyback analysis from different access methods.

The main advantages of the piggyback method are:

- *The user's burden for manually invoking a utility to update statistics is relieved*, since statistics are updated during query processing or automatic execution of the statistics collection utility. This advantage offers a great convenience to users.

- *The cost of maintaining statistics about rarely-used data is reduced*, since the piggyback method updates statistics only for the data accessed by or related to a user query. This advantage saves the time wasted by the utility method for maintaining useless statistics.

- *More useful statistical information is collected*, since extra unrequested data, which was not considered in the dynamic method, is considered and the user-defined functions, which were not handled in the utility method, are handled.

- *The changing of statistics is more smooth*, since statistics are updated more promptly. This advantage reduces the chance for the system to be jammed with the tasks of re-optimizing queries.

It is expected that a DBMS incorporating the piggyback method can better meet users' satisfaction in term of performance and convenience.

Acknowledgements

The authors would like to thank Peter Haas, John McPherson, David Ready, Enzo Cialini, Gabby Silberman, Weidong Kou, Per-Åke Larson, Alberto Mendelzon, Patrick Martin, Qi Cheng, and Roger Zheng for their valuable suggestions and comments for the work reported in this paper.

References

[1] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proceedings of the 9th International Conference on Data Engineering*, pages 538–47, Vienna, Austria, 1993.

[2] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann Publishers, Inc., 1996.

[3] S. Christodoulakis. Estimating record selectivities. *Information System*, 8(2):105–115, 1983.

[4] G. Copeland et al. Buffering schemes for permanent data. In *Proceedings of the Int'l Conf. on Data Engineering*, pages 214–21, Los Angeles, Calif., 1986.

[5] D. Gardy and G. Louchard. Dynamic analysis of some relational database parameters. *Theoretical Computer Science*, 144(1-2):125–59, 1995.

[6] P. J. Haas, J. F. Naughton, et al. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21st VLDB Conference*, pages 311–22, Zurich, Swizerland, 1995.

[7] IBM. *DB2 Universal Server Administration Guide Version 5.* International Business Machines Corporation, 1997.

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

[9] J. Kirkwood. *Sybase Architecture and Administration.* Ellis Horwood, 1993.

[10] R. J. Lipton and J. F. Naughton. Practical selectivity estimation through adaptive sampling. In *Proceedings of SIGMOD*, pages 1–11, 1990.

[11] L. Mackert and G. Lohman. Index scans using a finite LUR buffer: A validated i/o model. Technical Report RJ4836, IBM Almaden Research Center, 1985.

[12] C. Malamud. *INGRES Tools for Building an Information Architecture.* Van Nostrand Reinhold, 1989.

[13] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3), September 1988.

[14] J. McNally et al. *Informix Unleashed.* SAMS Publishing, 1997.

[15] C. S. Mullins. *DB2 Developer's Guide.* SAMS Publishing, 1994.

[16] P. G. Selinger and M. Adiba. Access path selection in distributed data base management systems. In *Proceedings of the International Conference on Data Bases*, pages 204–15, Aberdeen, Scotland, 1980.

[17] G. P. Shapiro and C. Connel. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of SIGMOD*, pages 256–76, 1984.

[18] W. Sun, W. Meng, and C. Yu. Query optimization in distributed object-oriented database systems. *Comput. J. (UK)*, 35(2):98–107, April 1992.

[19] E. Whalen. *Oracle Performance Tuning and Optimization*. SAMS Publishing, 1996.

[20] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, Dec. 1984.

[21] C. T. Yu, L. Lilien, K. C. Guh, M. Templeton, D. Brill, and A. Chen. Adaptive techniques for distributed query optimization. In *IEEE 1986 International Conference on Data Engineering*, pages 86–93, Los Angeles, USA, 1986.

[22] M. J. Yu and P. C.-Y. Sheu. Adptive query optimization in dynamic databases. *International Journal on Artificial Intelligence Tools*, 7(1):1–30, 1998.

[23] V. Zander, B. Taylor, et al. Estimating block accesses when attributes are correlated. In *Proceedings of the 12th Int'l Conf. on Very Large Databases*, pages 119–27, Kyoto, Japan, 1986.

[24] Qiang Zhu. Query optimization in multidatabase systems. In *Proceedings of the 1992 IBM CAS Conference, vol.II*, pages 111–27, Toronto, Canada, Nov. 1992.

[25] Qiang Zhu. An integrated method of estimating selectivities in a multidatabase system. In *Proceedings of the 1993 IBM CAS Conference*, pages 832–47, Toronto, Canada, Oct. 1993.

[26] Qiang Zhu and P.-Å. Larson. Global query processing and optimization in the CORDS multidatabase system. In *Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems*, pages 640–6, Dijon, France, Sept. 1996.